Александр Побегайло

bhv

Системное программирование в Windows

- Синхронизация потоков
- Каналы передачи данных и почтовые ящики
- Виртуальная память и файлы
- Асинхронная обработка данных
- Безопасность доступа к объектам

Наиболее полное руководство

+

В ПОДЛИННИКЕ®

Александр Побегайло

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ В WINDOWS

Санкт-Петербург

«БХВ-Петербург»

2006

УДК 681.3.06
ББК 32.973.26-018.1
П41

Побегайло А. П.

П41 Системное программирование в Windows. — СПб.: БХВ-Петербург, 2006. — 1056 с.: ил.

ISBN 5-94157-792-3

Подробно рассматриваются вопросы системного программирования с использованием интерфейса Win32 API. Описываются управление потоками и процессами, включая их диспетчеризацию; синхронизация потоков; передача данных между процессами, с использованием анонимных и именованных каналов, а также почтовых ящиков; структурная обработка исключений; управление виртуальной памятью; управление файлами и каталогами; асинхронная обработка данных; создание динамически подключаемых библиотек; разработка сервисов. Отдельная часть книги посвящена управлению безопасностью объектов в Windows. Каждая тема снабжена практическими примерами использования функций Win32 API, которые представлены работающими листингами. Это позволяет использовать книгу в качестве пособия по системному программированию или справочника для системного программиста. Прилагаемый компакт-диск содержит листинги и проекты всех программ, рассмотренных в книге.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Андрей Смышляев*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Наталия Першакова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 15.01.06.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 85,14.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Отпечатано с готовых диапозитивов
в ОАО "Техническая книга"
190005, Санкт-Петербург, Измайловский пр., 29.

ISBN 5-94157-792-3

Оглавление

Предисловие

Эта книга предназначена для начинающих системных программистов. Но, поскольку она содержит большой объем справочной информации по интерфейсу программирования приложений Win32 API (Application Programming Interface — интерфейс программирования приложений), то может использоваться и опытными программистами в качестве справочного пособия.

Начинающие системные программисты могут и не совсем ясно понимать, чем же отличается системное программирование от обычного (или прикладного) программирования, и, вообще-то, можно поговорить о том, кто такие системные программисты и чем они занимаются.

Очевидно, что как системные, так и прикладные программисты пишут программы. Чем же отличаются системные программы от прикладных? Ключом к ответу на этот вопрос является понятие системы, которое в различных прикладных областях знаний имеет разные определения. Если говорить понятным для программистов языком, то программная система это набор функций, при помощи которых можно решить любую задачу из некоторой предметной области.

Почему же системное программирование обычно ассоциируется с операционными системами и их интерфейсами для разработки программ? Так сложилось исторически. Первыми серьезными программными системами были именно операционные системы, поэтому и основные концепции системного программирования отрабатывались при разработке и реализации операционных систем. Затем эти технологии использовались при разработке других программных систем, таких как, например, системы управления базами данных (СУБД). Хотя в настоящее время наблюдается и обратное влияние. Классическое системное программирование рассматривает круг вопросов, связанных с синхронизацией и диспетчеризацией потоков и процессов, обменом данными между процессами, управлением устройствами компьютера и файлами. В последнее время большое внимание при проектировании систем также уделяется и обеспечению безопасности данных, что вызвано возросшими угрозами несанкционированного доступа к данным. Средства операционных

систем Windows, предназначенные для решения этих задач, исключая управление устройствами, и рассмотрены в этой книге.

Первоначально, материал, представленный в книге, был подготовлен как пособие для студентов, изучающих курс "Операционные системы". Как правило, продолжительность такого курса — 1 семестр, за который нужно разобрать основные концепции операционных систем, да еще выполнить лабораторные работы по данному курсу. Времени катастрофически не хватает даже на рассмотрение основных теоретических концепций и технических приемов, используемых при построении операционных систем. А тут еще надо и обучить студентов системному программированию. А студенты-то и прикладные программы еще не очень хорошо пишут, т. к. опыта программирования маловато. Хотя они и владеют языком программирования C (и в некоторой степени C++), но разрабатывать учебные проекты операционных систем при таком уровне знаний нереально, поэтому, как правило, лабораторные работы заключаются в разработке программ, решающих конкретные системные задачи. Для этого нужно использовать функции из интерфейса операционной системы, предназначенные для системного программирования. Объяснить на лекциях или семинарах назначение и работу этих функций невозможно из-за громоздкости материала и недостатка времени. В общем, нужно пособие по элементарным приемам системного программирования под Windows. После создания этого пособия на лекциях излагались только концептуальные теоретические и технические вопросы курса, а техника программирования изучалась студентами самостоятельно. Если же какие-то технические вопросы и возникали при программировании задач, то они, как правило, решались прямо на лабораторных занятиях. Это позволило разгрузить лекции от многих технических подробностей и облегчить концептуальное построение курса. Думаю, эти замечания, как и само пособие, окажутся полезными как преподавателям, так и студентам, изучающим курс "Операционные системы", используя для практической работы платформы Microsoft Windows.

После этого вступления становится понятной структура книги. Каждая глава посвящена отдельной теме, связанной с системным программированием под Windows. Чтобы иметь представление о задачах, которые решаются при помощи рассматриваемых функций, первый раздел или параграф каждой главы содержит основные теоретические моменты, относящиеся к данной тематике. После этого рассматриваются функции из интерфейса Windows, предназначенные для решения системных задач из данной области, и приведены примеры использования этих функций. Все примеры настолько элементарны, насколько это возможно. Поэтому можно надеяться, что они будут понятны начинающим программистам.

Все представленные материалы готовились довольно продолжительное время. Поэтому изложение может показаться неровным. Но переработка такого объема информации также займет продолжительное время. Поэтому я ре-

шил оставить все как есть. Все программы были протестированы на платформе операционной системы Windows 2000, используя среду разработки Microsoft Visual Studio 6.0. Думаю, что в настоящее время именно эта среда и используется в вузах при обучении. За исключением нескольких незначительных моментов, связанных с изменением типов параметров в прототипах функций, проблем с переходом на среду разработки Microsoft Visual Studio 7.0 (.NET) быть не должно. Хотя все программы и были протестированы, но не исключены ошибки, которые могли возникнуть при форматировании текста. Но, поскольку программы очень простые, то устранение этих ошибок не должно вызвать затруднений.

Теперь о двух вопросах, которые могут возникнуть при рассмотрении программ.

Первый вопрос касается стиля программирования, а именно — проверки значения переменной, которая чаще всего содержит дескриптор объекта, на равенство величине `NULL`. С одной стороны символическая константа `NULL` по стандартам языков программирования C и C++ — это ноль. Поэтому значение переменной в этом случае можно рассматривать просто как логическое значение. С другой стороны, в примерах из MSDN дескриптор объекта проверяется на равенство `NULL` посредством оператора сравнения `==`. По-видимому, это обусловлено тем, что символическая константа `NULL` также определена и в интерфейсе прикладного программирования Win32 API, который не стандартизирован. Естественно, что в Win32 API эта константа также определена как ноль. В программах, приведенных в книге, принят такой же подход, как и в примерах из MSDN (справочная система).

Второй вопрос связан с вводом/выводом на консоль. В программах для этих целей используются функции из заголовочных файлов stdio.h, iostream.h и conio.h. Я считаю, что программист должен одинаково хорошо знать стандартные функции языков программирования C и C++, поэтому использование функций из заголовочных файлов stdio.h и iostream.h не обсуждается. Что касается функций из заголовочного файла conio.h, то они полезны в том случае, если стандартные потоки ввода/вывода перенаправляются в файлы или анонимные каналы. В этом случае для тестирования программ приходится использовать функции из заголовочного файла conio.h, т. к. они всегда работают с консолью. Кроме того, по-видимому, эти функции имеют более простую реализацию, поэтому при их использовании не возникает проблем, связанных с синхронизацией ввода/вывода на консоль. Учитывая вышесказанное, думаю программистам полезно также познакомиться и с этими функциями.

Автор

Глава 1

Операционные системы и их интерфейсы

1.1. Назначение операционной системы

Физическими или *аппаратными ресурсами компьютера* называются физические устройства, из которых состоит компьютер. К таким устройствам относятся центральный процессор, оперативная память, внешняя память, шины передачи данных и различные устройства ввода-вывода информации. *Логическими* или *информационными ресурсами компьютера* называются данные и программы, которые хранятся в памяти компьютера. Когда говорят обо всех ресурсах компьютера, включая как физические, так и логические ресурсы, то обычно используют термины *ресурсы компьютера* или *системные ресурсы*.

Для выполнения на компьютере какой-либо программы необходимо, чтобы она имела доступ к ресурсам компьютера. Этот доступ обеспечивает операционная система. Можно сказать, что *операционная система* — это комплекс программ, который обеспечивает доступ к ресурсам компьютера и управляет ими. Другими словами, операционная система — это администратор или менеджер ресурсов компьютера. Назначение операционной системы состоит в обеспечении пользователя программными средствами для использования ресурсов компьютера и эффективном разделении этих ресурсов между пользователями. Отсюда следует, что главными функциями операционной системы являются управление ресурсами компьютера и диспетчеризация или планирование этих ресурсов.

1.2. Типы операционных систем

Все программы, которые работают на компьютере под управлением операционной системы, называются *пользовательскими программами*. Совокупность пользовательских программ, которая предназначена для решения определенной задачи, называется *приложением*. Если операционная система одновременно

может исполнять только одну пользовательскую программу, то она называется *однопрограммной* или *однопользовательской*. Если же под управлением операционной системы могут одновременно выполняться несколько пользовательских программ, то такая операционная система называется *мультипрограммной* или *многопользовательской*.

В зависимости от назначения операционной системы и аппаратуры компьютера, на котором она работает, можно определить несколько типов операционных систем. Если операционная система может работать только на компьютере с одним процессором, то такая операционная система называется *однопроцессорной*. Если же операционная система может работать также и на компьютере, который содержит несколько процессоров, то такая операционная система называется *мультипроцессорной*.

Следует делать различие между операционными системами, которые предназначены для обработки информации под управлением пользователя, и операционными системами, которые предназначены для управления объектами при помощи компьютера в реальном времени без участия пользователя. Такими объектами могут быть, например, робот или самолет. Операционная система, предназначенная для работы в режиме реального времени, называется *операционной системой реального времени*. Главное отличие операционных систем реального времени заключается в их быстром реагировании на внешние события и надежности функционирования. Если пользователь, сидя у компьютера, будет только раздражен медленной или ненадежной работой операционной системы, то медленная или ненадежная работа операционной системы реального времени может вызвать поломку оборудования и аварию.

В дальнейшем будут рассматриваться только операционные системы фирмы Microsoft, а именно Windows 98 и Windows 2000, которые предназначены для использования на персональных компьютерах. Эти операционные системы отличаются своей внутренней организацией, но используют один и тот же интерфейс для программирования приложений — Win32 API. Мы не будем рассматривать операционную систему Windows CE, которая предназначена для использования в таких различных устройствах, как, например, устройства бытовой электроники, контроллеры для управления технологическими процессами и устройства управления коммуникационным оборудованием. Но, разобравшись в изложенном материале, вы получите опыт, который поможет вам как в изучении Windows CE, так и других операционных систем.

Относительно операционной системы Windows XP можно сказать следующее. Те приемы системного программирования, которые рассмотрены в этой книге для операционной системы Windows 2000, также работают и в операционной системе Windows XP.

1.3. Интерфейс программирования приложений Win32 API

Интерфейс программирования приложений Win32 API представляет собой набор функций и классов, которые используются для программирования приложений, работающих под управлением операционных систем фирмы Microsoft. Следует отметить, что в работе многих функций Win32 API существуют различия, которые зависят от типа операционной системы. Кроме того, некоторые функции работают только в операционной системе Windows 2000 и не поддерживаются операционной системой Windows 98. Все эти случаи будут отмечаться отдельно. Но все же в работе функций Win32 API в разных версиях операционных систем гораздо больше общего, чем различий. Поэтому чаще всего мы будем говорить, что функции Win32 API предназначены для разработки приложений на платформах операционных систем Windows, не делая различия между операционными системами Windows 98 и Windows 2000. Это соглашение значительно облегчит изложение материала, не загромождая его ненужными подробностями, которые отвлекают от сути рассматриваемых вопросов.

Функционально Win32 API подразделяется на следующие категории:

- Base Services (базовые сервисы);
- Common Control Library (библиотека общих элементов управления);
- Graphics Device Interface (интерфейс графических устройств);
- Network Services (сетевые сервисы);
- User Interface (интерфейс пользователя);
- Windows NT Access Control (управление доступом для Windows NT);
- Windows Shell (оболочка Windows);
- Windows System Information (информация о системе Windows).

Кратко опишем функции, которые выполняются в рамках этих категорий. Функции базовых сервисов обеспечивают приложениям доступ к ресурсам компьютера. Категория Common Control Library содержит классы окон, которые часто используются в приложениях. Интерфейс графических устройств обеспечивает функции для вывода графики на дисплей, принтер и другие графические устройства. Сетевые сервисы используются при работе компьютеров в компьютерных сетях. Интерфейс пользователя обеспечивает функции для взаимодействия пользователя с приложением, используя окна для ввода-вывода информации. Категория Windows NT Access Control содержит функции, которые используются для защиты информации путем контроля и ограничения доступа к защищаемым объектам. Категории Windows Shell и Windows System Information содержат соответственно функции для работы с оболочкой и конфигурацией операционной системы Windows.

В курсе системного программирования главным образом изучается назначение и использование функций из категорий Base Services и Windows NT Access Control. Функции из категорий Common Control Library, Graphics Device Interface и User Interface используются для разработки интерфейса приложений, а курс, который изучает назначение и использование этих функций, как правило, называется "Программирование пользовательских интерфейсов в Windows". Изучив два этих курса и добавив сюда свои знания по программированию на языке C++, вы получите довольно содержательное представление о разработке приложений на платформе Win32 API.

В связи с тем, что программирование графических пользовательских интерфейсов в Windows само по себе является довольно трудоемким занятием, мы будем изучать функции ядра Windows, работая только с консольными приложениями. Это упростит изложение предмета и избавит нас от большого количества кода, не относящегося к существу рассматриваемых вопросов.

1.4. Типы данных в Win32 API

Прежде всего заметим, что интерфейс программирования приложений Win32 API ориентирован на язык программирования C или, в более широком смысле, на процедурные языки программирования. Поэтому в этом интерфейсе, не используются такие возможности языка программирования C++, как классы, ссылки и механизм обработки исключений.

Чтобы сделать интерфейс Win32 API более независимым от конкретного языка программирования или, может быть, более соответствующим аппаратному обеспечению компьютера, разработчики этого интерфейса определили новые простые типы данных. Эти типы данных используются в прототипах функций интерфейса Win32 API.

Новые простые типы данных определены как синонимы простых типов данных языка программирования C. Чтобы отличать эти типы от других типов, их имена определены прописными буквами. Общее количество простых типов данных, определенных в интерфейсе Win32 API, довольно велико. Поэтому ниже приведены определения только тех простых типов данных из этого интерфейса, которые очевидным образом переименовывают простые типы данных языка программирования C.

```
typedef char CHAR;

typedef unsigned char  UCHAR;
typedef UCHAR          *PUCHAR;
typedef unsigned char  BYTE;
typedef BYTE           *PBYTE;
typedef BYTE           *LPBYTE;
```

```
typedef short SHORT;

typedef unsigned short  USHORT;
typedef USHORT          *PUSHORT;
typedef unsigned short  WORD;
typedef WORD            *PWORD;
typedef WORD            *LPWORD;

typedef int   INT;
typedef int   *PINT;
typedef int   *LPINT;
typedef int   BOOL;
typedef BOOL  *PBOOL;
typedef BOOL  *LPBOOL;

typedef unsigned int  UINT;
typedef unsigned int  *PUINT;

typedef long LONG;
typedef long   *LPLONG;

typedef unsigned long  ULONG;
typedef ULONG           *PULONG;
typedef unsigned long   DWORD;
typedef DWORD           *PDWORD;
typedef DWORD           *LPDWORD;

typedef float  FLOAT;
typedef FLOAT  *PFLOAT;

typedef void   *LPVOID;
typedef CONST void  *LPCVOID;
```

Остальные простые типы данных, определенные в интерфейсе Win32 API, имеют, как правило, специфическое назначение и поэтому они будут описаны при их использовании.

Кроме того, в интерфейсе Win32 API определены символические константы `FALSE` и `TRUE` для обозначения соответственно ложного и истинного логических значений. Определения этих констант приведены ниже.

```
#ifndef FALSE
#define FALSE  0
```

```
#endif

#ifndef TRUE
#define TRUE  1
#endif
```

В интерфейсе Win32 API также определено множество сложных типов данных, таких как структуры и перечисления. Как правило, эти типы данных имеют специфическое назначение и поэтому будут описаны при их непосредственном использовании.

1.5. Объекты и их дескрипторы в Windows

Объектом в Windows называется структура данных, которая представляет системный ресурс. Таким ресурсом может быть, например, файл, канал, графический рисунок. Операционные системы Windows предоставляют приложению объекты трех категорий:

- User (объекты интерфейса пользователя);
- Graphics Device Interface (объекты интерфейса графических устройств);
- Kernel (объекты ядра).

Категория User включает объекты, которые используются приложением для интерфейса с пользователем. К таким объектам относятся, например, окна и курсоры. Категория Graphics Device Interface включает объекты, которые используются для вывода информации на графические устройства. К таким объектам относятся, например, кисти и перья. Категория Kernel включает объекты ядра операционной системы Windows. К таким объектам относятся, например, файлы и каналы. При изучении системного программирования подробно рассматриваются только объекты категории Kernel. Объекты двух оставшихся категорий рассматриваются при изучении программирования графических интерфейсов.

Под доступом к объектам понимается возможность приложения выполнять над объектом некоторые функции. Приложение не имеет прямого доступа к объектам, а обращается к ним косвенно. Для этого в операционных системах Windows каждому объекту ставится в соответствие дескриптор (handle). В Win32 API дескриптор имеет тип `HANDLE`. *Дескриптор объекта* представляет собой запись в таблице, которая поддерживается системой и содержит адрес объекта и средства для идентификации типа объекта. Дескрипторы объектов создаются операционной системой и возвращаются функциями Win32 API, которые создают объекты. За редким исключением, эти функции имеют вид `CreateObject`, где слово `Object` заменяется именем конкретного объекта. Например, процесс создается при помощи вызова функции `CreateProcess`. Как правило, такие функции возвращают дескриптор соз-

данного объекта. Если это значение не равно `NULL` (или отрицательному значению), то объект создан успешно.

После завершения работы с объектом его дескриптор нужно закрыть, используя функцию `CloseHandle`, которая имеет следующий прототип:

```
BOOL  CloseHandle(
  HANDLE  hObject    // дескриптор объекта
);
```

При успешном завершении функция `CloseHandle` возвращает ненулевое значение, в противном случае — `FALSE`. Функция `CloseHandle` удаляет дескриптор объекта, но сам объект удаляется не всегда. Дело в том, что в Windows на один и тот же объект могут ссылаться несколько дескрипторов, которые создаются другими функциями для доступа к уже созданному ранее объекту. Функция `CloseHandle` уничтожает объект только в том случае, если на него больше не ссылается ни один дескриптор.

Часть I

Управление потоками и процессами

Глава 2

Потоки и процессы

2.1. Определение потока

Определение потока тесно связано с последовательностью действий процессора во время исполнения программы. Исполняя программу, процессор последовательно выполняет инструкции программы, иногда осуществляя переходы в зависимости от некоторых условий. Такая последовательность выполнения инструкций программы называется *потоком управления* внутри программы. Отметим, что поток управления зависит от начального состояния переменных, которые используются в программе. В общем случае различные исходные данные порождают различные потоки управления. Поток управления можно представить как нить в программе, на которую нанизаны инструкции, выполняемые микропроцессором. Поэтому часто поток управления также называется *нитью* (thread). В русскоязычной литературе за потоком управления закрепилось название *поток*. Для пояснения понятия потока рассмотрим следующую программу, которая выводит минимальное число из двух целых чисел или сообщение о том, что числа равны.

```
#include <iostream.h>
int main()
{
  int a, b;

  cout << "Input two integers: ";
  cin >> a >> b;
  if (a == b)
  {
    cout << "There is no min." << endl;
    return 0;
  }
```

```
  if (a < b)
    cout << "min = " << a << endl;
  else
    cout << "min = " << b << endl;
  return 0;
}
```

Предположим, что перегруженные операторы ввода-вывода не образуют новых потоков. Тогда в зависимости от входных данных эта программа образует один из трех возможных потоков управления. А именно, если выполняется условие `(a == b)`, то образуется поток:

```
  cout << "Input two integers: ";
  cin >> a >> b;
  if (a == b)
  {
    cout << "There is no min." << endl;
    return 0;
  }
```

Если выполняется условие `(a < b)`, то образуется поток:

```
  cout << "Input two integers: ";
  cin >> a >> b;
  if (a == b)
  if (a < b)
    cout << "min = " << a << endl;
  return 0;
```

Если же выполняется условие `(a > b)`, то образуется поток

```
  cout << "Input two integers: ";
  cin >> a >> b;
  if (a == b)
  if (a < b)
    cout << "min = " << b << endl;
  return 0;
```

Теперь перейдем к классификации программ в зависимости от количества определяемых ими параллельных потоков управления. Будем говорить, что программа является *многопоточной*, если в ней может одновременно существовать несколько потоков. Сами потоки в этом случае называются *параллельными*. Если в программе одновременно может существовать только один поток, то такая программа называется *однопоточной*. Например, сле-

дующая программа, которая просто вычисляет сумму двух чисел, является однопоточной:

```
#include <iostream.h>
int sum(int a, int b)
{
  return a + b;
}
int main()
{
  int a, b;
  int c = 0;
  cout << "Input two integers: ";
  cin >> a >> b;
  c = sum(a, b);
  cout << "Sum = " << c << endl;
  return 0;
}
```

Теперь предположим, что после вызова функции `sum` функция `main` не ждет возвращения значения из функции `sum`, а продолжает выполняться. В этом случае получим программу, состоящую из двух потоков, один из которых определяется функцией `main`, а второй — функцией `sum`. Причем эти потоки независимы, т. к. они не имеют доступа к общим или, другими словами, разделяемым переменным. Правда в этом случае не гарантируется, что поток `main` выведет сумму чисел `a` и `b`, т. к. инструкция вывода значения суммы может отработать раньше, чем поток `sum` вычислит эту сумму.

Из этих рассуждений видно, что для того чтобы отметить функцию, которая порождает новый поток в программе, должна использоваться специальная нотация. В операционных системах Windows для обозначения того, что функция образует поток, используются специальные спецификаторы функции. Такая функция обычно также называется потоком.

2.2. Контекст потока

В общем случае содержимое памяти, к которой поток имеет доступ во время своего исполнения, называется *контекстом потока*. Определим, каким ограничениям на доступ к памяти должны удовлетворять функции, чтобы их можно было безопасно вызывать в параллельных потоках. Для этого рассмотрим следующую функцию:

```
int f(int n)
{
```

```
  if (n > 0)
    --n;
  if (n < 0)
    ++n;
  return n;
}
```

Сколько бы раз эта функция не вызывалась параллельно работающими потоками, она будет корректно изменять значение переменной `n`, т. к. эта переменная является локальной в функции `f`. То есть для каждого нового вызова функции `f` будет создан новый локальный экземпляр переменной `n`. Такая функция `f` называется *безопасной для потоков*. Теперь введем глобальную переменную `n` и изменим нашу функцию следующим образом:

```
int  n;
void g()
{
  if (n > 0)
    --n;
  if (n < 0)
    ++n;
}
```

В этом случае параллельный вызов функции `g` несколькими потоками может дать некорректное изменение значения переменной `n`, т. к. значение этой переменной будет изменяться одновременно несколькими функциями `g`. В этом случае функция `g` не является безопасной для потоков.

Та же проблема встречается и в случае, когда функция использует статические переменные. Для разбора этого случая рассмотрим функцию

```
int count()
{
        static int n = 0;
        ++n;
        return n;
}
```

которая возвращает количество своих вызовов. Если эта функция будет вызвана несколькими параллельно исполняемыми потоками, то нельзя точно определить значение переменной `n`, которое вернет эта функция, т. к. это значение изменяется всеми потоками параллельно.

В общем случае функция называется *повторно входимой* или *реентерабельной* (reentrant или reenterable), если она удовлетворяет следующим требованиям:

- не использует глобальные переменные, значения которых изменяются параллельно исполняемыми потоками;

- не использует статические переменные, определенные внутри функции;
- не возвращает указатель на статические данные, определенные внутри функции.

В системном программировании часто также рассматриваются программы в кодах микропроцессора, выполнение которых может прерываться и возобновляться в любой момент времени. Причем одна и та же программа может запускаться прежде, чем завершилось исполнение предыдущего экземпляра этой программы. В этом случае также необходимо, чтобы программный код допускал корректное параллельное выполнение нескольких экземпляров программы. Это условие обеспечивается в том случае, если программа не изменяет свой код во время исполнения. Здесь под кодом подразумеваются как команды, так и данные, принадлежащие программе. Программа в кодах микропроцессора, которая не изменяет свой код, также называется *реентерабельной*.

В дополнение к реентерабельным функциям определяют также функции, безопасные для вызова параллельно исполняемыми потоками. Функция называется *безопасной для потоков*, если она обеспечивает блокировку доступа к ресурсам, которые она использует. Как обеспечить блокирование доступа к ресурсам, рассматривается в *гл. 6, 7*, посвященных синхронизации потоков. Сейчас же только скажем, что в этом случае решается задача взаимного исключения доступа к разделяемым ресурсам, используя примитивы синхронизации.

Очевидно, что если функция не является реентерабельной, то она также не является и безопасной для потоков, т. к. в этом случае несколько потоков разделяют общую память, не блокируя доступ к ней. А память, как уже говорилось, также является системным ресурсом.

2.3. Состояния потока

Как видно из определения, поток описывает динамическое поведение всей программы или какой-либо функции в программе. Для удобства обозначений предположим, что программа является однопоточной. Тогда поток можно рассматривать как пару:

$$\text{поток} = (\text{процессор, программа}).$$

Программа может исполняться процессором только в том случае, если она готова к исполнению. То есть все системные ресурсы, которые необходимы для исполнения этой программы, свободны для использования. Кроме того, для исполнения программы необходимо, чтобы и сам процессор был свободен и готов к исполнению этой программы. Для более формального описания этих ситуаций вводятся понятия "состояние процессора" и "состояние

программы". При этом предполагают, что процессор и программа могут находиться в следующих состояниях.

- Состояния процессора:
  - процессор не выделен для исполнения программы;
  - процессор выделен для исполнения программы.
- Состояния программы:
  - программа не готова к исполнению процессором;
  - программа готова к исполнению процессором.

Для краткости записи введем для этих состояний следующие названия:

- Состояния процессора:
  - "не выделен";
  - "выделен".
- Состояния программы:
  - "не готова";
  - "готова".

Тогда мы можем определить *состояние потока* как пару состояний:

состояние потока = (состояние процессора, состояние программы).

Перечислив различные комбинации состояний процессора и программы, можно описать все возможные состояния потока. Введем для состояний потока следующие названия:

- поток блокирован = ("не выделен", "не готова");
- поток готов к выполнению = ("не выделен", "готова");
- поток выполняется = ("выделен", "готова");

Будем считать, что состояние ("выделен", "не готова") является недостижимым для потока. То есть программе, не готовой к исполнению, процессор не выделяется. Более кратко эти состояния потока будем просто обозначать словами: "блокирован", "готов" и "выполняется". Для полноты картины нужно ввести для потоков еще два состояния: "новый" и "завершен", которые описывают соответственно поток, еще не начавший свою работу, и поток, завершивший свою работу. Тогда диаграмма возможных переходов потока из состояния в состояние может быть изображена, как это показано на рис. 2.1.

В результате мы получили простейшую диаграмму переходов потока из состояния в состояние. Сами переходы потока из состояния в состояние, которые на диаграмме обозначаются дугами, описывают некоторые операции

над потоком. Названия этих операций указаны рядом со стрелками. Кратко опишем эти операции.

- Операция `Create` выполняется потоком, который создает новый поток из функции. Эта операция переводит поток из состояния "новый" в состояние "готов".
- Операция `Exit` выполняется самим исполняемым потоком в случае его завершения. Эта операция переводит поток из состояния "выполняется" в состояние "завершен".

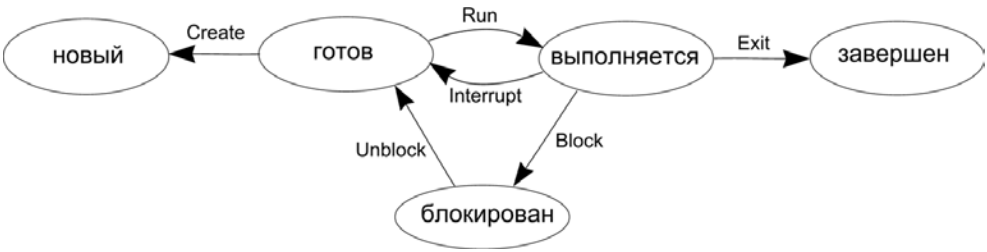


Рис. 2.1. Модель пяти состояний потока.

Оставшиеся четыре операции выполняются операционной системой.

- Операция `Run` запускает готовый поток на выполнение, т. е. выделяет ему процессорное время. Эта операция переводит поток из состояния "готов" в состояние "выполняется". Поток получает процессорное время в том случае, если подошла его очередь к процессору на обслуживание.
- Операция `Interrupt` задерживает исполнение потока и переводит его из состояния "выполняется" в состояние "готов". Эта операция выполняется над потоком в том случае, если истекло процессорное время, выделенное потоку на исполнение, или исполнение потока прервано по каким-либо другим причинам.
- Операция `Block` блокирует исполнение потока, т. е. переводит его из состояния "выполняется" в состояние "блокирован". Эта операция выполняется над потоком в том случае, если он ждет наступления некоторого события, например, завершения операции ввода-вывода или освобождения ресурса.
- Операция `Unblock` разблокирует поток, т. е. переводит его из состояния "блокирован" в состояние "готов". Эта операция выполняется над потоком в том случае, если событие, ожидаемое потоком, наступило.

Разрешим потокам также выполнять операции друг над другом. Для этого введем операции `Suspend` и `Resume`.

- Операция `Suspend` приостанавливает исполнение потока.
- Операция `Resume` возобновляет исполнение потока.

Используя эти операции, один поток может соответственно приостановить или возобновить исполнение другого потока независимо от того, в каком состоянии этот последний поток находится. Впрочем, заметим, что поток может приостановить и свое исполнение. Если над потоком выполнена операция `Suspend`, то будем говорить, что поток находится в *приостановленном* или *подвешенном состоянии*. Кратко будем говорить, что в этом случае поток "подвешен". Дополним диаграмму состояний потока, изображенную на рис. 2.1, этими новыми операциями и состояниями. Получим более полную диаграмму состояний потока, которая показана на рис. 2.2.

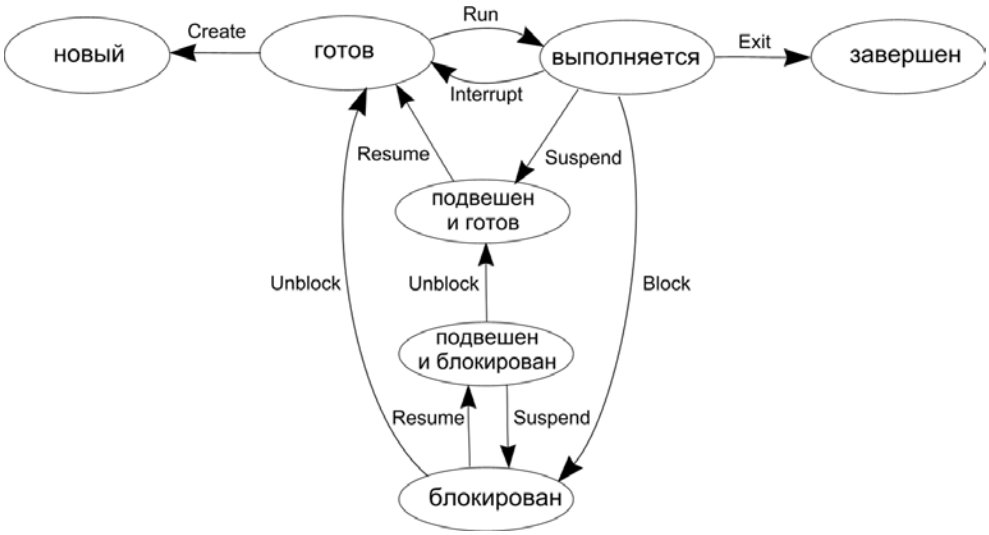


Рис. 2.2. Модель семи состояний потока

Теперь разрешим потоку выполнять операции над самим собой. Для этого введем операцию `Sleep`.

- Операция `Sleep` позволяет потоку приостановить свое исполнение на некоторый интервал времени или, другими словами, заснуть.

Разбудить поток должна операционная система по истечении заданного интервала времени, используя операцию `Wakeup`. Если поток выполнил операцию `Sleep`, то будем говорить, что он перешел в *сонное состояние* или "спит".

- Операция `Wakeup` позволяет операционной системе разбудить поток.

В результате можно построить полную диаграмму состояний потока, которая и приведена на рис. 2.3.

В заключение этого параграфа скажем, что в конкретных операционных системах для работы с потоками могут быть определены и другие состояния,

а также операции, которые переводят потоки в эти состояния. В *гл. 3* будет рассмотрена модель состояний потока в операционной системе Windows 2000.

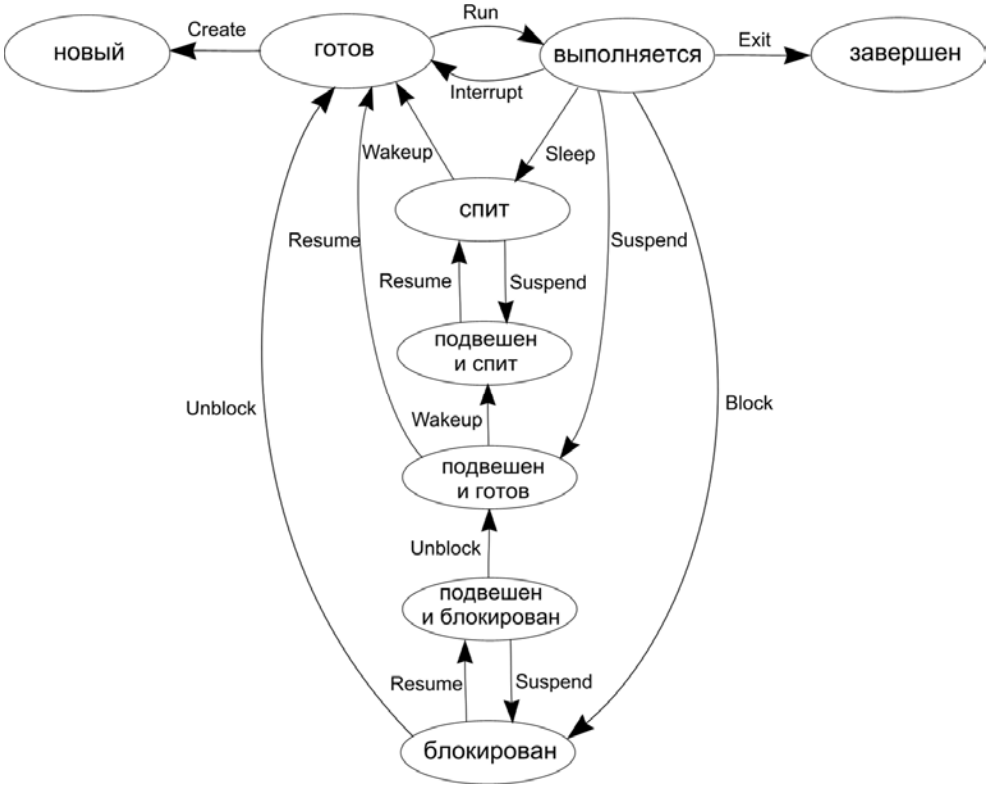


Рис. 2.3. Модель девяти состояний потока

2.4. Диспетчеризация и планирование потоков

В однопрограммной операционной системе одновременно может выполняться только один поток, которому доступны все ресурсы компьютера. Поэтому блокировка потока может происходить только в случаях ожидания этим потоком события, отмечающего завершение операций ввода-вывода. Недостатком однопрограммных операционных систем является их низкая производительность, т. к. процессор простаивает, если поток блокирован.

В мультипрограммных операционных системах одновременно могут существовать несколько потоков, что повышает производительность компьютера. Однако в этом случае требуется некоторая дисциплина обслуживания этих

потоков, смысл которой заключается в порядке выделения конкурирующим потокам ресурсов компьютера.

Для простоты дальнейшего изложения будем считать, что компьютер имеет только один процессор. Тогда общий подход к обслуживанию потоков в мультипрограммных операционных системах состоит в следующем. Время работы процессора делится на кванты (интервалы), которые выделяются потокам для работы. По истечении кванта времени исполнение потока прерывается и процессор назначается другому потоку. Распределением квантов времени между потоками занимается специальная программа, которая называется *менеджер потоков*.

Когда менеджер потоков переключает процессор на исполнение другого потока, он должен выполнить следующие действия:

- сохранить контекст прерываемого потока;
- восстановить контекст запускаемого потока на момент его прерывания;
- передать управление запускаемому потоку.

Контекст потока это содержимое памяти, с которой работает поток. Поэтому в каждый момент времени работы потока, его контекст полностью определяется содержимым регистров микропроцессора в этот момент времени. Отсюда следует, что для сохранения контекста потока необходимо сохранить содержимое регистров микропроцессора на момент прерывания потока, а при восстановлении контекста потока необходимо восстановить содержимое этих регистров.

Теперь кратко расскажем о сути алгоритмов управления потоками. Сначала предположим, что все потоки имеют одинаковый приоритет. Тогда они выстраиваются в одну очередь на обслуживание к процессору. Процессор обслуживает потоки в порядке FIFO (first in — first out), т. е. первым пришел — первым вышел, и прерванные потоки становятся в конец очереди. Такая дисциплина обслуживания называется *циклическим обслуживанием*. Так как незавершившиеся потоки блокируются до следующего обслуживания, а не уходят не обслуженными, то циклическое обслуживание также называется FCFS (first come − first served), т. е. первым пришел — первым обслужен. Схематически циклическое обслуживание потоков показано на рис. 2.4.

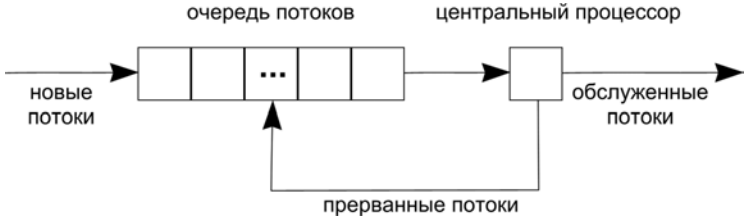


Рис. 2.4. Циклическое обслуживание потоков

Если потоки имеют разные приоритеты, то для управления ими используются более сложные дисциплины обслуживания с несколькими очередями. В этом случае каждая очередь включает потоки, которые имеют одинаковый приоритет. Схематически дисциплины обслуживания с несколькими очередями показаны на рис. 2.5.

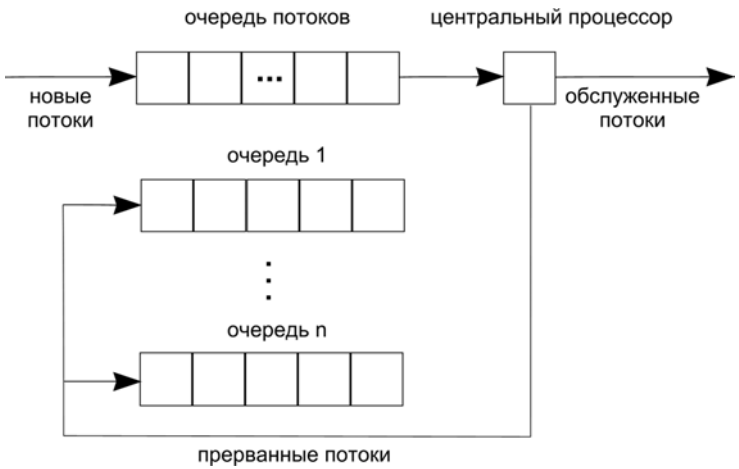


Рис. 2.5. Дисциплины обслуживания с несколькими очередями

Простейший алгоритм обслуживания нескольких очередей заключается в следующем: первыми обслуживаются потоки, которые имеют наивысший приоритет.

В общем случае управление потоками разделяется на планирование и диспетчеризацию. Под планированием потоков понимается алгоритм, используемый для постановки прерванных потоков в очереди. Менеджер потоков (диспетчер) может изменять приоритет прерванного потока, что изменяет очередь, в которую этот поток будет поставлен. Алгоритмы планирования изучаются математической дисциплиной, которая называется *теория расписаний*. Под диспетчеризацией потоков понимается алгоритм, устанавливающий порядок, в котором процессор обслуживает очереди. Алгоритмы диспетчеризации изучаются математической дисциплиной, которая называется *теория массового обслуживания*.

Алгоритмы управления потоками разрабатывают таким образом, чтобы оптимизировать следующие параметры системы:

- ❒ время загрузки микропроцессора работой должно быть максимальным;
- ❒ пропускная способность системы должна быть максимальной;
- ❒ время нахождения потока в системе должно быть минимальным;
- ❒ время ожидания потока в очереди должно быть минимальным;
- ❒ время реакции системы на обслуживание заявки должно быть минимальным.

При этом для каждой системы должен быть выбран оптимальный интервал обслуживания потоков, который снижает затраты на переключение контекстов потоков. В общем случае разделение времени работы процессора между потоками позволяет быстрее выполнять потоки, которые требуют немного времени на свое исполнение, но замедляет исполнение трудоемких потоков.

2.5. Определение процесса

Процессом или *задачей* называется исполняемое на компьютере приложение вместе со всеми ресурсами, которые требуются для его исполнения. Все ресурсы, необходимые для исполнения процесса, также называются *контекстом процесса*. Процессу обязательно принадлежат следующие ресурсы:

- адресное пространство процесса;
- потоки, исполняемые в контексте процесса.

Адресное пространство — это виртуальная память, выделенная процессу для запуска программ. Об устройстве виртуальной памяти будет рассказано в *гл. 20*. Адресные пространства разных процессов не пересекаются. Более того, процесс не имеет непосредственного доступа в адресное пространство другого процесса. Это позволяет избежать влияния ошибок, произошедших в каком-либо процессе, на исполнение других процессов, что повышает надежность системы в целом. Потоки, исполняемые в контексте процесса, запускаются в одном адресном пространстве, которое принадлежит этому процессу. В принципе, основной причиной, вызвавшей введение в системное программирование понятия потока, и было разделение адресных пространств процессов. Дело в том, что в этом случае взаимодействие между параллельными процессами требует больших затрат на пересылку данных, что заметно замедляет работу приложений. Потоки же выполняются в адресном пространстве одного процесса и, следовательно, могут обращаться к общим адресам памяти, что упрощает их взаимодействие.

Глава 3

Потоки в Windows

3.1. Определение потока

Потоком в Windows называется объект ядра, которому операционная система выделяет процессорное время для выполнения приложения. Каждому потоку принадлежат следующие ресурсы:

- код исполняемой функции;
- набор регистров процессора;
- стек для работы приложения;
- стек для работы операционной системы;
- маркер доступа, который содержит информацию для системы безопасности.

Все эти ресурсы образуют *контекст потока в Windows*. Кроме дескриптора каждый поток в Windows также имеет свой идентификатор, который уникален для потоков выполняющихся в системе. Идентификаторы потоков используются служебными программами, которые позволяют пользователям системы отслеживать работу потоков.

В операционных системах Windows различаются потоки двух типов:

- системные потоки;
- пользовательские потоки.

Системные потоки выполняют различные сервисы операционной системы и запускаются ядром операционной системы.

Пользовательские потоки служат для решения задач пользователя и запускаются приложением. На рис. 3.1 показана диаграмма состояний потока, работающего в среде операционной системе Windows 2000.

В работающем приложении различаются потоки двух типов:

- рабочие потоки (working threads);
- потоки интерфейса пользователя (user interface threads).

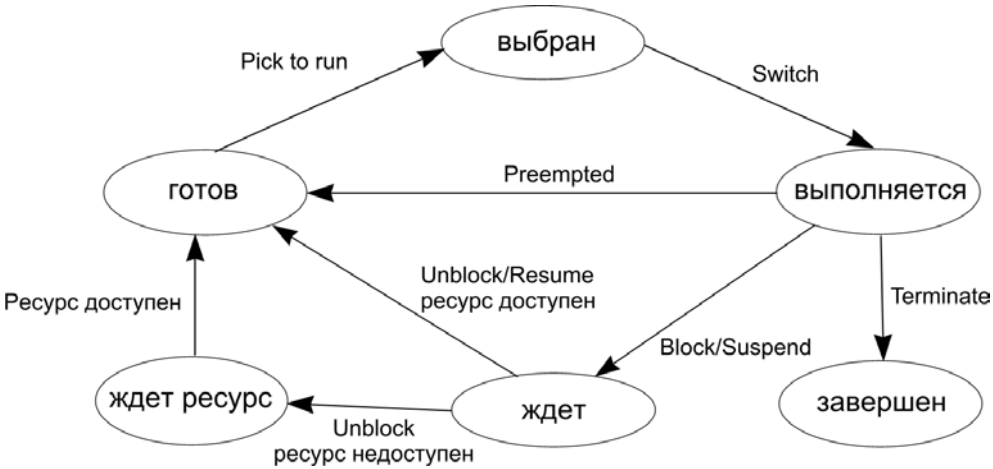


Рис. 3.1. Модель состояний потока в Windows 2000

Рабочие потоки выполняют различные фоновые задачи в приложении. *Потоки интерфейса пользователя* связаны с окнами и выполняют обработку сообщений, поступающих этим окнам. Каждое приложение имеет, по крайней мере, один поток, который называется *первичным* (primary) или *главным* (main) потоком. В консольных приложениях это поток, который исполняет функцию `main`. В приложениях с графическим интерфейсом это поток, который исполняет функцию `WinMain`.

3.2. Создание потоков

Создается поток функцией `CreateThread`, которая имеет следующий прототип:

```
HANDLE  CreateThread(
  LPSECURITY_ATTRIBUTES  lpThreadAttributes,  // атрибуты защиты
  DWORD    dwStackSize,          // размер стека потока в байтах
  LPTHREAD_START_ROUTINE  lpStartAddress,     // адрес функции
  LPVOID   lpParameter,          // адрес параметра
  DWORD    dwCreationFlags,      // флаги создания потока
  LPDWORD  lpThreadId            // идентификатор потока
);
```

При успешном завершении функция `CreateThread` возвращает дескриптор созданного потока и его идентификатор, который является уникальным для всей системы. В противном случае эта функция возвращает значение `NULL`. Кратко опишем назначение параметров функции `CreateThread`.

Параметр `lpThreadAttributes` устанавливает атрибуты защиты создаваемого потока. До тех пор пока мы не изучим систему безопасности в Windows, мы будем устанавливать значения этого параметра в `NULL` при вызове почти всех функций ядра Windows. В данном случае это означает, что операционная система сама установит атрибуты защиты потока, используя настройки по умолчанию. О процессах будет подробно рассказано в следующей главе.

Параметр `dwStackSize` определяет размер стека, который выделяется потоку при запуске. Если этот параметр равен нулю, то потоку выделяется стек, размер которого по умолчанию равен 1 Мбайт. Это наименьший размер стека, который может быть выделен потоку. Если величина параметра `dwStackSize` меньше значения, заданного по умолчанию, то все равно потоку выделяется стек размером в 1 Мбайт. Операционная система Windows округляет размер стека до одной страницы памяти, который обычно равен 4 Кбайт.

Параметр `lpStartAddress` указывает на исполняемую потоком функцию. Эта функция должна иметь следующий прототип:

```
DWORD  WINAPI  имя_функции_потока(LPVOID lpParameters);
```

Видно, что функции потока может быть передан единственный параметр `lpParameter`, который является указателем на пустой тип. Это ограничение следует из того, что функция потока вызывается операционной системой, а не прикладной программой. Программы операционной системы являются исполняемыми модулями и поэтому они должны вызывать только функции, сигнатура которых заранее определена. Поэтому для потоков определили самый простой список параметров, который содержит только указатель. Так как функции потоков вызываются операционной системой, то они также получили название *функции обратного вызова*.

Параметр `dwCreationFlags` определяет, в каком состоянии будет создан поток. Если значение этого параметра равно 0, то функция потока начинает выполняться сразу после создания потока. Если же значение этого параметра равно `CREATE_SUSPENDED`, то поток создается в подвешенном состоянии. В дальнейшем этот поток можно запустить вызовом функции `ResumeThread`.

Параметр `lpThreadId` является выходным, т. е. его значение устанавливает Windows. Этот параметр должен указывать на переменную, в которую Windows поместит идентификатор потока. Этот идентификатор уникален для всей системы и может в дальнейшем использоваться для ссылок на поток. Идентификатор потока главным образом используется системными функциями и редко функциями приложения. Действителен идентификатор потока только на время существования потока. После завершения потока тот же идентификатор может быть присвоен другому потоку. В операционной системе Windows 98 этот параметр не может быть равен `NULL`. В Windows NT и 2000 допускается установить его значение в `NULL` — тогда операционная система не возвратит идентификатор потока.

В листинге 3.1 приведен пример программы, которая использует функцию `CreateThread` для создания потока, и демонстрирует способ передачи параметров исполняемой потоком функции.

Листинг 3.1. Создание потока функцией `CreateThread`

```
#include <windows.h>
#include <iostream.h>

volatile int n;

DWORD WINAPI Add(LPVOID iNum)
{
  cout << "Thread is started." << endl;
  n += (int)iNum;
  cout << "Thread is finished." << endl;

  return 0;
}

int main()
{
  int  inc = 10;
  HANDLE  hThread;
  DWORD  IDThread;

  cout << "n = " << n << endl;
  // запускаем поток Add
  hThread = CreateThread(NULL, 0, Add, (void*)inc, 0, &IDThread);
  if (hThread == NULL)
    return GetLastError();

  // ждем, пока поток Add закончит работу
  WaitForSingleObject(hThread, INFINITE);
  // закрываем дескриптор потока Add
  CloseHandle(hThread);

  cout << "n = " << n << endl;

  return 0;
}
```

Отметим, что в этой программе используется функция `WaitForSingleObject`, которая ждет завершения потока `Add`. Подробно эта функция будет рассмотрена далее, в *разд. 6.2*, посвященном объектам синхронизации и функциям ожидания.

Замечание

Отметим, что перед компиляцией этой программы в консольном проекте необходимо установить режим отладки многопоточных приложений. В среде разработки Visual C++ 6.0 это делается следующим образом: в пункте меню **Project** выбирается команда **Settings**. Далее, в появившемся диалоговом окне **ProjectSettings** выбирается вкладка **C/C++**. Теперь в списке **Category** выбираем строку **Code Generation**, а в списке **Use run-time library** выбираем строку **Debug Multithreaded**, если программа будет отлаживаться, или **Multithreaded**, если программа уже готова к использованию. После этого нажимаем **OK** и программа готова к компиляции, редактированию связей и выполнению. Эти же действия необходимо выполнить перед отладкой любого многопоточного приложения.

Для создания потоков можно также использовать макрокоманду `_beginthreadex`, которая описана в заголовочном файле process.h и имеет те же параметры, что и функция `CreateThread`. Как утверждает Джеффри Рихтер в своей книге "Программирование приложений для Windows", использование этой макрокоманды более надежно, чем непосредственный вызов функции `CreateThread`. За более подробной информацией по этому вопросу нужно обратиться к первоисточнику, а именно к вышеупомянутой книге Джеффри Рихтера.

В листинге 3.2 приведен пример программы, которая использует макрокоманду `_beginthreadex` для создания потока и демонстрирует способ передачи параметров исполняемой потоком функции.

Листинг 3.2. Создание потока макрокомандой `_beginthreadex`

```
#include <windows.h>
#include <iostream.h>
#include <string.h>
#include <process.h>

UINT WINAPI thread(void *pString)
{
  int  i = 1;
  char  *pLexema;

  pLexema = strtok((char*) pString," ");
```

```
  while (pLexema != NULL)
  {
    cout << "Thread find the lexema " << i << " : "<< pLexema << endl;
    pLexema = strtok(NULL, " ");
    i++;
  }

  return 0;
}

int main()
{
  char  sentence[80];
  int  i, j, k = 0;
  HANDLE  hThread;
  UINT  IDThread;

  cout << "Input string: ";
  cin.getline(sentence, 80);
  j = strlen(sentence);

  // создаем поток для подсчета лексем
  hThread = (HANDLE)
    _beginthreadex(NULL, 0, thread, sentence, 0, &IDThread);
  if (hThread == NULL)
    return GetLastError();

    // сами подсчитываем количество букв "a" в строке
  for (i=0; i<j; i++)
    if (sentence[i] == 'a')
      k++;
  cout << "Number of symbols 'a' in the string = " << k << endl;

  // ждем окончания разбора на лексемы
  WaitForSingleObject(hThread, INFINITE);
  // закрываем дескриптор потока thread
  CloseHandle(hThread);

  return 0;
}
```

3.3. Завершение потоков

Поток завершается вызовом функции ExitThread, которая имеет следующий прототип:

```
VOID  ExitThread(
  DWORD  dwExitCode      // код завершения потока
);
```

Эта функция может вызываться как явно, так и неявно при возврате значения из функции потока. При выполнении этой функции система посылает динамическим библиотекам, которые загружены процессом, сообщение DLL_THREAD_DETACH, которое говорит о том, что поток завершает свою работу.

Если поток создается при помощи макрокоманды _beginthreadex, то для завершения потока нужно использовать макрокоманду _endthreadex, единственным параметром которой является код возврата из потока. Эта макрокоманда описана в заголовочном файле process.h. Причина использования в этом случае макрокоманды _endthreadex заключается в том, что она не только выполняет выход из потока, но и освобождает память, которая была распределена макрокомандой _beginthreadex. Если поток создан функцией _beginthreadex, то для выхода из потока функция _endthreadex может вызываться как явно, так и неявно при возврате значения из функции потока.

Один поток может завершить другой поток, вызвав функцию TerminateThread, которая имеет следующий прототип:

```
BOOL  TerminateThread(
  HANDLE  hThread,       // дескриптор потока
  DWORD  dwExitThread    // код завершения потока
);
```

В случае успешного завершения функция TerminateThread возвращает ненулевое значение, в противном случае — FALSE. Функция TerminateThread завершает поток, но не освобождает все ресурсы, принадлежащие этому потоку. Это происходит потому, что при выполнении этой функции система не посылает динамическим библиотекам, загруженным процессом, сообщение о том, что поток завершает свою работу. В результате динамическая библиотека не освобождает ресурсы, которые были захвачены для работы с этим потоком. Поэтому эта функция должна вызываться только в аварийных ситуациях при зависании потока.

В листинге 3.3 приведена программа, которая демонстрирует работу функции TerminateThread. В этой программе следует обратить внимание на квалификатор типа volatile, который указывает компилятору, что значение переменной count должно храниться в памяти, т. к. к этой переменной имеют доступ параллельные потоки. Дело в том, что сам компилятор языка

программирования C или C++ не знает, что такое поток. Для него это просто функция. А в языках программирования C и C++ любая функция вызывается только синхронно, т. е. функция, вызвавшая другую функцию, ждет завершения этой функции. Если не использовать квалификатор `volatile`, то компилятор может оптимизировать код и в одном потоке хранить значение переменной в регистре, а в другом потоке — в оперативной памяти. В результате параллельно работающие потоки будут обращаться к разным переменным.

Листинг 3.3. Завершение потока функцией `TerminateThread`

```
#include <windows.h>
#include <iostream.h>

volatile UINT count;

void thread()
{
  for (;;)
  {
    ++count;
    Sleep(100);    // немного отдохнем
  }
}

int main()
{
  HANDLE   hThread;
  DWORD  IDThread;
  char c;

  hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread, NULL,
                         0, &IDThread);
  if (hThread == NULL)
    return GetLastError();

  for (;;)
  {
    cout << "Input 'y' to display the count or any char to finish: ";
    cin >> c;
```

```
    if (c == 'y')
      cout << "count = " << count << endl;
    else
      break;
  }

  // прерываем выполнение потока thread
  TerminateThread(hThread, 0);

  // закрываем дескриптор потока
  CloseHandle(hThread);

  return 0;
}
```

3.4. Приостановка и возобновление потоков

Каждый созданный поток имеет счетчик приостановок, максимальное значение которого равно `MAXIMUM_SUSPEND_COUNT`. Счетчик приостановок показывает, сколько раз исполнение потока было приостановлено. Поток может исполняться только при условии, что значение счетчика приостановок равно нулю. В противном случае поток не исполняется или, как говорят, находится в подвешенном состоянии. Исполнение каждого потока может быть приостановлено вызовом функции `SuspendThread`, которая имеет следующий прототип:

```
DWORD  SuspendThread(
  HANDLE  hThread        // дескриптор потока
);
```

Эта функция увеличивает значение счетчика приостановок на 1 и, при успешном завершении, возвращает текущее значение этого счетчика. В случае неудачи функция `SuspendThread` возвращает значение, равное −1.

Отметим, что поток может приостановить также и сам себя. Для этого он должен передать функции `SuspendThread` свой псевдодескриптор, который можно получить при помощи функции `GetCurrentThread`. Подробнее псевдодескрипторы потоков будут рассмотрены в *разд. 3.5*.

Для возобновления исполнения потока используется функция `ResumeThread`, которая имеет следующий прототип:

```
DWORD  ResumeThread(
  HANDLE  hThread        // дескриптор потока
);
```

Функция `ResumeThread` уменьшает значение счетчика приостановок на 1 при условии, что это значение было больше нуля. Если полученное значение счетчика приостановок равно 0, то исполнение потока возобновляется, в противном случае поток остается в подвешенном состоянии. Если при вызове функции `ResumeThread` значение счетчика приостановок было равным 0, то это значит, что поток не находится в подвешенном состоянии. В этом случае функция не выполняет никаких действий. При успешном завершении функция `ResumeThread` возвращает текущее значение счетчика приостановок, в противном случае — значение −1.

Поток может задержать свое исполнение вызовом функции `Sleep`, которая имеет следующий прототип:

```
VOID  Sleep(
  DWORD  dwMilliseconds    // миллисекунды
);
```

Единственный параметр функции `Sleep` определяет количество миллисекунд, на которые поток, вызвавший эту функцию, приостанавливает свое исполнение. Если значение этого параметра равно 0, то выполнение потока просто прерывается, а затем возобновляется при условии, что нет других потоков, ждущих выделения процессорного времени. Если же значение этого параметра равно `INFINITE`, то поток приостанавливает свое исполнение навсегда, что приводит к блокированию работы приложения.

В листинге 3.4 приведена программа, которая демонстрирует работу функций `SuspendThread`, `ResumeThread` и `Sleep`.

Листинг 3.4. Пример работы функций `SuspendThread`, `ResumeThread` и `Sleep`

```
#include <windows.h>
#include <iostream.h>

volatile UINT  nCount;
volatile DWORD  dwCount;

void thread()
{
  for (;;)
  {
    nCount++;
    // приостанавливаем поток на 100 миллисекунд
    Sleep(100);
  }
```

```
}

int main()
{
  HANDLE   hThread;
  DWORD  IDThread;
  char c;

  hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread, NULL,
                         0, &IDThread);
  if (hThread == NULL)
    return GetLastError();

  for (;;)
  {
    cout << "Input :" << endl;
    cout << "\t'n' to exit" << endl;
    cout << "\t'y' to display the count" << endl;
    cout << "\t's' to suspend thread" << endl;
    cout << "\t'r' to resume thread" << endl;
    cin >> c;

    if (c == 'n')
      break;
    switch (c)
    {
    case 'y':
      cout << "count = " << nCount << endl;
      break;
    case 's':
      // приостанавливаем поток thread
      dwCount = SuspendThread(hThread);
      cout << "Thread suspend count = " << dwCount << endl;
      break;
    case 'r':
      // возобновляем поток thread
      dwCount = ResumeThread(hThread);
      cout << "Thread suspend count = " << dwCount << endl;
      break;
```

```
    }
  }

  // прерываем выполнение потока thread
  TerminateThread(hThread, 0);
  // закрываем дескриптор потока thread
  CloseHandle(hThread);

  return 0;
}
```

3.5. Псевдодескрипторы потоков

Иногда потоку требуется знать свой дескриптор, чтобы изменить какие-то свои характеристики. Например, поток может изменить свой приоритет. Для этих целей в Win32 API существует функция `GetCurrentThread`, которая имеет следующий прототип:

```
HANLDE  GetCurrentThread(VOID);
```

и возвращает псевдодескриптор текущего потока. *Псевдодескриптор текущего потока* отличается от настоящего дескриптора потока тем, что он может использоваться только самим текущим потоком и, следовательно, может наследоваться другими процессами. Псевдодескриптор потока не нужно закрывать после его использования. Из псевдодескриптора потока можно получить настоящий дескриптор потока, для этого псевдодескриптор нужно продублировать, вызвав функцию `DuplicateHandle`. Подробно дублирование дескрипторов рассматривается в *гл. 4*.

В листинге 3.5 приведен пример программы, которая вызывает функцию `GetCurrentThread`, а затем выводит на консоль полученный псевдодескриптор.

Листинг 3.5. Пример работы функции `GetCurrentThread`

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hThread;

  // получаем псевдодескриптор текущего потока
  hThread = GetCurrentThread();
```

```
  // выводим псевдодескриптор на консоль
  cout << hThread << endl;

  cin.get();

  return 0;
}
```

3.6. Обработка ошибок в Windows

Большинство функций Win32 API возвращают код, по которому можно определить, как завершилась функция: успешно или нет. Если функция завершилась неудачей, то код возврата обычно равен `FALSE`, `NULL` или −1. В этом случае функция Win32 API также устанавливает внутренний код ошибки, который называется *кодом последней ошибки* (last-error code) и поддерживается отдельно для каждого потока. Чтобы получить код последней ошибки, нужно вызвать функцию `GetLastError`, которая имеет следующий прототип:

```
DWORD  GetLastError(VOID);
```

Эта функция возвращает код последней ошибки, установленной в потоке. Установить код последней ошибки в потоке можно при помощи функции `SetLastError`, имеющей следующий прототип:

```
VOID  SetLastError(
  DWORD  dwErrCode       // код ошибки
);
```

Чтобы получить сообщение, соответствующее коду последней ошибки, необходимо использовать функцию `FormatMessage`, которая имеет следующий прототип:

```
DWORD  FormatMessage(
  DWORD    dwFlags,        // режимы форматирования
  LPCVOID  lpSource,       // источник сообщения
  DWORD    dwMessageId,    // идентификатор сообщения
  DWORD    dwLanguageId,   // идентификатор языка
  LPTSTR   lpBuffer,       // буфер для сообщения
  DWORD    nSize,          // максимальный размер буфера для сообщения
  va_list  *Arguments      // список значений для вставки в сообщение
);
```

Мы не будем подробно рассматривать эту функцию, которая предназначена для форматирования символьных сообщений, приведем только пример ее использования для вывода сообщения об ошибке (message box). Для этого

приведем сначала, в листинге 3.6, текст функции, которая выводит сообщение об ошибке, а затем, в листинге 3.7, программу, которая использует эту функцию.

Листинг 3.6. Функция для вывода сообщения об ошибке в MessageBox

```
#include <windows.h>

void ErrorMessageBox()
{
  LPVOID lpMsgBuf;

  FormatMessage(
    FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM |
    FORMAT_MESSAGE_IGNORE_INSERTS,
    NULL,
    GetLastError(),
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),  // язык по умолчанию
    (LPTSTR) &lpMsgBuf,
    0,
    NULL
  );
  // Показать ошибку в MessageBox.
  MessageBox(
    NULL,
    (LPCTSTR)lpMsgBuf,
    "Ошибка Win32 API",
    MB_OK | MB_ICONINFORMATION
  );
  // Освободить буфер.
  LocalFree(lpMsgBuf);
}
```

Листинг 3.7. Пример вывода сообщения об ошибке в MessageBox

```
#include <windows.h>

// прототип функции вывода сообщения об ошибке в MessageBox
```

```
void ErrorMessageBox();

// тест для функции вывода сообщения об ошибке на консоль
int main()
{
  HANDLE  hHandle=NULL;

  // неправильный вызов функции закрытия дескриптора
  if (!CloseHandle(hHandle))
    ErrorMessageBox();

  return 0;
}
```

Теперь, раз мы работаем с консольными приложениями, рассмотрим, как выводить сообщение об ошибке на консоль. Для этого нам нужно научиться выводить русский текст на консоль. Это можно сделать при помощи функции `CharToOem`, которая имеет следующий прототип:

```
BOOL  CharToOem(
  LPCTSTR  lpszSrc,      // строка для перекодировки
  LPSTR    lpszDst       // перекодированная строка
);
```

Эта функция перекодирует символы из кодировки Microsoft в кодировку, определенную производителем оборудования. Аббревиатура OEM расшифровывается как Original Equipment Manufacturer (настоящий производитель аппаратуры).

В листинге 3.8 приведен пример использования функции `CharToOem`.

Листинг 3.8. Пример перекодировки русских букв для вывода на консоль

```
#include <windows.h>
#include <iostream.h>

int main()
{

  char big[] = "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
  char sml[] = "абвгдеёжзийклмнопрстуфхцчшщъыьэюя";

  CharToOem(big,big);
```

```
  CharToOem(sml,sml);

  cout << big << endl;
  cout << sml << endl;

  return 0;
}
```

Теперь определим функцию, которая выводит сообщение об ошибке на консоль, на русском языке. Текст этой функции приведен в листинге 3.9.

Листинг 3.9. Функция для вывода сообщения об ошибке на консоль на русском языке

```
#include <windows.h>
#include <iostream.h>

void CoutErrorMessage()
{
  char prefix[] = "Ошибка Win32 API: ";
  LPVOID lpMsgBuf;

  CharToOem(prefix,prefix);  // перекодируем заголовок

  FormatMessage(
    FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM |
    FORMAT_MESSAGE_IGNORE_INSERTS,
    NULL,
    GetLastError(),
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // язык по умолчанию
    (LPTSTR) &lpMsgBuf,
    0,
    NULL
  );
  // перекодируем русские буквы
  CharToOem((char*)lpMsgBuf,(char*)lpMsgBuf);
  // выводим сообщение об ошибке на консоль
```

```
  cout << prefix << (char*)lpMsgBuf << endl;
  // освобождаем буфер
  LocalFree(lpMsgBuf);
}
```

В листинге 3.10 приведен пример использования функции `CoutErrorMessage` в консольном приложении.

Листинг 3.10. Пример вывода сообщения об ошибке на консоль

```
#include <windows.h>
#include <iostream.h>

// прототип функции для вывода сообщения об ошибке на консоль
void CoutErrorMessage();

// тест для функции вывода сообщения об ошибке на консоль
int main()
{
  HANDLE  hHandle=NULL;

  // неправильный вызов функции закрытия дескриптора
  if (!CloseHandle(hHandle))
    CoutErrorMessage();

  return 0;
}
```

Замечание

Если при запуске приложения используется отладчик, то текст сообщения, соответствующий коду последней ошибки, можно посмотреть в окне watch, если набрать в строке "имя переменной" следующий текст: @err,hr.

Глава 4

Процессы в Windows

4.1. Определение процесса

В Windows под *процессом* понимается объект ядра, которому принадлежат системные ресурсы, используемые исполняемым приложением. Поэтому можно сказать, что в Windows процессом является исполняемое приложение. Выполнение каждого процесса начинается с первичного потока. Во время своего исполнения процесс может создавать другие потоки. Исполнение процесса заканчивается при завершении работы всех его потоков. Каждый процесс в операционной системе Windows владеет следующими ресурсами:

- виртуальным адресным пространством;
- рабочим множеством страниц в реальной памяти;
- маркером доступа, содержащим информацию для системы безопасности;
- таблицей для хранения дескрипторов объектов ядра.

Кроме дескриптора, каждый процесс в Windows имеет свой идентификатор, который является уникальным для процессов, выполняющихся в системе. Идентификаторы процессов используются, главным образом, служебными программами, которые позволяют пользователям системы отслеживать работу процессов.

4.2. Создание процессов

Новый процесс в Windows создается вызовом функции `CreateProcess`, которая имеет следующий прототип:

```
BOOL  CreateProcess(
  LPCTSTR  lpApplicationName,      // имя исполняемого модуля
  LPTSTR   lpCommandLine,          // командная строка
```

```
    LPSECURITY_ATTRIBUTES  lpProcessAttributes,  // защита процесса
    LPSECURITY_ATTRIBUTES  lpThreadAttributes,   // защита потока
    BOOL     bInheritHandles,        // признак наследования дескриптора
    DWORD    dwCreationFlags,        // флаги создания процесса
    LPVOID   lpEnvironment,          // блок новой среды окружения
    LPCTSTR  lpCurrentDirectory,     // текущий каталог
    LPSTARTUPINFO  lpStartUpInfo,    // вид главного окна
    LPPROCESS_INFORMATION  lpProcessInformation  // информация о процессе
  );
```

Функция `CreateProcess` возвращает ненулевое значение, если процесс был создан успешно. В противном случае эта функция возвращает значение `FALSE`. Процесс, который создает новый процесс, называется *родительским процессом* (parent process) по отношению к создаваемому процессу. Новый же процесс, который создается другим процессом, называется *дочерним процессом* (child process) по отношению к процессу-родителю. Сейчас мы опишем только назначение некоторых параметров функции `CreateProcess`. Остальные параметры этой функции будут описываться по мере их использования.

Первый параметр `lpApplicationName` определяет строку с именем исполняемого файла, который имеет тип exe и будет запускаться при создании нового процесса. Эта строка должна заканчиваться нулем и содержать полный путь к исполняемому файлу.

Для примера рассмотрим программу, которая выводит на консоль свое имя и параметры. Эта программа приведена в листинге 4.1.

Листинг 4.1. Программа, которая выводит на консоль свое имя и параметры

```
#include <conio.h>

int main(int argc, char *argv[])
{
  int i;

  _cputs("I am created.");

  _cputs("\nMy name is: ");
  _cputs(argv[0]);

  for (i = 1; i < argc; ++i)
```

```
    _cprintf ("\n My %d parameter =  %s", i, argv[i]);

  _cputs("\nPress any key to finish.\n");
  _getch();

  return 0;
}
```

Скомпилируем эту программу. Полученный exe-файл сохраним на диске C: и назовем ConsoleProcess.exe. Наша задача состоит в запуске этого файла как нового процесса. Как это сделать показано в листинге 4.2, где приведена программа, запускающая созданный exe-файл как консольный процесс с новой консолью.

Листинг 4.2. Программа процесса, который создает процесс с новой консолью

```
#include <windows.h>
#include <conio.h>

int main()
{
  char lpszAppName[] = "C:\\ConsoleProcess.exe";

  STARTUPINFO si;
  PROCESS_INFORMATION piApp;

  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb = sizeof(STARTUPINFO);

  // создаем новый консольный процесс
  if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
      CREATE_NEW_CONSOLE, NULL, NULL, &si, &piApp))
  {
    _cputs("The new process is not created.\n");
    _cputs("Check a name of the process.\n");
    _cputs("Press any key to finish.\n");
    _getch();
    return 0;
  }

  _cputs("The new process is created.\n");
```

```
  // ждем завершения созданного процесса
  WaitForSingleObject(piApp.hProcess, INFINITE);
  // закрываем дескрипторы этого процесса в текущем процессе
  CloseHandle(piApp.hThread);
  CloseHandle(piApp.hProcess);

  return 0;
}
```

Отметим в этой программе два момента. Во-первых, перед запуском консольного процесса ConsoleProcess.exe все поля структуры `si` типа `STARTUPINFO` должны заполняться нулями. Это делается при помощи вызова функции `ZeroMemory`, которая предназначена для этой цели и имеет следующий прототип:

```
VOID  ZeroMemory(
  PVOID    Destination,    // адрес блока памяти
  SIZE_T   Length          // длина блока памяти
);
```

В этом случае вид главного окна запускаемого приложения определяется по умолчанию самой операционной системой Windows. Во-вторых, в параметре `dwCreationFlags` устанавливается флаг `CREATE_NEW_CONSOLE`. Это говорит системе о том, что для запускаемого процесса должна быть создана новая консоль. Если этот параметр будет равен `NULL`, то новая консоль для запускаемого процесса не создается и весь консольный вывод нового процесса будет направляться в консоль родительского процесса.

Структура `piApp` типа `PROCESS_INFORMATION` содержит идентификаторы и дескрипторы нового создаваемого процесса и его главного потока. Мы не используем эти дескрипторы в нашей программе и поэтому закрываем их. Значение `FALSE` параметра `bInheritHandle` говорит о том, что эти дескрипторы не являются наследуемыми. О наследовании дескрипторов мы поговорим подробнее в *разд. 4.4*.

Теперь запустим наш новый консольный процесс другим способом, используя второй параметр функции `CreateProcess`. Это можно сделать при помощи программы, приведенной в листинге 4.3.

Листинг 4.3. Программа процесса, который создает процесс с новой консолью

```
#include <windows.h>
#include <conio.h>

int main()
```

```
{
  char lpszCommandLine[] = "C:\\ConsoleProcess.exe p1 p2 p3";

  STARTUPINFO si;
  PROCESS_INFORMATION piCom;

  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb = sizeof(STARTUPINFO);

  // создаем новый консольный процесс
  CreateProcess(NULL, lpszCommandLine, NULL, NULL, FALSE,
      CREATE_NEW_CONSOLE, NULL, NULL, &si, &piCom);
  // закрываем дескрипторы этого процесса
  CloseHandle(piCom.hThread);
  CloseHandle(piCom.hProcess);

  _cputs("The new process is created.\n");
  _cputs("Press any key to finish.\n");
  _getch();

  return 0;
}
```

Отличие этой программы от программы, приведенной в листинге 4.2, состоит в том, что мы передаем системе имя нового процесса и его параметры через командную строку. В этом случае имя нового процесса может и не содержать полный путь к exe-файлу, а только имя самого exe-файла. При использовании параметра `lpCommandLine` система для запуска нового процесса осуществляет поиск требуемого exe-файла в следующей последовательности каталогов:

- каталог, из которого запущено приложение;
- текущий каталог родительского процесса;
- системный каталог Windows;
- каталог Windows;
- каталоги, которые перечислены в переменной PATH среды окружения.

Для иллюстрации сказанного запустим приложение Notepad.exe (Блокнот), используя командную строку. Программа, запускающая Блокнот из командной строки, приведена в листинге 4.4.

Листинг 4.4. Запуск приложения Notepad

```
#include <windows.h>
#include <iostream.h>

int main()
{
  STARTUPINFO si;
  PROCESS_INFORMATION pi;

  // заполняем значения структуры STARTUPINFO по умолчанию
  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb = sizeof(STARTUPINFO);

  // запускаем процесс Notepad
  if (!CreateProcess(
      NULL,    // имя не задаем
      "Notepad.exe",  // имя программы
      NULL,    // атрибуты защиты процесса устанавливаем по умолчанию
      NULL,    // атрибуты защиты первичного потока по умолчанию
      FALSE,   // дескрипторы текущего процесса не наследуются
      0,       // по умолчанию NORMAL_PRIORITY_CLASS
      NULL,    // используем среду окружения вызывающего процесса
      NULL,    // текущий диск и каталог, как и в вызывающем процессе
      &si,     // вид главного окна - по умолчанию
      &pi      // информация о новом процессе
)
    )
  {
    cout << "The mew process is not created." << endl
      << "Check a name of the process." << endl;
    return 0;
  }

  Sleep(1000);  // немного подождем и закончим свою работу
  // закроем дескрипторы запущенного процесса в текущем процессе
  CloseHandle(pi.hThread);
  CloseHandle(pi.hProcess);

  return 0;
}
```

4.3. Завершение процессов

Процесс может завершить свою работу вызовом функции `ExitProcess`, которая имеет следующий прототип:

```
VOID  ExitProcess(
  UINT    uExitCode    // код возврата из процесса
);
```

При вызове функции `ExitProcess` завершаются все потоки процесса с кодом возврата, который является параметром этой функции. При выполнении этой функции система посылает динамическим библиотекам, которые загружены процессом, сообщение `DLL_PROCESS_DETACH`, которое говорит о том, что динамическую библиотеку необходимо отсоединить от процесса.

В листинге 4.5 приведен пример программы, которая завершает свою работу вызовом функции `ExitProcess`.

Листинг 4.5. Завершение процесса функцией `ExitProcess`

```
#include <windows.h>
#include <iostream.h>

volatile UINT count;

void thread()
{
  for (;;)
  {
    count++;
    Sleep(100);
  }
}

int main()
{
  char c;
  HANDLE   hThread;
  DWORD  IDThread;

  hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread, NULL,
                         0, &IDThread);
```

```
  if (hThread == NULL)
    return GetLastError();

  for (;;)
  {
    cout << "Input 'y' to display the count or any char to exit: ";
    cin >> (char)c;
    if (c == 'y')
      cout << "count = " << count << endl;
    else
      ExitProcess(1);
  }
}
```

Один процесс может быть завершен другим при помощи вызова функции `TerminateProcess`, которая имеет следующий прототип:

```
BOOL  TerminateProcess(
  HANDLE  hProcess,      // дескриптор процесса
  UINT    uExitCode      // код возврата
);
```

Если функция `TerminateProcess` выполнилась успешно, то она возвращает ненулевое значение. В противном случае возвращаемое значение равно `FALSE`. Функция `TerminateProcess` завершает работу процесса, но не освобождает все ресурсы, принадлежащие этому процессу. Это происходит потому, что при выполнении этой функции система не посылает динамическим библиотекам, загруженным процессом, сообщение о том, что библиотеку необходимо отсоединить от процесса. Поэтому эта функция должна вызываться только в аварийных ситуациях при зависании процесса.

Приведем программу, которая демонстрирует работу функции `TerminateProcess`. Для этого сначала создадим бесконечный процесс-счетчик, который назовем ConsoleProcess.exe, и расположим на диске C: (листинг 4.6).

Листинг 4.6. Программа бесконечного процесса

```
#include <windows.h>
#include <iostream.h>

int count;

void main()
```

```
{
  for ( ; ; )
  {
    count++;
    Sleep(1000);
    cout << "count = " << count << endl;
  }
}
```

Теперь рассмотрим программу, которая создает этот бесконечный процесс-счетчик, а потом завершает его по требованию пользователя, используя для этого функцию `TerminateProcess`. Эта программа приведена в листинге 4.7.

Листинг 4.7. Завершение процесса функцией `TerminateProcess`

```
#include <windows.h>
#include <conio.h>

int main()
{
  char lpszAppName[] = "C:\\ConsoleProcess.exe";

  STARTUPINFO si;
  PROCESS_INFORMATION pi;

  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb=sizeof(STARTUPINFO);

  // создаем новый консольный процесс
  if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
      CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi))
  {
    _cputs("The new process is not created.\n");
    _cputs("Check a name of the process.\n");
    _cputs("Press any key to finish.\n");
    _getch();
    return 0;
  }

  _cputs("The new process is created.\n");
```

```
  while (true)
  {
    char c;

    _cputs("Input 't' to terminate the new console process: ");
    c = _getch();
    if (c == 't')
    {
      _cputs("t\n");
      // завершаем новый процесс
      TerminateProcess(pi.hProcess,1);
      break;
    }
  }

  // закрываем дескрипторы нового процесса в текущем процессе
  CloseHandle(pi.hThread);
  CloseHandle(pi.hProcess);

  return 0;
}
```

4.4. Наследование дескрипторов

Большинство объектов категории Kernel могут быть *наследуемыми* или *ненаследуемыми*. Свойство наследования объекта означает, что если наследуемый объект создан или открыт в некотором процессе, то к этому объекту будут также иметь доступ все процессы, которые создаются этим процессом, т. е. являются его потомками. Свойство наследования объекта определяется его дескриптором, который также может быть *наследуемым* или *ненаследуемым*. Для того чтобы объект стал наследуемым, необходимо сделать наследуемым его дескриптор и наоборот.

Свойство наследования не поддерживается для объектов, использование которых несколькими процессами нарушило бы изолированность памяти процесса от других процессов. Поэтому не могут наследоваться следующие дескрипторы:

- дескриптор виртуальной памяти, который возвращает любая из функций `LocalAlloc`, `GlobalAlloc`, `HeapCreate` или `HeapAlloc`;
- дескриптор динамической библиотеки, который возвращает функция `LoadLibrary`.

Однако для того чтобы дочерний процесс имел доступ к наследуемому объекту в родительском процессе, недостаточно просто сделать дескриптор этого объекта наследуемым. Кроме этого, нужно, во-первых, установить значение параметра `bInheritHandles` функции `CreateProcess` в `TRUE` и, во-вторых, передать сам дескриптор дочернему процессу, который создается функцией `CreateProcess`. Наследуемый дескриптор передается системой дочернему процессу неявно и поэтому он скрыт от программ, которые выполняются в дочернем процессе. То есть программа дочернего процесса должна явно знать этот дескриптор и передать его этой программе должна программа родительского процесса. Одним из способов передачи дескрипторов дочернему процессу является использование командной строки, которая позволяет передавать дескрипторы как параметры.

Для пояснения сказанного приведем пример двух процессов, в которых используются наследуемые дескрипторы. В листинге 4.8 приведена программа дочернего процесса. Этот процесс получает дескриптор потока от родительского процесса и, по требованию пользователя, прекращает выполнение этого потока в родительском процессе.

Листинг 4.8. Завершение потока в родительском процессе, используя дескриптор потока, который передается через командную строку

```
#include <windows.h>
#include <conio.h>

int main(int argc, char *argv[])
{
  HANDLE  hThread;
  char  c;

  // преобразуем символьное представление дескриптора в число
  hThread = (HANDLE)atoi(argv[1]);
  // ждем команды о завершении потока
  while (true)
  {
    _cputs("Input 't' to terminate the thread: ");
    c = _getch();
    if (c == 't')
    {
      _cputs("t\n");
      break;
    }
```

```
  }
  // завершаем поток
  TerminateThread(hThread, 0);
  // закрываем дескриптор потока
  CloseHandle(hThread);

  _cputs("Press any key to exit.\n");
  _getch();

  return 0;
}
```

В листинге 4.9 приведена программа родительского процесса, который создает дочерний процесс и передает ему через командную строку наследуемый дескриптор потока. Этот поток и должен быть завершен дочерним процессом, программа которого приведена в листинге 4.8.

Листинг 4.9. Процесс, который передает наследуемый дескриптор потока дочернему процессу через командную строку

```
#include <windows.h>
#include <conio.h>

volatile int count;

void thread()
{
  for (;;)
  {
    count++;
    Sleep(500);
    _cprintf ("count = %d\n", count);
  }
}

int main()
{
  char lpszComLine[80];  // для командной строки

  STARTUPINFO si;
```

```
PROCESS_INFORMATION pi;
SECURITY_ATTRIBUTES sa;

HANDLE  hThread;
DWORD  IDThread;

_cputs("Press any key to start the count-thread.\n");
_getch();

// устанавливает атрибуты защиты потока
sa.nLength = sizeof(SECURITY_ATTRIBUTES);
sa.lpSecurityDescriptor = NULL;     // защита по умолчанию
sa.bInheritHandle = TRUE;           // дескриптор потока наследуемый

// запускаем поток-счетчик
hThread = CreateThread(&sa, 0, (LPTHREAD_START_ROUTINE)thread, NULL, 0,
                       &IDThread);
if (hThread == NULL)
  return GetLastError();

// устанавливаем атрибуты нового процесса
ZeroMemory(&si, sizeof(STARTUPINFO));
si.cb=sizeof(STARTUPINFO);
// формируем командную строку
wsprintf(lpszComLine, "C:\\ConsoleProcess.exe %d", (int)hThread);
// запускаем новый консольный процесс
if (!CreateProcess(
    NULL,    // имя процесса
    lpszComLine,  // адрес командной строки
    NULL,    // атрибуты защиты процесса по умолчанию
    NULL,    // атрибуты защиты первичного потока по умолчанию
    TRUE,    // наследуемые дескрипторы текущего процесса
             // наследуются новым процессом
    CREATE_NEW_CONSOLE,  // новая консоль
    NULL,    // используем среду окружения процесса предка
    NULL,    // текущий диск и каталог, как и в процессе-предке
    &si,     // вид главного окна - по умолчанию
    &pi      // здесь будут дескрипторы и идентификаторы
             // нового процесса и его первичного потока
```

```
      )
    )
  {
    _cputs("The new process is not created.\n");
    _cputs("Press any key to finish.\n");
    _getch();
    return GetLastError();
  }
  // закрываем дескрипторы нового процесса
  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);

  // ждем закрытия потока-счетчика
  WaitForSingleObject(hThread, INFINITE);
  _cputs("Press any key to exit.\n");
  _getch();
  // закрываем дескриптор потока
  CloseHandle(hThread);

  return 0;
}
```

В программе из листинга 4.9 особенно нужно обратить внимание на два момента: значение параметра `bInheritHandle` функции `CreateProcess` и использование структуры `sa` типа `SECURITY_ATTRIBUTES`, адрес которой является первым параметром функции `CreateThread`. Если значение параметра `bInheritHandle` равно `TRUE`, то наследуемые дескрипторы родительского процесса передаются дочернему процессу. Поле `bInheritHandle` структуры `sa` имеет тип `BOOL`. Если значение этого поля установлено в `TRUE`, то дескриптор создаваемого потока является наследуемым, в противном случае — ненаследуемым.

Теперь рассмотрим следующую ситуацию. Предположим, что дескриптор созданного объекта является ненаследуемым, а нам необходимо сделать его наследуемым. Для решения этой проблемы в операционной системе Windows 2000 можно использовать функцию `SetHandleInformation`, которая используется для изменения свойств дескрипторов и имеет следующий прототип:

```
BOOL  SetHandleInformation(
  HANDLE  hObject,      // дескриптор объекта
  DWORD   dwMask,       // флаги, которые изменяем
```

```
    DWORD   dwFlags      // новые значения флагов
  );
```

В случае успешного завершения эта функция возвращает ненулевое значение, в противном случае — FALSE.

Для иллюстрации работы функции SetHandleInformation изменим программу, приведенную в листинге 4.9. Модифицированная программа приведена в листинге 4.10.

Листинг. 4.10. Изменение свойства наследования дескриптора

```
#include <windows.h>
#include <conio.h>

volatile int count;

void thread()
{
  for (;;)
  {
    count++;
    Sleep(500);
    _cprintf ("count = %d\n", count);
  }
}

int main()
{
  // имя нового процесса с пробелом
  char lpszComLine[80]="C:\\ConsoleProcess.exe ";
  // для символьного представления дескриптора
  char lpszHandle[20];

  STARTUPINFO si;
  PROCESS_INFORMATION pi;

  HANDLE  hThread;
  DWORD  IDThread;

  _cputs("Press any key to start the count-thread.\n");
```

```
_cputs("After terminating the thread press any key to exit.\n");
_getch();

// запускаем поток-счетчик
hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread, NULL,
                       0, &IDThread);
if (hThread == NULL)
  return GetLastError();

// делаем дескриптор потока наследуемым
if(!SetHandleInformation(
    hThread,               // дескриптор потока
    HANDLE_FLAG_INHERIT,   // изменяем наследование дескриптора
    HANDLE_FLAG_INHERIT))  // делаем дескриптор наследуемым
{
  _cputs("The inheritance is not changed.\n");
  _cputs("Press any char to finish.\n");
  _getch();
  return GetLastError();
}

// устанавливаем атрибуты нового процесса
ZeroMemory(&si, sizeof(STARTUPINFO));
si.cb=sizeof(STARTUPINFO);
// преобразуем дескриптор в символьную строку
_itoa((int)hThread,lpszHandle,10);
// создаем командную строку
strcat(lpszComLine,lpszHandle);
// запускаем новый консольный процесс
if (!CreateProcess(
    NULL,    // имя процесса
    lpszComLine,  // адрес командной строки
    NULL,    // атрибуты защиты процесса по умолчанию
    NULL,    // атрибуты защиты первичного потока по умолчанию
    TRUE,    // наследуемые дескрипторы текущего процесса
             // наследуются новым процессом
    CREATE_NEW_CONSOLE,  // новая консоль
    NULL,    // используем среду окружения процесса-предка
    NULL,    // текущий диск и каталог, как и в процессе-предке
```

```
      &si,      // вид главного окна - по умолчанию
      &pi       // здесь будут дескрипторы и идентификаторы
                // нового процесса и его первичного потока
      )
    )
  {
    _cputs("The new process is not created.\n");
    _cputs("Press any key to finish.\n");
    _getch();
    return GetLastError();
  }
  // закрываем дескрипторы нового процесса
  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);

  _getch();

  // закрываем дескриптор потока
  CloseHandle(hThread);

  return 0;
}
```

Для определения свойств дескриптора используется функция `GetHandleInformation`, которая имеет следующий прототип:

```
BOOL  GetHandleInformation(
  HANDLE   hObject,      // дескриптор объекта
  LPDWORD  lpdwFlags     // свойства дескриптора
);
```

Эта функция также в случае успешного завершения возвращает ненулевое значение. В противном случае возвращаемое значение равно `FALSE`.

Замечание

Отметим важную деталь, функции `SetHandleInformation` и `GetHandleInformation` правильно работают только в операционной системе Windows 2000. В операционной системе Windows 98 эти функции всегда возвращают значение `FALSE`. Для решения рассматриваемой проблемы в операционной системе Windows 98 нужно использовать дублирование дескрипторов, которое мы рассмотрим в *разд. 4.5*.

В завершение этого раздела разберем несколько подробнее, зачем нужно наследование объектов, тогда как любые процессы, включая дочерние, могут получить доступ к объекту по его имени. Проблема как раз и состоит в именовании объектов. Во-первых, затрачивается время на поиск имени созданного объекта. Но это не столь важно, важнее то, что, во-вторых, объекты должны иметь уникальные имена. Это требуется для того, чтобы не допустить ошибки при создании объекта по причине присутствия другого объекта с таким же именем. То есть нельзя допустить, чтобы совершенно разные приложения непреднамеренно создавали никак не связанные между собой объекты с одинаковыми именами. Кроме того, при неуникальном именовании объектов также возможны проблемы при одновременной работе двух экземпляров одного приложения. Эта проблема уже гораздо сложнее, и для ее решения используются специальные программы, которые могут генерировать уникальные имена. Такие имена обычно называются `GUID` — глобальными универсальными идентификаторами. Поэтому, как видим, проще и быстрее создавать анонимные наследуемые объекты и передавать их дескрипторы дочерним процессам, чем заниматься уникальным именованием объектов.

4.5. Дублирование дескрипторов

Дублирование дескрипторов необходимо для решения следующей задачи. Иногда при передаче дескриптора из одного процесса в другой необходимо изменить не только свойство наследования дескриптора, но и другие свойства этого дескриптора, которые управляют доступом к объекту. Для решения этой проблемы предназначена функция `DuplicateHandle`, которая имеет следующий прототип:

```
BOOL  DuplicateHandle(
  HANDLE  hSourceProcessHandle,    // дескриптор процесса источника
  HANDLE  hSourceHandle,           // исходный дескриптор
  HANDLE  hTargetProcessHandle,    // дескриптор процесса приемника
  LPHANDLE  lpTargetHandle,        // дубликат исходного дескриптора
  DWORD   dwDesiredAccess,         // флаги доступа к объекту
  BOOL    bInheritHandle,          // наследование дескриптора
  DWORD   dwOptions            // дополнительные необязательные флаги
);
```

Если функция `DuplicateHandle` завершается успешно, то она возвращает ненулевое значение. В противном случае эта функция возвращает значение `FALSE`.

Отметим назначение трех последних параметров в функции `DuplicateHandle`.

Начнем с последнего параметра — dwOptions, в котором может быть установлена комбинация флагов DUPLICATE_CLOSE_SOURCE и DUPLICATE_SAME_ACCESS. Если установлен флаг DUPLICATE_CLOSE_SOURCE, то при любом своем завершении функция DuplicateHandle закрывает исходный дескриптор. Если установлен флаг DUPLICATE_SAME_ACCESS, то режимы доступа к объекту через дублированный дескриптор совпадают с режимами доступа к объекту через исходный дескриптор. Совместное использование этих флагов обеспечивает выполнение двух указанных действий.

Теперь перейдем к параметру dwDesiredAccess, который определяет возможные режимы доступа к объекту через дубликат исходного дескриптора, используя определенную комбинацию флагов. Значения этих флагов отличаются для объектов разных типов и будут описаны далее, в процессе работы с объектами. Если доступ к объекту не изменяется, что определяется значением последнего параметра dwOptions, то система игнорирует значение параметра dwDesiredAccess.

Параметр bInheritHandle функции DuplicateHandle устанавливает свойство наследования нового дескриптора. Если значение этого параметра равно TRUE, то создаваемый дубликат исходного дескриптора является наследуемым, в случае FALSE — ненаследуемым.

В листинге 4.11 приведен пример программы, которая использует функцию DuplicateHandle для разрешения дочернему процессу завершить поток в родительском процессе. Эта программа является другим решением задачи, решаемой программой, приведенной в листинге 4.10.

Листинг 4.11. Создание наследуемого дескриптора функцией DuplicateHandle

```
#include <windows.h>
#include <conio.h>

volatile int count;

void thread()
{
  for (;;)
  {
    count++;
    Sleep(500);
    _cprintf ("count = %d\n", count);
  }
}
```

```
int main()
{
  // имя нового процесса с пробелом
  char lpszComLine[80]="C:\\ConsoleProcess.exe ";
  // для символьного представления дескриптора
  char lpszHandle[20];

  STARTUPINFO si;
  PROCESS_INFORMATION pi;

  HANDLE  hThread, hInheritThread;
  DWORD  IDThread;

  _cputs("Press any key to start the count-thread.\n");
  _cputs("After terminating the thread press any key to exit.\n");
  _getch();

  // запускаем поток-счетчик
  hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread, NULL,
                         0, &IDThread);
  if (hThread == NULL)
    return GetLastError();

  // создаем наследуемый дубликат дескриптора потока
  if(!DuplicateHandle(
      GetCurrentProcess(),  // дескриптор текущего процесса
      hThread,      // исходный дескриптор потока
      GetCurrentProcess(),  // дескриптор текущего процесса
      &hInheritThread,    // новый дескриптор потока
      0,      // этот параметр игнорируется
      TRUE,      // новый дескриптор наследуемый
      DUPLICATE_SAME_ACCESS))  // доступ не изменяем
  {
    _cputs("The handle is not duplicated.\n");
    _cputs("Press any key to finish.\n");
    _getch();
    return GetLastError();
  }
```

```
  // устанавливаем атрибуты нового процесса
  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb=sizeof(STARTUPINFO);
  // преобразуем наследуемый дескриптор в символьную строку
  _itoa((int)hInheritThread,lpszHandle,10);
  // создаем командную строку
  strcat(lpszComLine,lpszHandle);
  // запускаем новый консольный процесс
  if (!CreateProcess(
      NULL,    // имя процесса
      lpszComLine,  // адрес командной строки
      NULL,    // атрибуты защиты процесса по умолчанию
      NULL,    // атрибуты защиты первичного потока по умолчанию
      TRUE,    // наследуемые дескрипторы текущего процесса
               // наследуются новым процессом
      CREATE_NEW_CONSOLE,  // новая консоль
      NULL,    // используем среду окружения процесса-предка
      NULL,    // текущий диск и каталог, как и в процессе-предке
      &si,     // вид главного окна - по умолчанию
      &pi      // здесь будут дескрипторы и идентификаторы
               // нового процесса и его первичного потока
      )
    )
  {
    _cputs("The new process is not created.\n");
    _cputs("Press any key to finish.\n");
    _getch();
    return GetLastError();
  }
  // закрываем дескрипторы нового процесса
  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);

  _getch();

  // закрываем дескриптор потока
  CloseHandle(hThread);

  return 0;
}
```

Теперь приведем, в листинге 4.12, пример программы, которая дублирует дескриптор, но при этом изменяет доступ к нему. Разрешим дочернему процессу прекращать выполнение потока в родительском процессе. В этом случае остальные функции над потоками, такие как, например, `SuspendThread` и `ResumeThread`, будут недоступны для выполнения в дочернем процессе. Обратите внимание на изменение значений параметров функции `DuplicateHandle`.

Листинг 4.12. Создание наследуемого дескриптора функцией `DuplicateHandle` с изменением доступа к объекту

```
#include <windows.h>
#include <conio.h>

volatile int count;

void thread()
{
  for (;;)
  {
    count++;
    Sleep(500);
    _cprintf ("count = %d\n", count);
  }
}

int main()
{
  // имя нового процесса с пробелом
  char lpszComLine[80]="C:\\ConsoleProcess.exe ";
  // для символьного представления дескриптора
  char lpszHandle[20];

  STARTUPINFO si;
  PROCESS_INFORMATION pi;

  HANDLE  hThread, hInheritThread;
  DWORD  IDThread;

  _cputs("Press any key to start the count-thread.\n");
```

```
_cputs("After terminating the thread press any key to exit.\n");
_getch();

// запускаем поток-счетчик
hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread, NULL,
                       0, &IDThread);
if (hThread == NULL)
  return GetLastError();

// создаем наследуемый дубликат дескриптора потока
if(!DuplicateHandle(
    GetCurrentProcess(),   // дескриптор текущего процесса
    hThread,               // исходный дескриптор потока
    GetCurrentProcess(),   // дескриптор текущего процесса
    &hInheritThread,       // новый дескриптор потока
    THREAD_TERMINATE,      // только завершение потока
    TRUE,         // новый дескриптор наследуемый
    0))           // не используем
{
  _cputs("The handle is not duplicated.\n");
  _cputs("Press any key to finish.\n");
  _getch();
  return GetLastError();
}

// устанавливаем атрибуты нового процесса
ZeroMemory(&si, sizeof(STARTUPINFO));
si.cb=sizeof(STARTUPINFO);
// преобразуем наследуемый дескриптор в символьную строку
_itoa((int)hInheritThread,lpszHandle,10);
// создаем командную строку
strcat(lpszComLine,lpszHandle);
// запускаем новый консольный процесс
if (!CreateProcess(
    NULL,    // имя процесса
    lpszComLine,  // адрес командной строки
    NULL,    // атрибуты защиты процесса по умолчанию
    NULL,    // атрибуты защиты первичного потока по умолчанию
    TRUE,    // наследуемые дескрипторы текущего процесса
```

```
                // наследуются новым процессом
      CREATE_NEW_CONSOLE,  // новая консоль
      NULL,    // используем среду окружения процесса-предка
      NULL,    // текущий диск и каталог, как и в процессе-предке
      &si,     // вид главного окна - по умолчанию
      &pi      // здесь будут дескрипторы и идентификаторы
                // нового процесса и его первичного потока
      )
    )
  {
    _cputs("The new process is not created.\n");
    _cputs("Press any key to finish.\n");
    _getch();
    return GetLastError();
  }
  // закрываем дескрипторы нового процесса
  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);

  _getch();

  // закрываем дескриптор потока
  CloseHandle(hThread);

  return 0;
}
```

4.6. Псевдодескрипторы процессов

Иногда процессу требуется знать свой дескриптор, чтобы изменить какие-то свои характеристики. Например, процесс может изменить свой приоритет. Для этих целей в Win32 API существует функция `GetCurrentProcess`, которая имеет следующий прототип:

```
HANLDE  GetCurrentProcess(VOID);
```

и возвращает псевдодескриптор текущего процесса. *Псевдодескриптор текущего процесса* отличается от настоящего дескриптора процесса тем, что он может использоваться только текущим процессом и не может наследоваться другими процессами. Псевдодескриптор процесса не нужно закрывать после его использования. Из псевдодескриптора процесса можно получить на-

стоящий дескриптор процесса: для этого псевдодекскриптор нужно продублировать, вызвав функцию `DuplicateHandle`.

В листинге 4.13 приведен пример программы, которая поучает псевдодескриптор процесса посредством вызова функции `GetCurrentProcess`, а затем выводит полученный псевдодескриптор на консоль.

Листинг 4.13. Получение псевдодескриптора процесса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hProcess;

  // получаем псевдодескриптор текущего процесса
  hProcess = GetCurrentProcess();
  // выводим псевдодескриптор на консоль
  cout << hProcess << endl;

  cin.get();

  return 0;
}
```

4.7. Обслуживание потоков

Операционные системы Windows распределяют процессорное время между потоками в соответствии с их приоритетами. По истечении кванта времени исполнение текущего потока прерывается, его контекст запоминается и процессорное время передается потоку с высшим приоритетом. Часто говорят, что поток с высшим приоритетом вытесняет поток с низшим приоритетом. Такое обслуживание потоков в Windows называется *вытесняющая многозадачность* (preempting multitasking). Величина кванта времени, выделяемого потоку, зависит от типа операционной системы Windows, типа процессора и приблизительно равна 20 мс.

Приоритеты потоков в Windows определяются относительно приоритета процесса, в контексте которого они исполняются, и изменяются от 0 (низший приоритет) до 31 (высший приоритет). Приоритет процессов устанавливается при их создании функцией `CreateProcess`, используя параметр

dwCreationFlags. Для установки приоритета процесса в этом параметре нужно установить один из следующих флагов.

- IDLE_PRIORITY_CLASS — класс фоновых процессов;
- BELOW_NORMAL_PRIORITY_CLASS — класс процессов ниже нормальных;
- NORMAL_PRIORITY_CLASS — класс нормальных процессов;
- ABOVE_NORMAL_PRIORITY_CLASS — класс процессов выше нормальных;
- HIGH_PRIORITY_CLASS — класс высокоприоритетных процессов;
- REAL_TIME_PRIORITY_CLASS — класс процессов реального времени.

Отметим, что флаги BELOW_NORMAL_PRIORITY_CLASS и ABOVE_NORMAL_PRIORITY_CLASS используются только в операционных системах, начиная с версии Windows 2000.

Рассмотрим правила, используемые для назначения приоритетов процессам в Windows. Предполагается, что операционная система Windows различает четыре типа процессов в соответствии с их приоритетами: фоновые процессы, процессы с нормальным приоритетом, процессы с высоким приоритетом и процессы реального времени. Рассмотрим подробнее каждый из этих типов процессов.

- *Фоновые процессы* выполняют свою работу, когда нет активных пользовательских процессов. Обычно эти процессы следят за состоянием системы. Приоритет таких процессов устанавливается флагом IDLE_PRIORITY_CLASS.
- *Процессы с нормальным приоритетом* — это обычные пользовательские процессы. Приоритет таких процессов устанавливается флагом NORMAL_PRIORITY_CLASS. Этот приоритет также назначается пользовательским процессам по умолчанию. В Windows 2000 приоритет обычных пользовательских процессов может также устанавливаться флагами BELOW_NORMAL_PRIORITY_CLASS или ABOVE_NORMAL_PRIORITY_CLASS, которые соответственно немного повышают или понижают приоритет пользовательского процесса.
- *Процессы с высоким приоритетом* это такие пользовательские процессы, от которых требуется более быстрая реакция на некоторые события, чем от обычных пользовательских процессов. Приоритет таких процессов устанавливается флагом HIGH_PRIORITY_CLASS. Эти процессы должны содержать небольшой программный код и выполняться очень быстро, чтобы не замедлять работу системы. Обычно такие приоритеты имеют другие системы, работающие на платформе операционных систем Windows.
- К последнему типу процессов относятся *процессы реального времени*. Приоритет таких процессов устанавливается флагом REAL_TIME_PRIORITY_CLASS. Работа таких процессов обычно происходит в масштабе реального времени и связана с реакцией на внешние события. Эти процессы должны работать непосредственно с аппаратурой компьютера.

Приоритет процесса можно изменить при помощи функции SetPriorityClass, которая имеет следующий прототип:

```
BOOL  SetPriorityClass(
  HANDLE  hProcess,          // дескриптор процесса
  DWORD   dwPriorityClass    // приоритет
);
```

При успешном завершении функция SetPriorityClass возвращает ненулевое значение, в противном случае значение — FALSE. Параметр dwPriorityClass этой функции должен быть равен одному из флагов, которые приведены выше.

Узнать приоритет процесса можно посредством вызова функции GetPriorityClass, которая имеет следующий прототип:

```
DWORD  GetPriorityClass(
  HANDLE  hProcess         // дескриптор процесса
);
```

При успешном завершении эта функция возвращает флаг установленного приоритета процесса, в противном случае возвращаемое значение равно нулю.

В листинге 4.14 приведена программа, которая демонстрирует работу функций SetPriorityClass и GetPriorityClass.

Листинг 4.14. Пример работы функций SetPriorityClass и GetPriorityClass

```
#include <windows.h>
#include <conio.h>

int main()
{
  HANDLE  hProcess;
  DWORD  dwPriority;

  // получаем псевдодескриптор текущего потока
  hProcess = GetCurrentProcess();

  // узнаем приоритет текущего процесса
  dwPriority = GetPriorityClass(hProcess);
  _cprintf("The priority of the process = %d.\n", dwPriority);

  // устанавливаем фоновый приоритет текущего процесса
```

```
  if (!SetPriorityClass(hProcess, IDLE_PRIORITY_CLASS))
  {
    _cputs("Set priority class failed.\n");
    _cputs("Press any key to exit.\n");
    _getch();
    return GetLastError();
  }
  dwPriority = GetPriorityClass(hProcess);
  _cprintf("The priority of the process = %d.\n", dwPriority);

  // устанавливаем высокий приоритет текущего процесса
  if (!SetPriorityClass(hProcess, HIGH_PRIORITY_CLASS))
  {
    _cputs("Set priority class failed.\n");
    _cputs("Press any key to exit.\n");
    _getch();
    return GetLastError();
  }
  dwPriority = GetPriorityClass(hProcess);
  _cprintf("The priority of the process = %d.\n", dwPriority);

  _cputs("Press any key to exit.\n");
  _getch();

  return 0;
}
```

Отметим в связи с этой программой, что числовые значения флагов не соответствуют числовым значениям приоритетов процессов. Так, например, числовое значение флага `IDLE_PRIORITY_CLASS` больше чем числовое значение флага `NORMAL_PRIORITY_CLASS`. Но система считает, что приоритет нормального процесса выше, чем приоритет фонового процесса.

Теперь перейдем к приоритетам потоков, задание которых в Windows довольно запутанное. Приоритет потока, который учитывается системой при выделении потокам процессорного времени, называется *базовым* (base) или *основным приоритетом потока*. Всего существует 32 базовых приоритета — от 0 до 31. Для каждого базового приоритета существует очередь потоков. При диспетчеризации потоков квант процессорного времени выделяется потоку, который стоит первым в очереди с наивысшим базовым приоритетом. Базовый приоритет потока определяется как сумма приоритета процесса

и *уровня приоритета потока*, который может принимать одно из следующих значений, которые разобьем на две группы. Первая состоит из:

- `THREAD_PRIORITY_LOWEST` — низший приоритет;
- `THREAD_PRIORITY_BELOW_NORMAL` — приоритет ниже нормального;
- `THREAD_PRIORITY_NORMAL` — нормальный приоритет;
- `THREAD_PRIORITY_ABOVE_NORMAL` — приоритет выше нормального;
- `THREAD_PRIORITY_HIGHEST` — высший приоритет.

Вторая:

- `THREAD_PRIORITY_IDLE` — приоритет фонового потока;
- `THREAD_PRIORITY_TIME_CRITICAL` — приоритет потока реального времени.

Значения уровня приоритета потока из первой группы в сумме с приоритетом процесса, в контексте которого этот поток выполняется, уменьшают, оставляют неизменным или увеличивают значение базового приоритета потока соответственно на величину −2, −1, 0, 1, 2. Уровень приоритета потока `THREAD_PRIORITY_IDLE` устанавливает базовый приоритет потока равным 16, если приоритет процесса, в контексте которого выполняется поток, равен `REAL_TIME_PRIORITY_CLASS`, и 1 — в остальных случаях. Уровень приоритета потока `THREAD_PRIORITY_TIME_CRITICAL` устанавливает базовый приоритет потока равным 31, если приоритет процесса, в контексте которого выполняется поток, равен `REAL_TIME_PRIORITY_CLASS`, и 15 — в остальных случаях.

В табл. 4.1 приведены базовые приоритеты потоков в зависимости от приоритета процесса и уровня приоритета потока.

Таблица 4.1. *Базовые приоритеты потоков*

	Real time	High	Above normal	Normal	Below normal	Idle
Time critical	31	15	15	15	15	15
Highest	26	15	12	10	8	6
Above normal	25	14	11	9	7	5
Normal	24	13	10	8	6	4
Below normal	23	12	9	7	5	3
Lowest	22	11	8	6	4	2
Idle	16	1	1	1	1	1

В этой таблице по столбцам указаны приоритеты классов процессов, а по строкам — уровни приоритетов потоков.

При создании потока его базовый приоритет устанавливается как сумма приоритета процесса, в контексте которого этот поток выполняется, и уровня приоритета потока `THREAD_PRIORITY_NORMAL`. Для изменения приоритета потока используется функция `SetThreadPriority`, которая имеет следующий прототип:

```
BOOL  SetThreadPriority(
  HANDLE  hThread,        // дескриптор потока
  Int     nPriority       // уровень приоритета потока
);
```

При удачном завершении функция `SetThreadPriority` возвращает ненулевое значение, в противном случае — `FALSE`. Параметр `nPriority` этой функции должен быть равен одному из перечисленных уровней приоритетов.

Узнать уровень приоритета потока можно посредством вызова функции `GetThreadPriority`, которая имеет следующий прототип:

```
DWORD  GetThreadPriority(
  HANDLE  hThread         // дескриптор потока
);
```

При успешном завершении эта функция возвращает одно из значений уровня приоритета, в противном случае функция `GetThreadPriority` возвращает значение `THREAD_PRIORITY_ERROR_RETURN`.

В листинге 4.15 приведена программа, которая демонстрирует работу функций `SetThreadPriority` и `GetThreadPriority`.

Листинг 4.15. Получение и изменение приоритетов потоков

```
#include <windows.h>
#include <conio.h>

int main()
{
  HANDLE  hThread;
  DWORD  dwPriority;

  // получаем псевдодескриптор текущего потока
  hThread = GetCurrentThread();

  // узнаем уровень приоритета текущего процесса
  dwPriority = GetThreadPriority(hThread);
  _cprintf("The priority level of the thread = %d.\n", dwPriority);
```

```
  // понижаем приоритет текущего потока
  if (!SetThreadPriority(hThread, THREAD_PRIORITY_LOWEST))
  {
    _cputs("Set thread priority failed.\n");
    _cputs("Press any key to exit.\n");
    _getch();
    return GetLastError();
  }
  // узнаем уровень приоритет текущего потока
  dwPriority = GetThreadPriority(hThread);
  _cprintf("The priority level of the thread = %d.\n", dwPriority);

  // повышаем приоритет текущего потока
  if (!SetThreadPriority(hThread, THREAD_PRIORITY_HIGHEST))
  {
    _cputs("Set thread priority failed.\n");
    _cputs("Press any key to exit.\n");
    _getch();
    return GetLastError();
  }
  // узнаем уровень приоритета текущего потока
  dwPriority = GetThreadPriority(hThread);
  _cprintf("The priority level of the thread = %d.\n", dwPriority);

  _cputs("Press any key to exit.\n");
  _getch();

  return 0;
}
```

4.8. Динамическое изменение приоритетов потоков

Базовый приоритет потока может динамически изменяться системой, если этот приоритет находится в пределах между уровнями от 0 до 15. При получении потоком сообщения или при его переходе в состояние готовности система повышает базовый приоритет этого потока на 2. В процессе выполнения базовый приоритет такого потока понижается на 1, с каждым отрабо-

танным квантом времени, но никогда не опускается ниже исходного базового приоритета.

В операционной системе Windows 2000 возможно программное управление режимом динамического изменения базовых приоритетов потоков. Отмена или возобновление режима динамического изменения базового приоритета всех потоков, исполняемых в контексте процесса, выполняется при помощи функции SetProcessPriorityBoost, которая имеет следующий прототип:

```
BOOL  SetProcessPriorityBoost(
  HANDLE  hProcess,              // дескриптор процесса
  BOOL    DisablePriorityBoost   // состояние повышения приоритета
);
```

Если функция SetProcessPriorityBoost завершается успешно, то она возвращает ненулевое значение, в противном случае возвращаемое значение равно FALSE.

Значение параметра DisablePriorityBoost устанавливает состояние режима динамического повышения базовых приоритетов потоков. Если это значение равно TRUE, то режим динамического повышения базовых приоритетов потоков, выполняемых в контексте процесса с дескриптором hProcess, запрещается. Если же значение этого параметра равно FALSE, то, наоборот, режим динамического повышения базовых приоритетов этих потоков разрешается.

Узнать, разрешен ли режим динамического повышения базовых приоритетов потоков, можно посредством вызова функции GetProcessPriorityBoost, которая имеет следующий прототип:

```
BOOL  GetProcessPriorityBoost(
  HANDLE  hProcess,              // дескриптор процесса
  PBOOL   pDisablePriorityBoost  // состояние повышения приоритета
);
```

Если функция GetProcessPriorityBoost завершается успешно, то она возвращает ненулевое значение, в противном случае возвращаемое значение равно FALSE.

Значение булевой переменной, на которую указывает параметр pDisablePriorityBoost, определяет состояние режима динамического повышения базовых приоритетов потоков. Если это значение равно TRUE, то режим динамического повышения базовых приоритетов потоков, выполняемых в контексте процесса с дескриптором hProcess, запрещен. Если же значение этого параметра равно FALSE, то, наоборот, режим динамического повышения базовых приоритетов этих потоков разрешен.

Для отмены или возобновления режима динамического изменения базового приоритета только одного потока используется функция `SetThreadPriorityBoost`, которая имеет следующий прототип:

```
BOOL  SetThreadPriorityBoost(
  HANDLE  hThread,               // дескриптор потока
  BOOL    DisablePriorityBoost   // состояние повышения приоритета
);
```

Эта функция работает аналогично функции `SetProcessPriorityBoost`, но только для одного потока с дескриптором `hThread`.

Чтобы определить, разрешен ли режим динамического повышения базового приоритета для какого-то конкретного потока, используется функция `GetThreadPriorityBoost`, которая имеет следующий прототип:

```
BOOL  GetThreadPriorityBoost(
  HANDLE  hThread,               // дескриптор потока
  PBOOL   pDisablePriorityBoost  // состояние повышения приоритета
);
```

Эта функция работает так же, как и функция `GetProcessPriorityBoost`, но только для одного потока с дескриптором `hThread`.

В листинге 4.16 приведена программа, которая демонстрирует работу функций `SetProcessPriorityBoost`, `GetProcessPriorityBoost`, `SetThreadPriorityBoost` и `GetThreadPriorityBoost`. Еще раз отметим, что эти функции работают корректно только в операционной системе Windows 2000. В операционной системе Windows 98 эти функции всегда возвращают значение `FALSE`.

Листинг 4.16. Управление динамическим изменением приоритетов потоков

```
#include <windows.h>
#include <conio.h>

int main()
{
  HANDLE  hProcess, hThread;
  BOOL    bPriorityBoost;

  // получаем псевдодескриптор текущего процесса
  hProcess = GetCurrentProcess();

  // узнаем режим динамического повышения приоритетов для процесса
  if (!GetProcessPriorityBoost(hProcess, &bPriorityBoost))
  {
```

```
    _cputs("Get process priority boost failed.\n");
    _cputs("Press any key to exit.\n");
    _getch();
    return GetLastError();
  }
  _cprintf("The process priority boost = %d.\n", bPriorityBoost);

  // выключаем режим динамического повышения приоритетов для процесса
  if (!SetProcessPriorityBoost(hProcess, TRUE))
  {
    _cputs("Set process priority boost failed.\n");
    _cputs("Press any key to exit.\n");
    _getch();
    return GetLastError();
  }

  // получаем псевдодескриптор текущего потока
  hThread = GetCurrentThread();
  // узнаем режим динамического повышения приоритетов для потока
  if (!GetThreadPriorityBoost(hThread, &bPriorityBoost))
  {
    _cputs("Get process priority boost failed.\n");
    _cputs("Press any key to exit.\n");
    _getch();
    return GetLastError();
  }
  _cprintf("The thread priority boost = %d.\n", bPriorityBoost);

  // включаем режим динамического повышения приоритетов для потока
  if (!SetThreadPriorityBoost(hThread, FALSE))
  {
    _cputs("Set process priority boost failed.\n");
    _cputs("Press any key to exit.\n");
    _getch();
    return GetLastError();
  }
  _cputs("Press any key to exit.\n");
  _getch();

  return 0;
}
```

Часть II

Синхронизация потоков и процессов

Глава 5

Синхронизация

5.1. Непрерывные действия и команды

Действием называется изменение контекста потока или, другими словами, действием можно назвать любую последовательность команд, которая изменяет контекст потока. Под *контекстом действия* понимается только та часть контекста потока, которая используется этим действием. В простейшем случае можно считать, что контекст действия определяется только переменными, которые используются этим действием. Действие называется *непрерывным*, если оно удовлетворяет следующим двум требованиям:

- не прерывается во время своего исполнения;
- контекст действия изменяется только самим действием.

Если непрерывное действие представляется одной командой микропроцессора, то оно называется *непрерывной инструкцией* или *командой*.

В англоязычной литературе для обозначения непрерывного действия используется термин atomic action. Поэтому в русскоязычной литературе можно встретить также термины *атомарное действие* и *неделимое действие*, используемые для обозначения непрерывных действий. Более подробное обсуждение этого вопроса можно найти в книге Грегори Р. Эндрюса "Основы многопоточного, параллельного и распределенного программирования".

Теперь поговорим подробнее о тех требованиях, которым должны удовлетворять непрерывные действия.

Сначала рассмотрим первое требование, которое касается непрерывности действия. Действие может быть прервано только сигналом прерывания, который устанавливает соответствующий флаг микропроцессора. Этот сигнал может генерироваться как внешним устройством, требующим обслуживания, так и самим микропроцессором для обработки исключительных ситуаций, возникающих при выполнении программы. Будем считать, что обработка исключительных ситуаций неизбежна для нормального продолжения работы

программы или ее завершения. Поэтому для того, чтобы обеспечить непрерывность действия, необходимо запретить обработку сигнала прерывания от внешних устройств во время выполнения этого действия. Однако отметим, что такой подход работает только в однопроцессорных системах, т. к. в мультипроцессорных системах действия могут выполняться параллельно разными процессорами. Причем возможна такая ситуация, в которой контексты параллельных действий пересекаются. В этом случае даже запрещение обработки прерываний на каждом микропроцессоре не обеспечивает непрерывности действий, т. к. может быть нарушено второе условие непрерывности действия.

То есть, собственно говоря, второе требование к непрерывности действия и обеспечивает непрерывность действия на мультипроцессорных системах или, другими словами, запрещает действию, исполняемому одним процессором, изменять контекст действия, исполняемого другим процессором.

5.2. Определение синхронизации

Если рассматривать параллельные процессы абстрактно, то *синхронизация процессов* — это есть достижение некоторого фиксированного соотношения (порядка) между сигналами, которыми обмениваются эти процессы. В программировании рассматриваются параллельные процессы, которые являются программами, исполняемыми процессором. Поэтому, чтобы избежать технических подробностей, связанных с обменом сигналами между процессами, упростим задачу. То есть будем рассматривать не параллельные процессы, а параллельные потоки. В этом случае обмен сигналами между потоками может происходить только через глобальные переменные. Отсюда следует, что установить некоторый порядок выполнения инструкций потоков можно только посредством проверки этими потоками значений глобальных переменных. Для формализации этого подхода определим *условные непрерывные действия* путем введения оператора `await`, который имеет следующий синтаксис:

```
await(логическое условие) действие;
```

Будем считать, что этот оператор работает следующим образом. Он ждет, пока логическое условие не примет значение "истина", и как только это случилось, то выполняется действие. Причем ожидание истинности значения логического условия и выполнение действия происходят непрерывным образом. То есть оператор `await` является непрерывным.

Теперь можно дать определение синхронизации, которое является более естественным для практики программирования. Под *синхронизацией потоков* понимается исполнение этими потоками условных непрерывных действий. Рассмотрим частные случаи синхронизации, которые наиболее часто встречаются на практике.

Если оператор `await` имеет следующий вид:

```
await(логическое условие);
```

то он просто ждет оповещения о выполнении некоторого логического условия. Этот случай называется *условной синхронизацией*, а само логическое условие также называется *событием*. В этом случае часто говорят, что оператор `await` ждет наступления некоторого события.

Если оператор `await` имеет вид:

```
await(TRUE) действие;
```

то происходит безусловное выполнение непрерывного действия. Этот случай называется *взаимным исключением*, а программный код, исполняемый внутри непрерывного действия, называется *критической секцией*.

5.3. Программная реализация синхронизации

Сначала кратко поговорим о том, как же оператор `await` может быть реализован на практике. Фактически на однопроцессорной системе исполнение оператора `await` аналогично исполнению программы обработки прерывания. В этом случае сигнал прерывания можно рассматривать как оповещение о том, что логическое условие приняло значение "истина", а действие является программой обработки прерывания, которая своей первой командой запрещает прерывания. В мультипроцессорных системах нужно также обеспечить, чтобы контексты программ обработки прерываний не пересекались.

Теперь перейдем к частным случаям синхронизации, которые могут быть реализованы и без использования механизма обработки прерываний. Сначала рассмотрим условную синхронизацию, а затем проблему взаимного исключения.

Задача условной синхронизации

Для постановки этой задачи рассмотрим два потока `thread_1` и `thread_2`, которые работают следующим образом. Поток `thread_1` выполняет некоторые действия, а затем ждет наступления события `event`, после которого выполняет другие действия. В свою очередь поток `thread_2` также выполняет некоторые действия, а после их завершения оповещает поток `thread_1` о наступлении события `event`. Затем поток `thread_2` выполняет оставшиеся действия. Такая синхронизация работы потоков и называется *задачей условной синхронизации*.

Схематически программный код потоков `thread_1` и `thread_2`, который решает поставленную задачу условной синхронизации, приведен ниже.

```
bool  event = false;      // событие event


void  thread_1()          // поток thread_1
```

```
{
  actions_before_event();     // действия до наступления события
  while(!event);              // ждем, пока событие не произошло
  actions_after_event();      // действия после наступления события
}

void  thread_2()          // поток thread_2
{
  some_actions();             // действия, о которых оповещает событие
  event = true;               // отмечаем о наступлении события
  other_actions();            // действия, происходящие после события
}
```

Рассматривая этот код, во-первых, заметим следующее — фактически наступление некоторого события равносильно выполнению некоторого действия. Поэтому событие часто и определяют как действие. В потоке `thread_1` это действие обозначается функцией `some_actions()`.

Во-вторых, как видно из приведенного программного кода, для решения задачи условной синхронизации для двух потоков достаточно определить глобальную булеву переменную `event`, начальное значение которой установить в `false`. Затем в потоке `thread_2` установить значение этой переменной в `true` после наступления события `event`. Тогда поток `thread_1` ждет наступления события `event` посредством оператора `while(!event)`, который циклически проверяет значение булевой переменной `event` до тех пор, пока эта переменная не примет значение `true`. Очевидно, что подобным образом задача условной синхронизации может быть решена и для произвольного количества потоков, ждущих наступления события `event`.

Задача взаимного исключения

Теперь рассмотрим задачу взаимного исключения. Чтобы упростить рассуждения, эта задача будет сформулирована только для двух параллельных потоков. Сначала предположим, что два параллельных потока работают с одним и тем же ресурсом, который в этом случае называется *разделяемым* или *совместно используемым ресурсом*. Далее считаем, что в каждом потоке программный код, который осуществляет доступ к этому ресурсу, заключен в свою критическую секцию. Тогда задача взаимного исключения для двух потоков может быть сформулирована следующим образом: обеспечить двум потокам взаимоисключающий доступ к некоторому совместно используемому ресурсу. Причем решение этой задачи должно удовлетворять следующим требованиям:

- требование безопасности — в любой момент времени в своей критической секции может находиться только один поток;

- требование поступательности — потоки не могут блокировать работу друг друга, ожидая разрешения на вход в критическую секцию;
- требование справедливости — каждый поток получает доступ в критическую секцию за ограниченное время.

Ниже приведено простейшее из известных решений задачи взаимного исключения для двух потоков, которое было опубликовано Гэри Л. Петерсоном в 1981 году.

```
bool  x1 = false;
bool  x2 = false;
int  q;    // номер потока, которому предоставляется очередь входа в
           // критическую секцию
void  thread_1()   // поток thread_1
{
  while(true)
  {
    non_critical_section_1();    // код вне критической секции
    x1 = true;     // поток thread_1 хочет войти в критическую секцию
    q = 2;         // предоставить очередь потоку thread_2
    while(x2 && q == 2);     // ждем, пока в критической секции находится
                             // поток thread_2
    critical_section_1();    // входим в критическую секцию
    x1 = false;    // поток thread_1 находится вне критической секции
  }
}

void  thread_2()  // поток thread_2
{
  while(true)
  {
    non_critical_section_2();    // код вне критической секции
    x2 = true;     // поток thread_2 хочет войти в критическую секцию
    q = 1;         // предоставить очередь потоку thread_1
    while(x1 && q == 1);     // ждем, пока в критической секции находится
                             // поток thread_1
    critical_section_2();    // входим в критическую секцию
    x2 = false;    // поток thread_2 находится вне критической секции
  }
}
```

Покажем, что это решение удовлетворяет трем вышеперечисленным требованиям, предъявляемым к решению задачи взаимного исключения.

Сначала рассмотрим требование безопасности. Поток `thread_1` входит в критическую секцию `critcal_section_1()` только в том случае, если не выполняется условие `(x2 && q == 2)`. Это эквивалентно тому, что выполняется условие `!(x2 && q == 2)`, которое в свою очередь можно привести к условию `(!x2 || q == 1)`, используя логический закон отрицания конъюнкции. Кроме того, заметим, что если поток `thread_1` находится в критической секции, то выполняется условие `(x1 == true)`. Теперь введем предикат:

```
Q1 = x1 && (!x2 || q == 1)
```

который является инвариантом критической секции `critical_section_1()`. То есть, если поток `thread_1` находится внутри своей критической секции, то предикат `Q1` принимает значение "истина". Аналогично определим предикат

```
Q2 = x2 && (!x1 || q == 2)
```

который является инвариантом критической секции `critical_section_2()`. Теперь найдем истинностное значение предиката `(Q1 && Q2)`. Используя законы логики высказываний, получим:

```
Q1 && Q2  ≡
≡  (x1 && (!x2 || q == 1)) && (x2 && (!x1 || q == 2))  ≡
≡  x1 && x2 && (!x1 || q == 2) && (!x2 || q == 1)  ≡
≡  x1 && x2 && ((!x1 && !x2) || (!x1 && q == 1) || (!x2 && q == 2) ||
   (q ==1 && q == 2))  ≡
≡  x1 && x2 && ((!x1 && !x2) || (!x1 && q == 1) || (!x2 && q == 2))  ≡
≡  (x1 && !x1 && x2 && !x2) || ((x1 && x2 && !x1 && q == 1) ||
   (x1 && x2 && !x2 && q == 2)  ≡
≡  false
```

Символ ≡ обозначает равносильность или, другими словами, логическую эквивалентность выражений.

В результате получили, что предикат `(Q1 && Q2)` имеет тождественно ложное истинностное значение. Отсюда следует, что потоки `thread_1` и `thread_2` не могут одновременно находиться в своих критических секциях. Таким образом, доказано, что решение Петерсона удовлетворяет требованию безопасности.

Теперь рассмотрим требование поступательности. Поток `thread_1` может быть заблокирован, только если выполняется условие `(x2 && q == 2)`. Аналогично, поток `thread_2` может заблокироваться только при выполнении

условия `(x1 && q == 1)`. Рассмотрим предикат `(x2 && q == 2) && (x1 && q == 1)` и найдем его истинностное значение. Получим:

```
(x2 && q == 2) && (x1 && q == 1)  ≡
≡  x1 && x2 && q == 1 && q == 2  ≡  false
```

т. к. переменная `q` не может принимать одновременно значения 1 и 2. То есть этот предикат имеет тождественно ложное значение. Откуда следует, что потоки `thread_1` и `thread_2` не могут быть заблокированы одновременно. Таким образом, доказано, что решение Петерсона удовлетворяет требованию поступательности.

Наконец покажем, что решение Петерсона удовлетворяет требованию справедливости. Для этого предположим противное, т. е. поток `thread_1` заблокирован. Тогда должно выполняться условие `(x2 && q == 2)`. Из истинности этого условия следует, что выполняются условия `(x2 == true)` и `(q == 2)`. Но из выполнения условия поступательности следует, что поток `thread_2` не может быть заблокирован одновременно с потоком `thread_1`. Следовательно, должны также выполняться условия `(x2 == false)` или `(q == 1)`. Получили противоречие с нашим предположением. Поэтому наше предположение неверно и поток `thread_1` не может быть заблокирован.

Таким образом, доказано, что алгоритм Петерсона решает задачу взаимного исключения для двух потоков. Решение задачи взаимного исключения для произвольного количества потоков является более сложным.

Подводя итог изложенного материала, можно сделать следующий вывод: синхронизация потоков может осуществляться чисто программными средствами, используя глобальные переменные.

5.4. Аппаратная реализация синхронизации

Программная реализация синхронизации возможна только на однопроцессорных системах. Как уже было сказано выше, для синхронизации параллельных потоков, каждый из которых выполняется отдельным процессором, необходимо обеспечить взаимоисключающий доступ этих потоков к общим переменным. Для этого в каждом процессоре существуют специальные инструкции, которые изменяют содержимое памяти и не прерываются во время своего исполнения. При исполнении такой инструкции процессор "запирает" шину передачи данных, блокируя доступ к памяти другим процессорам. Мы рассмотрим только две из таких инструкций.

Инструкция "проверить и установить"

В наборе инструкций процессора Motorola 68000 существует инструкция `tas` (test and set), которая имеет следующую реализацию:

```
int  tas(int& target)
{
```

```
    int  temp;

    temp = target;
    target = 1;
    return temp;
}
```

Выполняется она непрерывным образом, т. е. выполнение этой инструкции не прерывается. Используя эту инструкцию, можно реализовать взаимоисключающий доступ параллельных потоков к критической секции по следующему алгоритму:

```
int  lock = 0;      // переменная, которая запирает вход в критическую
                    // секцию
void  thread()
{
  while(true)
  {
    while(tas(lock));          // ждать, пока замок закрыт
    critical_section();        // критическая секция
    lock = 0;                  // открыть замок
    non_critical_section();    // остальной код
  }
}
```

Покажем, что этот алгоритм удовлетворяет требованию безопасности. Для этого предположим противное, т. е. что критические секции выполняются одновременно в двух разных потоках. Это может произойти только в том случае, если инструкции `tas(lock)`, исполняемые в разных потоках, прервали друг друга. Но это невозможно, т. к. инструкция `tas` выполняется непрерывным образом. Следовательно, наше предположение неверно и в критической секции может находиться только один из потоков.

Теперь покажем, что этот алгоритм удовлетворяет требованию поступательности. Для этого предположим противное, т. е. что все потоки одновременно заблокированы на выполнении своих циклов `while(tas(lock))`. Но это возможно только в том случае, если начальное значение переменной `lock` было равно 1, что противоречит тексту программы. Следовательно, наше предположение неверно и требование поступательности выполняется.

Наконец поговорим о справедливости алгоритма. В общем случае выполнение этого требования зависит от арбитра шины данных, который управляет доступом микропроцессоров к шине данных. Для выполнения условия справедливости необходимо, чтобы арбитр шины обеспечивал справедливый доступ процессоров к шине данных.

Инструкция "обменять содержимое переменных"

В наборе инструкций процессоров семейства Intel x86 существует инструкция `xchg` (exchange), которая имеет следующую реализацию:

```
void  xchg(register int r, int  x)
{
  int  temp;

  temp = r;
  r = x;
  x = temp;
}
```

и выполняется непрерывным образом, т. е. выполнение этой инструкции не прерывается. Используя эту инструкцию, можно реализовать взаимоисключающий доступ параллельных потоков к критической секции по следующему алгоритму:

```
int  lock = 0;      // переменная, которая запирает вход в критическую
                    // секцию
void  thread()
{
  while(true)
  {
    register int  key;          // ключ к замку

    key = 1;
    while(key == 1)             // ждать, пока замок закрыт
      xchg(key, lock);
    critical_section();         // критическая секция
    lock = 0;                   // открыть замок
    non_critical_section();     // остальной код
  }
}
```

По аналогии с алгоритмом, использующим инструкцию `tas`, можно показать, что приведенный алгоритм также удовлетворяет двум первым требованиям, выдвигаемым к решению задачи взаимного исключения.

Для того чтобы терминологию, относящуюся к аппаратной синхронизации потоков, сделать независимой от типов команд, которые используются в процессорах для этих целей, ввели такое понятие, как *спин блокировка* (spinlock) или *активное ожидание*. Спин блокировкой называется цикл `while` с непре-

рываемой командой процессора, который ждет разрешения на вход в критическую секцию.

В заключение этого раздела сделаем два важных замечания. Во-первых, аппаратные алгоритмы синхронизации могут использоваться любым количеством параллельных потоков. Во-вторых, как программные, так аппаратные алгоритмы синхронизации имеют один существенный недостаток: впустую тратится процессорное время в циклах `while`, ждущих разрешения на вход в критическую секцию. Поэтому все эти алгоритмы получили общее название *алгоритмы, занимающиеся ожиданием* (busy waiting algorithms), или *алгоритмы активного ожидания.*

5.5. Примитивы синхронизации

Примитивом синхронизации называется программное средство высокого уровня для решения задач синхронизации. Обычно примитивы синхронизации реализованы как объекты ядра операционной системы, которые предназначены для решения задач синхронизации потоков и процессов. В пользовательских программах доступ к примитивам синхронизации выполняется или посредством вызова функций, которые работают с объектами синхронизации, или при помощи специальных инструкций, которые встроены в язык программирования. Часто языки программирования, в которые встроены объекты синхронизации, называются *языками системного программирования.* Мы рассмотрим реализацию примитивов синхронизации только для случая однопроцессорных систем. Поэтому непрерывность действий будем обеспечивать запрещением прерываний.

Теперь перейдем непосредственно к реализации примитивов синхронизации. Для этих целей будем использовать символический код в стиле языка программирования C++, избегая при этом деталей реализации функций, которые затруднят изучение сути вопроса.

Во-первых, чтобы избежать "занятия активным ожиданием", будем блокировать исполнение потоков. Для этого с каждым примитивом синхронизации свяжем очередь заблокированных потоков. Ниже приведена спецификация такой очереди:

```
class Thread { /* ... */ };          // класс потоков

class ThreadQueue                    // класс очередей потоков
{
  Thread*    tp;        // список потоков

  void  IncludeThreadToList(Thread&  t);   // включить поток в список
  Thread&  ExcludeThreadFromList();        // исключить поток из очереди
```

```
  void  SuspendThread(Thread&  t);          // заблокировать поток
  void  ResumeThread(Thread&  t);           // разблокировать поток
public:
  ThreadQueue(): tp(NULL) {}                // конструктор
  ~ThreadQueue() { /* очищаем список потоков */ }    // деструктор

  void  EnqueueThread(Thread&  t)           // поставить поток в очередь
  {
    IncludeThreadToList(t);
    SuspendThread(t);
  }

  bool  DequeueThread()                     // исключить поток из очереди
  {
    if(tp == NULL)
      return false;
    else
    {
      ResumeThread(ExcludeThreadFromList());
      return true;
    }
  }
};
```

Теперь можно перейти к реализации примитивов синхронизации. Рассмотрим реализацию только двух примитивов синхронизации: `condition` (условие) и `semaphore` (семафор), наиболее часто встречающихся на практике. Другие примитивы синхронизации могут быть реализованы подобным им образом. После реализации этих примитивов покажем, как они могут применяться для решения задач синхронизации параллельных потоков.

Примитив синхронизации *condition* (условие)

Ниже приведен класс `Condition`, определяющий одноименный примитив:

```
class Condition
{
  bool    event;
  ThreadQueue  tq;                  // очередь потоков
public:
  Condition(bool  b): event(b) {}    // конструктор
```

```
  ~Condition() {}                    // деструктор

  void  Signal()   // сигнализировать о том, что условие выполнено
  {
    disable_interrupt();      // запрещаем прерывания
    if(!tq.DequeueThread())
      event = true;
    enable_interrupt();       // разрешаем прерывания
  }

  void  Wait(Thread t)   // ждать выполнения условия
  {
    disable_interrupt();      // запрещаем прерывания
    if(event)
      event = false;          // сбрасываем условие
    else
      tq.EnqueueThread(t);    // ставим поток в очередь ожидания
    enable_interrupt();       // разрешаем прерывания
  }
};
```

Этот примитив может использоваться для решения как задачи условной синхронизации, так и задачи взаимного исключения. Приведем решение задачи условного ожидания, используя класс (примитив синхронизации) `Condition`. В следующем примере поток `thread_2` сигнализирует потоку `thread_1` о наступлении события `event`.

```
Condition  event(false);      // событие event

void  thread_1()      // поток thread_1
{
  actions_before_event();     // действия до наступления события
  event.Wait(thread_1);       // ждем, пока событие не произошло
  actions_after_event();      // действия после наступления события
}

void  thread_2()      // поток thread_2
{
  some_actions();             // действия, о которых оповещает событие
  event.Signal();             // отмечаем о наступлении события
  other_actions();            // действия, происходящие после события
}
```

Как видно, в этом случае поток `thread_1` не тратит процессорное время на ожидание события `event`, а блокируется до наступления этого события.

Теперь приведем решение задачи взаимного исключения, используя примитив синхронизации `Condition`. В этом примере примитив синхронизации `event` управляет входом в критическую секцию потока `thread`.

```
  Condition  event(true);     // переменная, которая разрешает вход в
                              // критическую секцию

  void  thread()
  {
    while(true)
    {
      event.Wait(thread);         // ждать разрешения на вход
      critical_section();         // критическая секция
      event.Signal();             // разрешить вход
      non_critical_section();     // остальной код
    }
  }
```

Примитив синхронизации *semaphore* (семафор)

Ниже приведен класс `Semaphore`, определяющий одноименный примитив.

```
class Semaphore
{
  unsigned  counter;          // счетчик
  ThreadQueue  tq;            // очередь потоков
public:
  Semaphore(unsigned  n): counter(n) {}     // конструктор
  ~Semaphore() {}                           // деструктор

  void  Signal()   // сигнализировать о том, что условие выполнено
  {
    disable_interrupt();      // запрещаем прерывания
    if(!tq.DequeueThread())
      ++counter;
    enable_interrupt();       // разрешаем прерывания
  }

  void  Wait(Thread  t)  // ждать выполнения условия
  {
```

```
    disable_interrupt();      // запрещаем прерывания
    if(counter > 0)
      --counter;              // сбрасываем условие
    else
      tq.EnqueueThread(t);    // ставим поток в очередь ожидания
    enable_interrupt();       // разрешаем прерывания
  }
};
```

При помощи примитива `semaphore` (семафор) также могут быть решены задачи условной синхронизации и взаимного исключения. Эти решения полностью совпадают с решениями, использующими примитив `condition` (условие) за исключением того, что семафор должен принимать значения 0 и 1 вместо `true` и `false` соответственно.

Семафор, который может принимать только значения 0 или 1 называется *бинарным*. Такой семафор собственно ничем не отличается от примитива синхронизации `condition` (условие). Если же семафор может принимать любые положительные целочисленные значения, то такой семафор называется *считающим*. Считающие семафоры используются для подсчета количества ресурсов, производимых параллельными потоками. Такие потоки обычно называются *производителями*. Уменьшают значение считающего семафора потоки, которые потребляют эти ресурсы, такие потоки называются *потребителями*.

Сделаем еще одно замечание относительно семафоров, которое, впрочем, касается и примитива синхронизации `condition` (условие). Если очередь семафора обслуживается по алгоритму FIFO, то семафор называется *сильным*, а в противном случае *слабым*. На этот момент следует обратить внимание при разработке приложений. Если в спецификациях операционной системы ничего не сказано о порядке обслуживания очереди потоков, ждущих некоторого события или положительного значения семафора, то не нужно делать на этот счет никаких предположений. Так как потоки, стоящие в очереди к примитиву синхронизации, могут обслуживаться как по алгоритму FIFO, так и учитывая их приоритеты.

Глава 6

Синхронизация потоков в Windows

6.1. Критические секции

В операционных системах Windows проблема взаимного исключения для параллельных потоков, выполняемых в контексте одного процесса, решается при помощи объекта типа `CRITICAL_SECTION`, который не является объектом ядра операционной системы. Для работы с объектами типа `CRITICAL_SECTION` используются следующие функции:

```
// инициализация критической секции
VOID  InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
// вход в критическую секцию
VOID  EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
// попытка войти в критическую секцию
BOOL  TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
// выход из критической секции
VOID  LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
// разрушение объекта критическая секция
VOID  DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Каждая из этих функций имеет единственный параметр, указатель на объект типа `CRITICAL_SECTION`. Все эти функции, за исключением `TryEnterCriticalSection`, не возвращают значения. Функция `TryEnterCriticalSection` возвращает ненулевое значение, если поток вошел в критическую секцию или уже находится в ней, в противном случае функция возвращает значение `FALSE`. Отметим также, что функция `TryEnterCriticalSection` поддерживается только операционной системой Windows 2000.

Кратко рассмотрим порядок работы с этими функциями. Для этого предположим, что при проектировании программы мы выделили в параллельных потоках критические секции, в которых используется ресурс, разделяемый

этими потоками. Тогда мы определяем в нашей программе объект типа CRITICAL_SECTION и считаем, что имя этого объекта логически связано с используемым разделяемым ресурсом. Перед тем как начать работу с объектом типа CRITICAL_SECTION, его необходимо инициализировать. Для этого и предназначена функция InitializeCriticalSection. После инициализации нашего объекта типа CRITICAL_SECTION мы в каждом из параллельных потоков пред входом в критическую секцию вызываем функцию EnterCriticalSection, которая исключает одновременный вход в критические секции, выполняющиеся в параллельных потоках и связанные с нашим разделяемым ресурсом. После завершения работы с разделяемым ресурсом поток должен покинуть свою критическую секцию, что выполняется посредством вызова функции LeaveCriticalSection. После окончания работы с объектом типа CRITICAL_SECTION необходимо освободить все системные ресурсы, которые использовались этим объектом. Для этой цели служит функция DeleteCriticalSection.

Теперь покажем работу этих функций. Для этого сначала рассмотрим пример, в котором выполняются несинхронизированные параллельные потоки, а затем синхронизируем их работу, используя критические секции. Программа, которая иллюстрирует работу несинхронизированных потоков, приведена в листинге 6.1.

Листинг 6.1. Пример работы несинхронизированных потоков

```
#include <windows.h>
#include <iostream.h>

DWORD WINAPI thread(LPVOID)
{
  int i,j;

  for (j = 0; j < 10; ++j)
  {
    // выводим строку чисел j
    for (i = 0; i < 10; ++i)
    {
      cout << j << ' ' << flush;
      Sleep(17);
    }
    cout << endl;
  }
```

```
  return 0;
}

int main()
{
  int i,j;
  HANDLE  hThread;
  DWORD  IDThread;

  hThread=CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
  if (hThread == NULL)
    return GetLastError();

for (j = 10; j < 20; ++j)
  {
    for (i = 0; i < 10; ++i)
    {
      // выводим строку чисел j
      cout << j << ' ' << flush;
      Sleep(17);
    }
    cout << endl;
  }
  // ждем, пока поток thread закончит свою работу
  WaitForSingleObject(hThread, INFINITE);

  return 0;
}
```

В программе из листинга 6.1 каждый из потоков `main` и `thread` выводит строки одинаковых чисел. Но из-за параллельной работы потоков каждая выведенная строка может содержать не равные между собой элементы. Наша задача будет заключаться в следующем: нужно так синхронизировать потоки `main` и `thread`, чтобы в каждой строке выводились только равные между собой числа. Программа из листинге 6.2 показывает решение этой задачи с помощью объекта типа `CRITICAL_SECTION`.

Листинг 6.2. Пример синхронизации потоков при помощи объекта CRITICAL_SECTION

```
#include <windows.h>
#include <iostream.h>

CRITICAL_SECTION cs;

DWORD WINAPI thread(LPVOID)
{
  int i,j;

  for (j = 0; j < 10; ++j)
  {
    // входим в критическую секцию
    EnterCriticalSection (&cs);
    for (i = 0; i < 10; ++i)
    {
      cout << j << ' '<< flush;
      Sleep(7);
    }
    cout << endl;
    // выходим из критической секции
    LeaveCriticalSection(&cs);
  }

  return 0;
}

int main()
{
  int i,j;
  HANDLE  hThread;
  DWORD  IDThread;

  // инициализируем критическую секцию
  InitializeCriticalSection(&cs);

  hThread=CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
```

```
  if (hThread == NULL)
    return GetLastError();

  for (j = 10; j < 20; ++j)
  {
    // входим в критическую секцию
    EnterCriticalSection(&cs);
    for (i = 0; i < 10; ++i)
    {
      cout << j << ' ' << flush;
      Sleep(7);
    }
    cout << endl;
    // выходим из критической секции
    LeaveCriticalSection(&cs);
  }
  // ждем, пока поток thread закончит свою работу
  WaitForSingleObject(hThread, INFINITE);
  // закрываем критическую секцию
  DeleteCriticalSection(&cs);

  return 0;
}
```

Теперь рассмотрим использование функции `TryEnterCriticalSection`. Для этого просто заменим в программе из листинга 6.2 вызовы функции `EnterCriticalSection` на вызовы функции `TryEnterCriticalSection` и будем отмечать успешные входы потоков в свои критические секции. Еще раз подчеркнем, что функция `TryEnterCriticalSection` работает только на платформе операционной системы Windows 2000. Программа, в которой для входа в критические секции используются функции `TryEnterCriticalSection`, приведена в листинге 6.3.

Листинг 6.3. Пример попытки входа в критическую секцию

```
#define _WIN32_WINNT 0x0500

#include <windows.h>
#include <iostream.h>
```

```
CRITICAL_SECTION cs;

DWORD WINAPI thread(LPVOID)
{
  int i,j;

  for (j = 0; j < 10; ++j)
    // попытка войти в критическую секцию
    if (TryEnterCriticalSection (&cs))
    {
      for (i = 0; i < 10; i++)
        cout << j << flush;
      cout << endl;
      // выход из критической секции
      LeaveCriticalSection(&cs);
    }

  return 0;
}

int main()
{
  int i,j;
  HANDLE  hThread;
  DWORD  IDThread;
  // инициализируем критическую секцию
  InitializeCriticalSection(&cs);

  hThread=CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
  if (hThread == NULL)
    return GetLastError();

  for (j = 10; j < 20; j++)
    // попытка войти в критическую секцию
    if (TryEnterCriticalSection(&cs))
    {
      for (i = 0; i < 10; i++)
        cout << j << ' ' << flush;
      cout << endl;
```

```
      // выход из критической секции
      LeaveCriticalSection(&cs);
    }
  // ждем завершения работы потока thread
  WaitForSingleObject(hThread, INFINITE);
  // удаляем критическую секцию
  DeleteCriticalSection(&cs);

  return 0;
}
```

В заключение этого раздела отметим следующее. Так как объекты типа `CRITICAL_SECTION` не являются объектами ядра операционной системы, то работа с ними происходит несколько быстрее, чем с объектами синхронизации, которые являются объектами ядра операционной системы. Это происходит потому, что обращение к объектам ядра операционной системы требует дополнительной работы на переключение контекстов потоков из режима пользователя в защищенный режим ядра операционной системы. Поэтому при разработке многопоточных приложений для решения задач взаимного исключения, как правило, используют объекты типа `CRITICAL_SECTION`.

6.2. Объекты синхронизации и функции ожидания

В операционных системах Windows *объектами синхронизации* называются объекты ядра, которые могут находиться в одном из двух состояний: *сигнальном* (signaled) и *несигнальном* (nonsignaled). Объекты синхронизации могут быть разбиты на четыре класса.

К первому классу относятся собственно объекты синхронизации, т. е. те, которые служат только для решения задач синхронизации параллельных потоков. К таким объектам синхронизации в Windows относятся:

- мьютекс (mutex);
- событие (event);
- семафор (semaphore).

Ко второму классу объектов синхронизации относится *ожидающий таймер* (waitable timer), который переходит в сигнальное состояние по истечении заданного интервала времени.

К третьему классу синхронизации относятся объекты, которые переходят в сигнальное состояние по завершении своей работы:

- работа (job);

- процесс (process);
- поток (thread).

К четвертому классу относятся объекты синхронизации, которые переходят в сигнальное состояние после получения сообщения об изменении содержимого объекта. К ним относятся:

- изменение состояния каталога (change notification);
- консольный ввод (console input).

В этой части мы будем рассматривать только следующие объекты синхронизации: мьютексы, события, семафоры, а также потоки и процессы. Остальные объекты, которые могут использоваться в функциях ожидания, будут рассмотрены по мере их изучения в следующих главах.

Теперь перейдем к функциям ожидания. *Функции ожидания* в Windows это такие функции, параметрами которых являются объекты синхронизации. Эти функции обычно используются для блокировки потоков. Сама блокировка потока выполняется следующим образом. Если дескриптор объекта синхронизации является параметром функции ожидания, а сам объект синхронизации находится в несигнальном состоянии, то поток, вызвавший эту функцию ожидания, блокируется до перехода этого объекта синхронизации в сигнальное состояние. Сейчас мы будем использовать только две функции ожидания `WaitForSingleObject` и `WaitForMultipleObject`. Остальные функции ожидания будут описаны в части, посвященной асинхронному вызову процедур в Windows.

Для ожидания перехода в сигнальное состояние одного объекта синхронизации используется функция `WaitForSingleObject`, которая имеет следующий прототип:

```
DWORD  WaitForSingleObject(
  HANDLE  hHandle,         // дескриптор объекта
  DWORD   dwMilliseconds   // интервал ожидания в миллисекундах
);
```

Функция `WaitForSingleObject` в течение интервала времени, равного значению параметра `dwMilliseconds`, ждет перехода объекта синхронизации, дескриптор которого задается параметром `hHandle`, в сигнальное состояние. Если значение параметра `dwMilliseconds` равно нулю, то функция только проверяет состояние объекта синхронизации. Если же значение параметра `dwMilliseconds` равно `INFINITE`, то функция ждет перехода объекта синхронизации в сигнальное состояние бесконечно долго.

В случае успешного завершения функция `WaitForSingleObject` возвращает одно из следующих значений:

- `WAIT_OBJECT_0` — объект перешел в сигнальное состояние;

- WAIT_ABANDONED — забытый мьютекс;
- WAIT_TIMEOUT — время ожидания истекло.

Значение WAIT_OBJECT_0 означает, что объект синхронизации находился или перешел в сигнальное состояние.

Значение WAIT_ABANDONED означает, что объектом синхронизации являлся мьютекс, который не освободился завершившимся потоком. В этом случае мьютекс освобождается операционной системой и поэтому также переходит в сигнальное состояние. Такой мьютекс иногда называется *забытый* или *заброшенный* мьютекс (abandoned mutex).

Значение WAIT_TIMEOUT означает, что время ожидания истекло, а объект синхронизации так и не перешел в сигнальное состояние.

В случае неудачи функция WaitForSingleObject возвращает значение WAIT_FAILED.

В листинге 6.4 приведен пример программы, которая использует функцию WaitForSingleObject для ожидания завершения потока. Отметим также, что эта функция уже неоднократно использовалась нами ранее для ожидания завершения работы потоков.

Листинг 6.4. Пример использования функции WaitForSingleObject

```
#include <windows.h>
#include <iostream.h>

void thread()
{
  int i;

  for (i = 0; i < 10 ; ++i)
  {
    cout << i << ' ' << flush;
    Sleep(100);
  }
  cout << endl;
}

int main()
{
  HANDLE   hThread;
  DWORD  dwThread;
```

```
  hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread, NULL,
                         0, &dwThread);
  if (hThread == NULL)
    return GetLastError();

  // ждем завершения потока thread
  if(WaitForSingleObject(hThread, INFINITE) != WAIT_OBJECT_0)
  {
    cout << "Wait for single object failed." << endl;
    cout << "Press any key to exit." << endl;
    cin.get();
  }
  // закрываем дескриптор потока thread
  CloseHandle(hThread);

  return 0;
}
```

Для ожидания перехода в сигнальное состояние нескольких объектов синхронизации или одного из нескольких объектов синхронизации используется функция `WaitForMultipleObject`, которая имеет следующий прототип:

```
DWORD  WaitForMultipleObjects(
  DWORD  nCount,                  // количество объектов
  CONST  HANDLE  *lpHandles,      // массив дескрипторов объектов
  BOOL   bWaitAll,                // режим ожидания
  DWORD  dwMilliseconds           // интервал ожидания в миллисекундах
);
```

Функция `WaitForMultipleObjects` работает следующим образом. Если значение параметра `bWaitAll` равно `TRUE`, то эта функция в течение интервала времени, равного значению параметра `dwMilliseconds`, ждет пока все объекты синхронизации, дескрипторы которых заданы в массиве `lpHandles`, перейдут в сигнальное состояние. Если же значение параметра `bWaitAll` равно `FALSE`, то эта функция в течение заданного интервала времени ждет, пока любой из заданных объектов синхронизации перейдет в сигнальное состояние. Если значение параметра `dwMilliseconds` равно нулю, то функция только проверяет состояние объектов синхронизации. Если же значение параметра `dwMilliseconds` равно `INFINITE`, то функция ждет перехода объектов синхронизации в сигнальное состояние бесконечно долго. Количество объектов синхронизации, ожидаемых функцией `WaitForMultipleObjects`,

не должно превышать значения MAXIMUM_WAIT_OBJECTS. Также отметим, что объекты синхронизации не должны повторяться.

В случае успешного завершения функция WaitForMultipleObjects возвращает следующие значения:

- от WAIT_OBJECT_0 до (WAIT_OBJECT_0 + nCount - 1);
- от WAIT_ABANDONED_0 до (WAIT_ABANDONED_0 + nCount - 1);
- WAIT_TIMEOUT.

Интерпретация значений, возвращаемых функцией WaitForMultipleObjects, зависит от значения входного параметра bWaitAll. Сначала рассмотрим случай, когда значение этого параметра равно TRUE. Тогда возвращаемые значения интерпретируются следующим образом:

- любое из возвращаемых значений, находящихся в диапазоне от WAIT_OBJECT_0 до (WAIT_OBJECT_0 + nCount - 1), означает, что все объекты синхронизации находились или перешли в сигнальное состояние;
- любое из возвращаемых значений, находящихся в диапазоне от WAIT_ABANDONED_0 до (WAIT_ABANDONED_0 + nCount - 1), означает, что все объекты синхронизации находились или перешли в сигнальное состояние и по крайней мере один их них был забытым мьютексом;
- возвращаемое значение WAIT_TIMEOUT означает, что время ожидания истекло и не все объекты синхронизации перешли в сигнальное состояние.

Теперь рассмотрим случай, когда значение входного параметра bWaitAll равно FALSE. В этом случае значения, возвращаемые функцией WaitForMultipleObjects, интерпретируются следующим образом:

- любое из возвращаемых значений, находящихся в диапазоне от WAIT_OBJECT_0 до (WAIT_OBJECT_0 + nCount - 1), означает, что, по крайней мере, один из объектов синхронизации находился или перешел в сигнальное состояние. Индекс дескриптора этого объекта в массиве определяется как разница между возвращаемым значением и величиной WAIT_OBJECT_0;
- любое из возвращаемых значений, находящихся в диапазоне от WAIT_ABANDONED_0 до (WAIT_ABANDONED_0 + nCount - 1), означает, что одним из объектов синхронизации, перешедшим в сигнальное состояние, является забытый мьютекс. Индекс дескриптора этого мьютекса в массиве определяется как разница между возвращаемым значением и величиной WAIT_OBJECT_0;
- возращаемое значение WAIT_TIMEOUT означает, что время ожидания истекло, и ни один из объектов синхронизации не перешел в сигнальное состояние.

В случае неудачи функция WaitForMultipleObjects возвращает значение WAIT_FAILED.

В листинге 6.5 приведен пример программы, которая использует функцию `WaitForMultipleObjects` для ожидания завершения двух потоков.

Листинг 6.5. Пример использования функции `WaitForMultipleObjects` для ожидания завершения двух потоков

```
#include <windows.h>
#include <iostream.h>

void thread_0()
{
  int i;

  for (i = 0; i < 5 ; ++i)
  {
    cout << i << ' ' << flush;
    Sleep(7);
  }
  cout << endl;
}

void thread_1()
{
  int i;

  for (i = 5; i < 10 ; ++i)
  {
    cout << i << ' ' << flush;
    Sleep(7);
  }
  cout << endl;
}

int main()
{
  HANDLE   hThread[2];
  DWORD  dwThread[2];

  // запускаем первый поток
```

```
  hThread[0] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread_0,
        NULL, 0, &dwThread[0]);
  if (hThread[0] == NULL)
    return GetLastError();
  // запускаем второй поток
  hThread[1] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread_1,
        NULL, 0, &dwThread[1]);
  if (hThread[1] == NULL)
    return GetLastError();

  // ждем завершения потоков thread_1 и thread_2
  if (WaitForMultipleObjects(2, hThread, TRUE, INFINITE) == WAIT_FAILED)
  {
    cout << "Wait for multiple objects failed." << endl;
    cout << "Input any char to exit." << endl;
    cin.get();
  }

  // закрываем дескрипторы потоков thread_0 и thread_1
  CloseHandle(hThread[0]);
  CloseHandle(hThread[1]);

  return 0;
}
```

6.3. Мьютексы

Для решения проблемы взаимного исключения между параллельными потоками, выполняющимися в контекстах разных процессов, в операционных системах Windows используется объект ядра *мьютекс*. Слово мьютекс происходит от английского слова mutex, которое в свою очередь является сокращением от выражения **mut**ual **ex**clusion, что на русском языке значит "взаимное исключение". Мьютекс находится в сигнальном состоянии, если он не принадлежит ни одному потоку. В противном случае мьютекс находится в несигнальном состоянии. Одновременно мьютекс может принадлежать только одному потоку.

Потоки, ждущие сигнального состояния мьютекса, обслуживаются в порядке FIFO, т. е. потоки становятся в очередь к мьютексу с дисциплиной обслуживания "первый пришел — первый вышел". Однако если поток ждет

наступления асинхронного события, то функции ядра могут исключить поток из очереди к мьютексу для обслуживания наступления этого события. После этого поток становится в конец очереди мьютекса.

Создается мьютекс вызовом функции `CreateMutex`, которая имеет следующий прототип:

```
HANDLE  CreateMutex(
  LPSECURITY_ATTRIBUTES  lpMutexAttributes,  // атрибуты защиты
  BOOL     bInitialOwner,    // начальный владелец мьютекса
  LPCTSTR  lpName            // имя мьютекса
);
```

Пока значение параметра `LPSECURITY_ATTRIBUTES` будем устанавливать в `NULL`. Это означает, что атрибуты защиты заданы по умолчанию, т. е. дескриптор мьютекса не является наследуемым и доступ к мьютексу открыт для всех пользователей. Теперь перейдем к другим параметрам.

Если значение параметра `bInitialOwner` равно `TRUE`, то мьютекс сразу переходит во владение потоку, которым он был создан. В противном случае вновь созданный мьютекс свободен. Поток, создавший мьютекс, имеет все права доступа к этому мьютексу.

Значение параметра `lpName` определяет уникальное имя мьютекса для всех процессов, выполняющихся под управлением операционной системы. Это имя позволяет обращаться к мьютексу из других процессов, запущенных под управлением этой же операционной системы. Длина имени не должна превышать значение `MAX_PATH`. Значением параметра `lpName` может быть пустой указатель `NULL`. В этом случае система создает безымянный мьютекс. Отметим также, что имена мьютексов являются чувствительными к нижнему и верхнему регистрам.

В случае удачного завершения функция `CreateMutex` возвращает дескриптор созданного мьютекса. В случае неудачи эта функция возвращает значение `NULL`. Если мьютекс с заданным именем уже существует, то функция `CreateMutex` возвращает дескриптор этого мьютекса, а функция `GetLastError`, вызванная после функции `CreateMutex`, вернет значение `ERROR_ALREADY_EXISTS`.

Мьютекс захватывается потоком посредством любой функции ожидания, а освобождается функцией `ReleaseMutex`, которая имеет следующий прототип:

```
BOOL  ReleaseMutex (
  HANDLE  hMutex    // дескриптор мьютекса
);
```

В случае успешного завершения функция `ReleaseMutex` возвращает ненулевое значение, а в случае неудачи — `FALSE`. Если поток освобождает мьютекс, которым он не владеет, то функция `ReleaseMutex` возвращает значение `FALSE`.

Для доступа к существующему мьютексу поток может использовать одну из функций CreateMutex или OpenMutex. Функция CreateMutex используется в тех случаях, когда поток не знает, создан или нет мьютекс с указанным именем другим потоком. В этом случае значение параметра bInitialOwner нужно установить в FALSE, т. к. невозможно определить какой из потоков создает мьютекс. Если поток использует для доступа к уже созданному мьютексу функцию CreateMutex, то он получает полный доступ к этому мьютексу.

Для того чтобы получить доступ к уже созданному мьютексу поток может также использовать функцию OpenMutex, которая имеет следующий прототип:

```
HANDLE  OpenMutex (
  DWORD    dwDesiredAccess,    // доступ к мьютексу
  BOOL     bInheritHandle      // свойство наследования
  LPCTSTR  lpName              // имя мьютекса
);
```

Параметр dwDesiredAccess этой функции может принимать одно из двух значений:

- MUTEX_ALL_ACCESS — полный доступ;
- SYNCHRONIZE — синхронизация.

В первом случае поток получает полный доступ к мьютексу. Во втором случае поток может использовать мьютекс только в функциях ожидания, чтобы захватить мьютекс, или в функции ReleaseMutex для его освобождения.

Параметр bInheritHandle определяет свойство наследования мьютекса. Если значение этого параметра равно TRUE, то дескриптор открываемого мьютекса является наследуемым. В противном случае дескриптор не наследуется.

В случае успешного завершения функция OpenMutex возвращает дескриптор открытого мьютекса, в случае неудачи эта функция возвращает значение NULL.

Покажем пример использования мьютекса для синхронизации потоков из разных процессов. Для этого сначала рассмотрим пример несинхронизированных потоков. Программы несинхронизированных процессов приведены в листингах 6.6, 6.7.

Листинг 6.6. Несинхронизированные потоки, выполняющиеся в разных процессах

```
#include <windows.h>
#include <iostream.h>

int main()
```

```
{
  char  lpszAppName[] = "C:\\ConsoleProcess.exe";
  STARTUPINFO  si;
  PROCESS_INFORMATION  pi;

  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb = sizeof(STARTUPINFO);

  // создаем новый консольный процесс
  if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
      NULL, NULL, NULL, &si, &pi))
  {
    cout << "The new process is not created." << endl;
    cout << "Press any key to exit." << endl;
    cin.get();

    return GetLastError();
  }

  // выводим на экран строки
  for (int j = 0; j < 10; ++j)
  {
    for (int i = 0; i < 10; ++i)
    {
      cout << j << ' ' << flush;
      Sleep(10);
    }
    cout << endl;
  }

  // ждем, пока дочерний процесс закончит работу
  WaitForSingleObject(pi.hProcess, INFINITE);

  // закрываем дескрипторы дочернего процесса в текущем процессе
  CloseHandle(pi.hThread);
  CloseHandle(pi.hProcess);

  return 0;
}
```

Листинг 6.7. Несинхронизированные потоки, выполняющиеся в разных процессах

```
#include <windows.h>
#include <iostream.h>

int main()
{
  int  i,j;

  for (j = 10; j < 20; ++j)
  {
    for (i = 0; i < 10; ++i)
    {
      cout << j << ' ' << flush;
      Sleep(5);
    }
    cout << endl;
  }

  return 0;
}
```

Кратко опишем работу программ, приведенных в листингах 6.6, 6.7. Первая из них запускает вторую программу, не создавая при этом новую консоль. После этого потоки из разных процессов начинают выводить числа в одну консоль. Из-за отсутствия синхронизации числа в одной строке могут быть из разных потоков. Для того чтобы избежать перемешивания чисел, синхронизируем вывод с помощью мьютекса. Далее, в листингах 6.8, 6.9, приведены модификации этих программ с использованием мьютекса для синхронизации работы этих потоков.

Листинг 6.8. Синхронизация потоков, выполняющихся в разных процессах, с использованием мьютекса

```
#include <windows.h>
#include <iostream.h>

int main()
{
```

```
HANDLE  hMutex;
char  lpszAppName[] = "C:\\ConsoleProcess.exe";
STARTUPINFO  si;
PROCESS_INFORMATION  pi;

// создаем мьютекс
hMutex = CreateMutex(NULL, FALSE, "DemoMutex");
if (hMutex == NULL)
{
  cout << "Create mutex failed." << endl;
  cout << "Press any key to exit." << endl;
  cin.get();

  return GetLastError();
}

ZeroMemory(&si, sizeof(STARTUPINFO));
si.cb = sizeof(STARTUPINFO);

// создаем новый консольный процесс
if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
    NULL, NULL, NULL, &si, &pi))
{
  cout << "The new process is not created." << endl;
  cout << "Press any key to exit." << endl;
  cin.get();

  return GetLastError();
}

// выводим на экран строки
for (int j = 0; j < 10; ++j)
{
  // захватываем мьютекс
  WaitForSingleObject(hMutex, INFINITE);
  for (int i = 0; i < 10; i++)
  {
    cout << j << ' ' << flush;
    Sleep(10);
```

```
    }
    cout << endl;
    // освобождаем мьютекс
    ReleaseMutex(hMutex);
  }
  // закрываем дескриптор мьютекса
  CloseHandle(hMutex);

  // ждем пока дочерний процесс закончит работу
  WaitForSingleObject(pi.hProcess, INFINITE);

  // закрываем дескрипторы дочернего процесса в текущем процессе
  CloseHandle(pi.hThread);
  CloseHandle(pi.hProcess);

  return 0;
}
```

Листинг 6.9. Синхронизации потоков, выполняющихся в разных процессах, с использованием мьютекса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hMutex;
  int    i,j;

  // открываем мьютекс
  hMutex = OpenMutex(SYNCHRONIZE, FALSE, "DemoMutex");
  if (hMutex == NULL)
  {
    cout << "Open mutex failed." << endl;
    cout << "Press any key to exit." << endl;
    cin.get();

    return GetLastError();
  }
```

```
  for (j = 10; j < 20; j++)
  {
    // захватываем мьютекс
    WaitForSingleObject(hMutex, INFINITE);
    for (i = 0; i < 10; i++)
    {
      cout << j << ' ' << flush;
      Sleep(5);
    }
    cout << endl;
    // освобождаем мьютекс
    ReleaseMutex(hMutex);
  }
  // закрываем дескриптор объекта
  CloseHandle(hMutex);

  return 0;
}
```

6.4. События

Событием называется оповещение о некотором выполненном действии. В программировании события используются для оповещения одного потока о том, что другой поток выполнил некоторое действие. Сама же задача оповещения одного потока о некотором действии, которое совершил другой поток, называется *задачей условной синхронизации* или иногда *задачей оповещения*. В операционных системах Windows события описываются объектами ядра Events. При этом различают два типа событий:

- события с ручным сбросом;
- события с автоматическим сбросом.

Различие между этими типами событий заключается в том, что событие с ручным сбросом можно перевести в несигнальное состояние только посредством вызова функции `ResetEvent`, а событие с автоматическим сбросом переходит в несигнальное состояние как при помощи функции `ResetEvent`, так и при помощи функции ожидания. При этом отметим, что если события с автоматическим сбросом ждут несколько потоков, используя функцию `WaitForSingleObject`, то из состояния ожидания освобождается только один из этих потоков.

Создаются события вызовом функции CreateEvent, которая имеет следующий прототип:

```
HANDLE  CreateEvent(
  LPSECURITY_ATTRIBUTES  lpSecurityAttributes,   // атрибуты защиты
  BOOL     bManualReset,       // тип события
  BOOL     bInitialState,      // начальное состояние события
  LPCTSTR  lpName              // имя события
);
```

Как и обычно, пока значение параметра lpSecurityAttributes будем устанавливать в NULL. Основную смысловую нагрузку в этой функции несут второй и третий параметры. Если значение параметра bManualReset равно TRUE, то создается событие с ручным сбросом, в противном случае — с автоматическим сбросом. Если значение параметра bInitialState равно TRUE, то начальное состояние события является сигнальным, в противном случае — несигнальным. Параметр lpName задает имя события, которое позволяет обращаться к нему из потоков, выполняющихся в разных процессах. Этот параметр может быть равен NULL, тогда создается безымянное событие.

В случае удачного завершения функция CreateEvent возвращает дескриптор события, а в случае неудачи — значение NULL. Если событие с заданным именем уже существует, то функция CreateEvent возвращает дескриптор этого события, а функция GetLastError, вызванная после функции CreateEvent, вернет значение ERROR_ALREADY_EXISTS.

Для перевода любого события в сигнальное состояние используется функция SetEvent, которая имеет следующий прототип:

```
BOOL  SetEvent(
  HANDLE  hEvent     // дескриптор события
);
```

При успешном завершении эта функция возвращает ненулевое значение, а в случае неудачи — FALSE.

В листинге 6.10 приведен пример программы, в которой безымянные события с автоматическим сбросом используются для синхронизации работы потоков, выполняющихся в одном процессе.

Листинг 6.10. Синхронизации потоков при помощи событий с автоматическим сбросом

```
#include <windows.h>
#include <iostream.h>

HANDLE hOutEvent, hAddEvent;
```

```
DWORD WINAPI thread(LPVOID)
{
  for (int i = 0; i < 10; ++i)
    if (i == 4)
    {
      SetEvent(hOutEvent);
      WaitForSingleObject(hAddEvent, INFINITE);
    }

  return 0;
}

int main()
{
  HANDLE  hThread;
  DWORD  IDThread;

  // создаем события с автоматическим сбросом
  hOutEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
  if (hOutEvent == NULL)
    return GetLastError();
  hAddEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
  if (hAddEvent == NULL)
    return GetLastError();

  // создаем поток thread
  hThread = CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
  if (hThread == NULL)
    return GetLastError();

  // ждем, пока поток thread выполнит половину работы
  WaitForSingleObject(hOutEvent, INFINITE);
  // выводим значение переменной
  cout << "A half of the work is done." << endl;
  cout << "Press any key to continue." << endl;
  cin.get();
  // разрешаем дальше работать потоку thread
  SetEvent(hAddEvent);
```

```
  WaitForSingleObject(hThread, INFINITE);
  CloseHandle(hThread);

  CloseHandle(hOutEvent);
  CloseHandle(hAddEvent);

  cout << "The work is done." << endl;

  return 0;
}
```

Теперь опишем функцию `ResetEvent`, которая используется для перевода любого события в несигнальное состояние. Эта функция имеет следующий прототип:

```
BOOL  ResetEvent(
  HANDLE  hEvent     // дескриптор события
);
```

При успешном завершении эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`.

Для освобождения потоков, ждущих сигнального состояния события с ручным сбросом, используется функция `PulseEvent`, которая имеет следующий прототип:

```
BOOL  PulseEvent(
  HANDLE  hEvent     // дескриптор события
);
```

При вызове этой функции все потоки, ждущие события с дескриптором `hEvent`, выводятся из состояния ожидания, а само событие сразу переходит в несигнальное состояние. В этом случае говорят, что все потоки, ждущие сигнального состояния события с ручным сбросом, подошли к *барьеру*, через который они все могут перешагнуть только при переходе события с ручным сбросом в сигнальное состояние.

Если функция `PulseEvent` вызывается для события с автоматическим сбросом, то из состояния ожидания выводится только один из ожидающих потоков. Если нет потоков, ожидающих сигнального состояния события с дескриптором `hEvent`, то состояние этого события остается несигнальным.

В листинге 6.11 приведена программа, использующая функцию `PulseEvent` для синхронизации потоков, выполняющихся в одном процессе.

Листинг 6.11. Синхронизации потоков при помощи событий с ручным сбросом

```
#include <windows.h>
#include <iostream.h>

HANDLE hOutEvent[2], hAddEvent;

DWORD WINAPI thread_1(LPVOID)
{
  for (int i = 0; i < 10; ++i)
    if (i == 4)
    {
      SetEvent(hOutEvent[0]);
      WaitForSingleObject(hAddEvent, INFINITE);
    }

  return 0;
}

DWORD CALLBACK thread_2(LPVOID)
{
  for (int i = 0; i < 10; ++i)
    if (i == 4)
    {
      SetEvent(hOutEvent[1]);
      WaitForSingleObject(hAddEvent, INFINITE);
    }

  return 0;
}

int main()
{
  HANDLE  hThread_1, hThread_2;
  DWORD  IDThread_1, IDThread_2;

  // создаем события с автоматическим сбросом
  hOutEvent[0] = CreateEvent(NULL, FALSE, FALSE, NULL);
```

```
if (hOutEvent[0] == NULL)
  return GetLastError();
hOutEvent[1] = CreateEvent(NULL, FALSE, FALSE, NULL);
if (hOutEvent[1] == NULL)
  return GetLastError();

// создаем событие с ручным сбросом
hAddEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
if (hAddEvent == NULL)
  return GetLastError();

// создаем потоки
hThread_1 = CreateThread(NULL, 0, thread_1, NULL, 0, &IDThread_1);
if (hThread_1 == NULL)
  return GetLastError();
hThread_2 = CreateThread(NULL, 0, thread_2, NULL, 0, &IDThread_2);
if (hThread_2 == NULL)
  return GetLastError();

// ждем, пока потоки-счетчики выполнят половину работы
WaitForMultipleObjects(2, hOutEvent, TRUE, INFINITE);
cout << "A half of the work is done." << endl;
cout << "Press any key to continue." << endl;
cin.get();

// разрешаем потокам-счетчикам продолжать работу
PulseEvent(hAddEvent);

// ждем завершения потоков
WaitForSingleObject(hThread_1, INFINITE);
WaitForSingleObject(hThread_2, INFINITE);

// закрываем дескрипторы
CloseHandle(hThread_1);
CloseHandle(hThread_2);

CloseHandle(hOutEvent[0]);
CloseHandle(hOutEvent[1]);
```

```
  CloseHandle(hAddEvent);

  cout << "The work is done." << endl;

  return 0;
}
```

Доступ к существующему событию можно открыть с помощью функции `CreateEvent` или `OpenEvent`.

Если для этой цели используется функция `CreateEvent`, то значения параметров `bManualReset` и `bInitialState` этой функции игнорируются, т. к. они уже установлены другим потоком, а поток, вызвавший эту функцию, получает полный доступ к событию с именем, заданным параметром `lpName`.

Теперь рассмотрим функцию `OpenEvent`, которая используется в том случае, если известно, что событие с заданным именем уже существует. Эта функция имеет следующий прототип:

```
HANDLE  OpenEvent(
  DWORD    dwDesiredAccess,    // флаги доступа
  BOOL     bInheritHandle,     // режим наследования
  LPCTSTR  lpName              // имя события
);
```

Параметр `dwDesiredAccess` определяет доступ к событию и может быть равен любой логической комбинации следующих флагов:

- `EVENT_ALL_ACCESS` — полный доступ;
- `EVENT_MODIFY_STATE` — модификация состояния;
- `SYNCHRONIZE` — синхронизация.

Эти флаги устанавливают следующие режимы доступа к событию:

- флаг `EVENT_ALL_ACCESS` означает, что поток может выполнять над событием любые действия;
- флаг `EVENT_MODIFY_STATE` означает, что поток может использовать функции `SetEvent` и `ResetEvent` для изменения состояния события;
- флаг `SYNCHRONIZE` означает, что поток может использовать событие в функциях ожидания.

В листингах 6.12, 6.13 приведены программы, иллюстрирующие синхронизацию потоков, выполняющихся в разных процессах, при помощи события с автоматическим сбросом. В этом примере также используется функция `OpenEvent` для доступа к уже существующему событию.

Листинг 6.12. Родительский процесс, ожидающий наступления события

```
#include <windows.h>
#include <iostream.h>

HANDLE hInEvent;
CHAR lpEventName[] = "InEventName";

int main()
{
  DWORD dwWaitResult;

  char szAppName[] = "D:\\ConsoleProcess.exe";
  STARTUPINFO si;
  PROCESS_INFORMATION pi;

  // создем событие, отмечающее ввод символа
  hInEvent = CreateEvent(NULL, FALSE, FALSE, lpEventName);
  if (hInEvent == NULL)
    return GetLastError();

  // запускаем процесс, который ждет ввод символа
  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb = sizeof(STARTUPINFO);
  if (!CreateProcess(szAppName, NULL, NULL, NULL, FALSE,
      CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi))
    return 0;
  // закрываем дескрипторы этого процесса
  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);

  // ждем оповещение о наступлении события о вводе символа
  dwWaitResult = WaitForSingleObject(hInEvent, INFINITE);
  if (dwWaitResult != WAIT_OBJECT_0)
    return dwWaitResult;

  cout << "A symbol has got." << endl;
```

```
  CloseHandle(hInEvent);

  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

Листинг 6.13. Дочерний процесс, устанавливающий событие в сигнальное состояние

```
#include <windows.h>
#include <iostream.h>

HANDLE hInEvent;
CHAR lpEventName[]="InEventName";

int main()
{
  char c;

  hInEvent = OpenEvent(EVENT_MODIFY_STATE, FALSE, lpEventName);
  if (hInEvent == NULL)
  {
    cout << "Open event failed." << endl;
    cout << "Input any char to exit." << endl;
    cin.get();

    return GetLastError();
  }

  cout << "Input any char: ";
  cin >> c;
  // устанавливаем событие о вводе символа
  SetEvent(hInEvent);
  // закрываем дескриптор события в текущем процессе
  CloseHandle(hInEvent);

  cin.get();
```

```
  cout << "Press any key to exit." << endl;
  cin.get();

  return 0;
}
```

Кратко опишем работу этих программ. Родительский процесс, программа которого приведена в листинге 6.12, создает событие с автоматическим сбросом, а затем запускает дочерний процесс, программа которого приведена в листинге 6.13. После этого родительский процесс ждет до тех пор, пока дочерний процесс не установит это событие в сигнальное состояние. Дождавшись этого, он завершает свою работу. Дочерний процесс открывает событие, созданное в родительском процессе, после чего вводит символ с консоли. Если символ введен, то он устанавливает событие в сигнальное состояние и завершает свою работу.

6.5. Семафоры

Семафоры в операционных системах Windows описываются объектами ядра `Semaphores`. Семафор находится в сигнальном состоянии, если его значение больше нуля. В противном случае семафор находится в несигнальном состоянии. Потоки, ждущие сигнального состояния семафора, обслуживаются в порядке FIFO, т. е. потоки становятся в очередь к семафору с дисциплиной обслуживания "первый пришел, первый вышел". Однако если поток ждет наступления асинхронного события, то функции ядра могут исключить поток из очереди к семафору для обслуживания наступления этого события. После этого поток становится в конец очереди семафора.

Создаются семафоры посредством вызова функции `CreateSemaphore`, которая имеет следующий прототип:

```
HANDLE  CreateSemaphore(
  LPSECURITY_ATTRIBUTES  lpSemaphoreAttribute,   // атрибуты защиты
  LONG     lInitialCount,    // начальное значение семафора
  LONG     lMaximumCount,    // максимальное значение семафора
  LPCTSTR  lpName            // имя семафора
);
```

Пока значение параметра `lpSemaphoreAttributes` будем устанавливать в `NULL`.

Основную смысловую нагрузку в этой функции несут второй и третий параметры. Значение параметра `lInitialCount` устанавливает начальное значение семафора, которое должно быть не меньше 0 и не больше его максимального значения, которое устанавливается параметром `lMaximumCount`.

Параметр lpName может указывать на имя семафора или содержать значение NULL. В последнем случае создается безымянный семафор.

В случае успешного завершения функция CreateSemaphore возвращает дескриптор семафора, в случае неудачи — значение NULL. Если семафор с заданным именем уже существует, то функция CreateSemaphor возвращает дескриптор этого семафора, а функция GetLastError, вызванная после функции CreateSemaphor, вернет значение ERROR_ALREADY_EXISTS.

Значение семафора уменьшается на 1 при его использовании в функции ожидания. Увеличить значение семафора можно посредством вызова функции ReleaseSemaphore, которая имеет следующий прототип:

```
BOOL  ReleaseSemaphore(
  HANDLE  hSemaphore,        // дескриптор семафора
  LONG    lReleaseCount,     // положительное число, на которое
                             // увеличивается значение семафора
  LPLONG  lpPreviousCount    // предыдущее значение семафора
);
```

В случае успешного завершения функция ReleaseSemaphore возвращает ненулевое значение, а в случае неудачи — FALSE. Если значение семафора плюс значение параметра lReleaseCount больше максимального значения семафора, то функция ReleaseSemaphore возвращает значение FALSE и значение семафора не изменяется.

Значение параметра lpPreviousCount этой функции может быть равно NULL. В этом случае предыдущее значение семафора не возвращается.

Доступ к существующему семафору можно открыть с помощью одной из функций CreateSemaphore или OpenSemaphore. Если для этой цели используется функция CreateSemaphore, то значения параметров lInitialCount и lMaximalCount этой функции игнорируются, т. к. они уже установлены другим потоком, а поток, вызвавший эту функцию, получает полный доступ к семафору с именем, заданным параметром lpName.

Теперь рассмотрим функцию OpenSemaphore, которая используется в случае, если известно, что семафор с заданным именем уже существует. Эта функция имеет следующий прототип:

```
HANDLE  OpenSemaphore(
  DWORD    dwDesiredAccess,    // флаги доступа
  BOOL     bInheritHandle,     // режим наследования
  LPCTSTR  lpName              // имя события
);
```

Параметр dwDesiredAccess определяет доступ к семафору и может быть равен любой логической комбинации следующих флагов:

- SEMAPHORE_ALL_ACCESS — полный доступ к семафору;

- SEMAPHORE_MODIFY_STATE — изменение состояния семафора;
- SYNCHRONIZE — синхронизация.

Опишем назначение этих флагов чуть подробнее.

- флаг SEMAPHORE_ALL_ACCESS устанавливает для потока полный доступ к семафору. Это означает, что поток может выполнять над семафором любые действия;
- флаг SEMAPHORE_MODIFY_STATE означает, что поток может использовать только функцию ReleaseSemaphore для изменения значения семафора;
- флаг SYNCHRONIZE означает, что поток может использовать семафор только в функциях ожидания. Отметим, что этот флаг поддерживается только на платформе Windows NT/2000.

В листинге 6.15 приведен пример программы, в которой считающий семафор используется для синхронизации работы потоков. Для этого сначала рассмотрим несинхронизированный вариант этой программы, который представлен в листинге 6.14.

Листинг 6.14. Несинхронизированные потоки

```
#include <windows.h>
#include <iostream.h>

volatile int a[10];

DWORD WINAPI thread(LPVOID)
{
  for (int i = 0; i < 10; i++)
  {
    a[i] = i + 1;
    Sleep(7);
  }

  return 0;
}

int main()
{
  int i;
  HANDLE  hThread;
  DWORD  IDThread;
```

```
   cout << "An initial state of the array: ";
  for (i = 0; i < 10; i++)
    cout << a[i] <<' ';
  cout << endl;

  // создаем поток, который готовит элементы массива
  hThread = CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
  if (hThread == NULL)
    return GetLastError();

  // поток main выводит элементы массива
  cout << "A modified state of the array: ";
  for (i = 0; i < 10; i++)
  {
    cout << a[i] << ' ' << flush;
    Sleep(11);
  }
  cout << endl;

  CloseHandle(hThread);

  return 0;
}
```

Теперь кратко опишем работу программы из листинга 6.14. Поток `thread` последовательно присваивает элементам массива `a` значения, которые на единицу больше, чем их индекс. Поток `main` последовательно выводит элементы массива `a` на консоль. Так как потоки `thread` и `main` не синхронизированы, то измененное состояние массива, которое выведет на консоль поток `main`, неизвестно. Наша задача состоит в том, чтобы поток `main` выводил на консоль элементы массива `a` сразу после их подготовки потоком `thread`. Для этого мы используем считающий семафор. Программа из листинга 6.15 показывает, как этот считающий семафор используется для синхронизации работы потоков.

Листинг 6.15. Синхронизации потоков с использованием семафора

```
#include <windows.h>
#include <iostream.h>

volatile int a[10];
```

```
HANDLE hSemaphore;

DWORD WINAPI thread(LPVOID)
{
  for (int i = 0; i < 10; i++)
  {
    a[i] = i + 1;
    // отмечаем, что один элемент готов
    ReleaseSemaphore(hSemaphore,1,NULL);
    Sleep(500);
  }

  return 0;
}

int main()
{
  int i;
  HANDLE  hThread;
  DWORD  IDThread;

  cout << "An initial state of the array: ";
  for (i = 0; i < 10; i++)
    cout << a[i] <<' ';
  cout << endl;
  // создаем семафор
  hSemaphore=CreateSemaphore(NULL, 0, 10, NULL);
  if (hSemaphore == NULL)
    return GetLastError();

  // создаем поток, который готовит элементы массива
  hThread = CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
  if (hThread == NULL)
    return GetLastError();

  // поток main выводит элементы массива
  // только после их подготовки потоком thread
  cout << "A final state of the array: ";
  for (i = 0; i < 10; i++)
```

```
  {
    WaitForSingleObject(hSemaphore, INFINITE);
    cout << a[i] << " \a" << flush;
  }
  cout << endl;

  CloseHandle(hSemaphore);
  CloseHandle(hThread);

  return 0;
}
```

Может возникнуть следующий вопрос: почему для решения этой задачи используется именно считающий семафор и почему его максимальное значение равно 10? Конечно, поставленную задачу можно было бы решить и другими способами. Но дело в том, что считающие семафоры предназначены именно для решения подобных задач. Считающие семафоры используются для синхронизации доступа к однотипным ресурсам, которые производятся одним или несколькими потоками, а потребляются другими — одним или несколькими потоками. В этом случае значение считающего семафора равно количеству произведенных ресурсов, а его максимальное значение устанавливается равным максимально возможному количеству таких ресурсов. При производстве единицы ресурса значение семафора увеличивается на единицу, а при потреблении единицы ресурса значение семафора уменьшается на единицу. В нашем примере ресурсами являются элементы массива, заполненные потоком `thread`, который является производителем этих ресурсов. В свою очередь, поток `main` является потребителем этих ресурсов — он выводит их на консоль. Так как в общем случае мы не можем сделать предположений о скоростях работы параллельных потоков, то максимальное значение считающего семафора должно быть установлено в максимальное количество производимых ресурсов. Если поток-потребитель ресурсов работает быстрее чем поток-производитель ресурсов, то, вызвав функцию ожидания считающего семафора, он вынужден будет ждать, пока поток-производитель не произведет очередной ресурс. Если же наоборот — поток-производитель работает быстрее чем поток-потребитель, то первый поток произведет все ресурсы и закончит свою работу, не дожидаясь, пока второй использует их. Такая синхронизация потоков-производителей и потребителей обеспечивает их максимально быструю работу.

Глава 7

Взаимоисключающий доступ к переменным

7.1. Атомарные операции

Иногда параллельным потокам необходимо выполнять некоторые несложные действия над общими переменными, исключая совместный доступ к этим переменным. Если в этом случае для синхронизации доступа к переменным использовать критические секции или мьютексы (*см. разд. 6.1*, *6.3*), то может возникнуть такая ситуация, что затраты на синхронизацию потоков значительно превышают затраты на выполнение самих операций. Для решения этой проблемы используются специальные функции, которые выполняют несложные действия над переменными, блокируя при этом доступ к этим переменным со стороны других потоков. Такие функции называются *блокирующими функциями* (interlocked functions). Блокирующие функции выполняют одну или несколько элементарных операций, которые объединяются в одну неделимую операцию, которая в этом случае также называется *атомарной операцией*. Блокирующие функции могут использоваться потоками, выполняющимися в разных процессах, для взаимоисключающего доступа к переменным, расположенным в разделяемой этими процессами памяти. Процессы могут разделять общую память при помощи механизма отображения файлов в память, который описан в *гл. 30*.

Атомарная операция обычно включает операцию, выполняющую некоторое действие, и, может быть, операцию сравнения, которая позволяет выполнять это действие при некотором условии. В качестве операций, которые выполняют действия, могут выступать операции замены значения переменной или арифметические операции. Такие операции характеризуют атомарную операцию и используются для ее названия.

В операционных системах Windows блокирующие функции можно разбить на четыре группы, принимая во внимание типы характерных элементарных операций, которые они выполняют. Эти блокирующие функции рассмотрены в разделах этой главы.

Заметим, что все блокирующие функции требуют, чтобы адреса переменных были выровнены на границу слова, т. е. были кратны 32. Для такого выравнивания адреса достаточно, чтобы переменная была объявлена в программе со спецификатором типа `long`, или `unsigned long`, или одним из их синонимов `LONG`, `ULONG` или `DWORD`.

7.2. Замена значения переменной

Для замены значения переменной в операционных системах Windows используется блокирующая функция `InterlockedExchange`, которая имеет следующий прототип:

```
LONG  InterlockedExchange(
  LPLONG  lpTarget,    // адрес переменной, значение которой заменяется
  LONG    lValue       // новое значение переменной
);
```

Эта функция предназначена для замены значения переменной, адрес которой задан параметром `lpTarget`, на новое значение, которое задано параметром `lValue`. Значение, возвращаемое функцией `InterlockedExchange`, равно старому значению изменяемой переменной.

Замечание

Адрес `lpTarget`, который указывает на изменяемую переменную, должен быть выровнен на границу слова, т. е. должен быть кратен 32.

В листинге 7.1 приведен пример использования функции `InterlockedExchange` для взаимоисключающей замены значения переменной. В этой программе поток `producer` производит товары — целые числа, а поток `consumer` потребляет эти товары. После того как товар произведен, он помещается потоком `producer` в контейнер, роль которого выполняет переменная `n`. Чтобы потребить товар, поток `consumer` должен забрать его из контейнера. Требуется, чтобы операции загрузки нового товара в контейнер и извлечение товара из контейнера не прерывали друг друга. Для выполнения этого требования в программе и используется функция `InterlockedExchange`.

Листинг 7.1. Пример использования функции `InterlockedExchange`

```
#include <windows.h>
#include <iostream.h>

volatile long n;
```

```
void producer()
{
  long goods = 0;
  for (;;)
  {
    ++goods;   // производим новое число
    InterlockedExchange((long*)&n, goods);  // помещаем число в контейнер
    Sleep(150);
  }
}

void consumer()
{
  long goods;
  for (;;)
  {
    Sleep(400);
    InterlockedExchange(&goods, n);  // извлекаем число из контейнера
    cout << "Goods are consumed: " << goods << endl;
  }
}

int main()
{
  HANDLE   hThread_p, hThread_c;
  DWORD  IDThread_p, IDThread_c;

  cout << "Press any key to terminate threads." << endl;
  // создаем потоки
  hThread_p = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)producer,
                           NULL, 0, &IDThread_p);
  if (hThread_p == NULL)
    return GetLastError();
  hThread_c = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)consumer,
                           NULL, 0, &IDThread_c);
  if (hThread_c == NULL)
    return GetLastError();

  cin.get();
```

```
  // прерываем выполнение потоков
  TerminateThread(hThread_p, 0);
  TerminateThread(hThread_c, 0);

  // закрываем дескрипторы потоков
  CloseHandle(hThread_c);
  CloseHandle(hThread_p);

  return 0;
}
```

7.3. Условная замена значения переменной

Для условной замены значения переменной в операционных системах Windows используется блокирующая функция `InterlockedCompareExchange`, которая имеет следующий прототип:

```
PVOID  InterlockedCompareExchange(
  PVOID  *Destination, // адрес переменной, значение которой заменяется
  PVOID  Exchange,     // новое значение переменной
  PVOID  Comperand     // значение для сравнения
);
```

Эта функция предназначена для замены значения переменной, адрес которой задан параметром `Destination`, на новое значение, которое задано параметром `Exchange`, при условии, что старое значение переменной равно значению, заданному параметром `Comperand`. Значение, возвращаемое функцией `InterlockedCompareExchange`, равно старому значению изменяемой переменной.

Замечание

Адреса всех переменных, используемых функцией `InterlockedCompareExchange`, должны быть выровнены на границу слова, т. е. должны быть кратны 32.

В листинге 7.2 приведен пример использования функции `InterlockedCompareExchange` для условной замены значения переменной. Эта программа отличается от программы из листинга 7.1 только тем, что поток `producer` будет помещать новое число в контейнер только в том случае, если поток `consumer` освободил контейнер от старого числа.

Листинг 7.2. Пример использования функции `InterlockedCompareExchange`

```
#include <windows.h>
#include <iostream.h>

volatile long n;

void producer()
{
  long goods = 0;
  for (;;)
  {
    ++goods;   // производим новое число
  // помещаем число в контейнер, если он пустой
    InterlockedCompareExchange((PVOID*)&n, (PVOID)goods, 0);
    Sleep(150);
  }
}

void consumer()
{
  long goods;
  for (;;)
  {
    Sleep(400);
    goods = n;  // извлекаем число из контейнера
    InterlockedExchange((LONG*)&n, 0);  // отмечаем, что контейнер пустой
    cout << "Goods are consumed: " << goods << endl;
  }
}

int main()
{
  HANDLE   hThread_p, hThread_c;
  DWORD  IDThread_p, IDThread_c;

  cout << "Press any key to terminate threads." << endl;
  // создаем потоки
```

```
  hThread_p = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)producer,
                           NULL, 0, &IDThread_p);
  if (hThread_p == NULL)
    return GetLastError();
  hThread_c = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)consumer,
                           NULL, 0, &IDThread_c);
  if (hThread_c == NULL)
    return GetLastError();

  cin.get();

  // прерываем выполнение потоков
  TerminateThread(hThread_p, 0);
  TerminateThread(hThread_c, 0);

  // закрываем дескрипторы потоков
  CloseHandle(hThread_c);
  CloseHandle(hThread_p);

  return 0;
}
```

7.4. Инкремент и декремент переменной

Для изменения значения переменной на единицу в операционных системах Windows используются следующие блокирующие функции: `InterlockedIncrement` и `InterlockedDecrement`.

Функция `InterlockedIncrement` имеет следующий прототип:

```
LONG  InterlockedIncrement(
  LPLONG  lpAddend  // адрес переменной для инкремента
);
```

Эта функция предназначена для увеличения значения переменной, адрес которой задан параметром `lpAddend`, на единицу. Значение, возвращаемое функцией `InterlockedIncrement`, равно новому значению изменяемой переменной.

Функция `InterlockedDecrement` имеет следующий прототип:

```
LONG  InterlockedDecrement(
  LPLONG  lpAddend  // адрес переменной для декремента
);
```

Эта функция предназначена для уменьшения значения переменной, адрес которой задан параметром `lpAddend`, на единицу. Значение, возвращаемое функцией `InterlockedDecrement`, равно новому значению изменяемой переменной.

Замечание

В этих функциях адрес изменяемой переменной, который задается параметром `lpTarget`, должен быть выровнен на границу слова, т. е. должен быть кратен 32.

В листинге 7.3 приведен пример использования функции `InterlockedIncrement` для взаимоисключающего увеличения значения переменной на 1. Эта программа отличается от программы из листинга 7.1 только тем, что поток `producer` корректирует значение числа прямо в контейнере, не используя для этого локальную переменную.

Листинг 7.3. Пример использования функции `InterlockedIncrement`

```
#include <windows.h>
#include <iostream.h>

volatile long n;

void producer()
{
  for (;;)
  {
    InterlockedIncrement((long*)&n);  /* изменяем число в контейнере */
    Sleep(150);
  }
}

void consumer()
{
  long goods;
  for (;;)
  {
    Sleep(400);
    InterlockedExchange(&goods, n);  // извлекаем число из контейнера

    cout << "Goods are consumed: " << goods << endl;
```

```
  }
}

int main()
{
  HANDLE   hThread_p, hThread_c;
  DWORD  IDThread_p, IDThread_c;

  cout << "Press any key to terminate threads." << endl;
  // создаем потоки
  hThread_p = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)producer,
    NULL, 0, &IDThread_p);
  if (hThread_p == NULL)
    return GetLastError();
  hThread_c = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)consumer,
    NULL, 0, &IDThread_c);
  if (hThread_c == NULL)
    return GetLastError();

  cin.get();

  // прерываем выполнение потоков
  TerminateThread(hThread_p, 0);
  TerminateThread(hThread_c, 0);

  // закрываем дескрипторы потоков
  CloseHandle(hThread_c);
  CloseHandle(hThread_p);

  return 0;
}
```

7.5. Изменение значения переменной

Для изменения значения переменной в операционных системах Windows используется блокирующая функция `InterlockedExchangeAdd`, которая имеет следующий прототип:

```
  LONG  InterlockedExchangeAdd (
    LPLONG  lpAddend,    // адрес переменной, значение которой изменяется
```

```
    LONG    Increment    // прибавляемое значение
  );
```

Эта функция прибавляет значение, заданное параметром `Increment`, к переменной, адрес которой задан параметром `lpAddend`. Значение, возвращаемое функцией `InterlockedExchangeAdd`, равно старому значению изменяемой переменной.

Замечание

Адрес изменяемой переменной, который задается параметром `lpAddend`, должен быть выровнен на границу слова, т. е. должен быть кратен 32.

В листинге 7.4 приведен пример использования функции `InterlockedExchangeAdd` для взаимоисключающего увеличения значения переменной. Эта программа отличается от программы из листинга 7.3 только тем, что значение переменной `n` увеличивается не на 1, а на 10.

Листинг 7.4. Пример использования функции `InterlockedExchangeAdd`

```
#include <windows.h>
#include <iostream.h>

volatile long n;

void producer()
{
  for (;;)
  {
    /* изменяем число в контейнере */
    InterlockedExchangeAdd((long*)&n, 10);
    Sleep(150);
  }
}

void consumer()
{
  long goods;
  for (;;)
  {
    Sleep(400);
```

```
    InterlockedExchange(&goods, n);  // извлекаем число из контейнера

    cout << "Goods are consumed: " << goods << endl;
  }
}

int main()
{
  HANDLE   hThread_p, hThread_c;
  DWORD  IDThread_p, IDThread_c;

  cout << "Press any key to terminate threads." << endl;
  // создаем потоки
  hThread_p = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)producer,
    NULL, 0, &IDThread_p);
  if (hThread_p == NULL)
    return GetLastError();
  hThread_c = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)consumer,
    NULL, 0, &IDThread_c);
  if (hThread_c == NULL)
    return GetLastError();

  cin.get();

  // прерываем выполнение потоков
  TerminateThread(hThread_p, 0);
  TerminateThread(hThread_c, 0);

  // закрываем дескрипторы потоков
  CloseHandle(hThread_c);
  CloseHandle(hThread_p);

  return 0;
}
```

Глава 8

Тупики

8.1. Определение тупиков

Говорят, что поток находится в *тупике* (deadlock), если он ждет событие, которое никогда не произойдет. Событие может никогда не произойти по следующим двум причинам:

- не существует потока, который оповещает о наступлении ожидаемого события;
- поток, оповещающий о наступлении ожидаемого события, существует, но сам находится в тупике.

Если в тупике находится хотя бы один из потоков процесса, то считается, что этот процесс также находится в тупике. Если мы рассматриваем систему, которая состоит из нескольких процессов, и хотя бы один из этих процессов находится в тупике, то и сама рассматриваемая система также находится в тупике.

Например, рассмотрим два потока одного процесса:

```
void  thread_1( )
{
  EnterCriticalSection(&cs1);
  cin >> a;
  EnterCriticalSection(&cs2);
  x = a - b;
  LeaveCriticalSection(&cs2);
  LeaveCriticalSection(&cs1);
}
void  thread_2( )
{
    EnterCriticalSection(&cs2);
```

```
    cin >> b;
    EnterCriticalSection(&cs1);
    y = a + b;
    LeaveCriticalSection(&cs1);
    LeaveCriticalsection(&cs2);
}
```

Если после операции ввода (`cin >> a`) в первом потоке выполнение этого потока прервется и управление будет передано второму потоку в функцию `EnterCriticalSection(&cs2)`, то процесс войдет в тупик. Это произойдет потому, что поток `thread_1` будет ждать освобождения входа в критическую секцию `cs2`, которая занята потоком `thread_2`, а поток `thread_2` будет ждать освобождения входа в критическую секцию `cs1`, которая занята потоком `thread_1`. Но эти события никогда не наступят. Таким образом, потоки `thread_1` и `thread_2` блокируют друг друга. Процесс находится в тупике, т. к. каждый поток ждет ресурс, захваченный другим потоком, при этом не освобождая принадлежащие ему ресурсы.

Аналогичная ситуация произойдет и в том случае, если после операции ввода (`cin >> b`) во втором потоке выполнение этого потока прервется и управление будет передано первому потоку в функцию `EnterCriticalSection(&cs1)`.

Очевидно, что при разработке программ необходимо избегать ситуаций, которые могут привести к тупикам, т. к. в этом случае система становится неработоспособной. В следующих разделах мы рассмотрим способы обнаружения, предотвращения и восстановления после тупиков.

8.2. Классификация системных ресурсов

Прежде чем разбираться с тупиками, нам необходимо более точно классифицировать ресурсы, используемые потоками, и рассмотреть специальные ориентированные графы, которые называются графами распределения ресурсов процесса. Этими вопросами мы сейчас и займемся. Системные ресурсы можно классифицировать по нескольким признакам, которые перечислены ниже.

Если классифицировать ресурсы по количеству потоков, которые могут одновременно иметь доступ к ним, то получим следующие классы ресурсов:

- *совместно используемые ресурсы*, т. е. ресурсы, каждый из которых может использоваться одновременно несколькими потоками (например — файл);
- *монопольные ресурсы*, т. е. ресурсы, каждый из которых не может одновременно использоваться несколькими потоками (например — квант процессорного времени).

Если классифицировать ресурсы по способу распределения этих ресурсов между потоками, то получим следующие классы ресурсов:

- *перераспределяемые ресурсы* (preemptable) — те, которые могут быть отобраны у потока, владеющего этим ресурсом, и переданы другому потоку (например, страницы реальной памяти в системах с виртуальной памятью);
- *неперераспределяемые ресурсы* (non-preemtable) — те, которые не могут быть отобраны у потока, владеющего ресурсом (например, принтер).

Если классифицировать ресурсы по времени их существования, то получим следующие классы ресурсов:

- *повторно используемые ресурсы* — те, которые после их освобождения одним потоком могут использоваться другим потоком (например, страницы виртуальной памяти);
- *потребляемые ресурсы* — те, которые исчезают после их использования потоком (например, сообщения).

Последние два класса ресурсов рассмотрим более подробно. Сначала остановимся на повторно используемых ресурсах. Как правило, к ним относятся физические ресурсы компьютера и файлы. Повторно используемые ресурсы имеют следующие свойства:

- количество единиц ресурса постоянно;
- ресурсы являются монопольными;
- ресурсы не перераспределяются.

Теперь перейдем к потребляемым ресурсам. Потребляемые ресурсы создаются потоками-производителями этих ресурсов, а используются потоками-потребителями. Причем после использования потребляемого ресурса он перестает существовать. Очевидно, что потребляемыми ресурсами могут быть только логические ресурсы компьютера. Потребляемые ресурсы характеризуются следующими свойствами:

- количество единиц ресурса может изменяться;
- ресурс может быть как монопольным, так и совместным;
- ресурс может быть как перераспределяемым, так и неперераспределяемым.

Примером совместно используемого потребляемого ресурса может быть сообщение, которое предназначено для нескольких процессов. Примером перераспределяемого потребляемого ресурса является квант процессорного времени, который выделяется потоку для работы. Так как поток может быть прерван другим потоком с более высоким приоритетом, то оставшееся от кванта время будет передано новому потоку.

8.3. Обнаружение тупиков

После классификации ресурсов можно перейти к обнаружению тупиков. Для этого определим *граф распределения ресурсов* процесса. Это такой ориентированный граф, вершины которого обозначают потоки, а дуга, изображенная на рис. 8.1, интерпретируется следующим образом:

- в случае повторно используемых ресурсов: поток T_2 запрашивает ресурс R, занятый потоком T_1;
- в случае потребляемых ресурсов: поток T_2 использует ресурс R, произведенный потоком T_1.

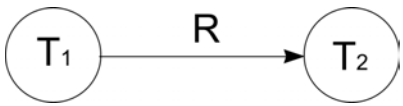


Рис. 8.1. Дуга графа распределения ресурсов процесса

Сначала рассмотрим процесс, в котором потоки используют только повторно используемые ресурсы. В этом случае поток такого процесса находится в тупике, если он бесконечно долго ждет ресурс, захваченный другим потоком. Критерием для обнаружения тупика в случае повторно используемых ресурсов является следующая теорема.

Теорема 1

Процесс, потоки которого используют только повторно используемые ресурсы, находится в тупике тогда и только тогда, когда граф распределения ресурсов этого процесса содержит цикл.

Доказательство

Докажем необходимость. Для этого рассмотрим процесс, который находится в тупике. Построим граф G распределения ресурсов этого процесса. Предположим противное, т. е., что этот граф не имеет циклов. Возьмем произвольную вершину T графа G и найдем все пути, началом которых служит эта вершина. По предположению каждый из этих путей не образует цикл. Следовательно, не исключена возможность следующей работы потоков. Потоки, которые являются концами путей, завершают свою работу и освобождают все захваченные ими ресурсы. Тогда будут разблокированы потоки, которые связаны с концевыми вершинами. Повторяя эти рассуждения, в конце концов, получим, что будет разблокирован поток, представленный вершиной T. Получили противоречие с условием, что рассматриваемый процесс находится в тупике. Следовательно, наше предположение неверно и в графе распределения ресурсов есть цикл.

Теперь докажем достаточность. Для этого предположим, что граф распределения ресурсов процесса содержит цикл: T_1, T_2, ..., T_n. Следовательно, поток T_2 будет заблокирован до тех пор, пока ресурс, требуемый этим потоком, не будет освобожден потоком T_1. Повторяя эти рассуждения и учитывая транзитивность блокировки потоков, получим, что поток T_n будет заблокирован до тех пор, пока поток T_1 не освободит ресурс, требуемый потоком T_2. Но этого не произойдет никогда, т. к. в свою очередь поток T_1 ждет ресурс, захваченный заблокированным потоком T_n. Следовательно, рассматриваемый процесс находится в тупике.

Теорема доказана.

Например, если потоки, рассмотренные в *разд. 8.1*, находятся в тупике, то граф распределения ресурсов процесса может быть таким, как это показано на рис. 8.2. Здесь под ресурсами понимаются критические секции, освобождения которых ждут потоки.

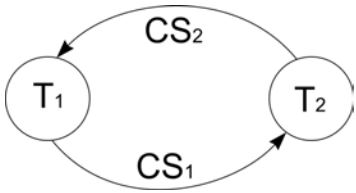
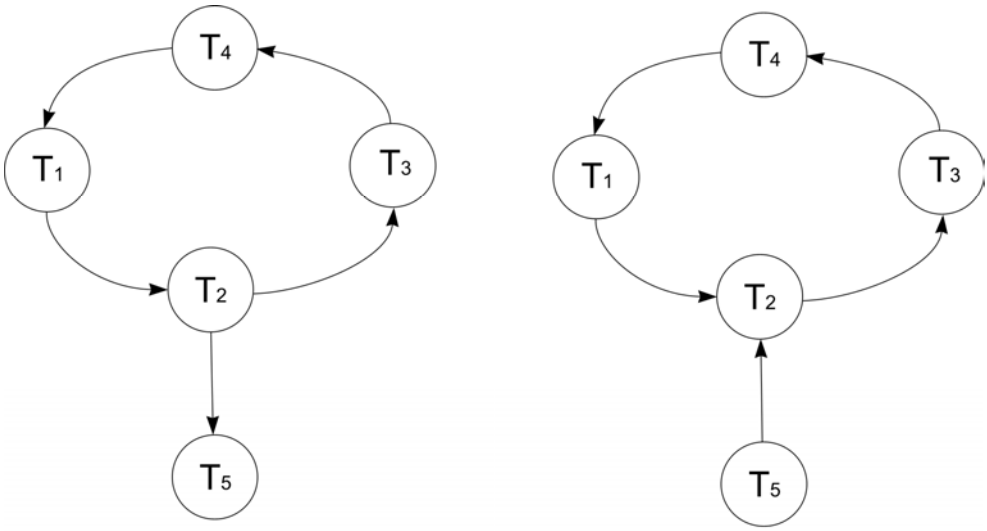


Рис. 8.2. Граф распределения ресурсов процесса (для тупика)



а) узел *б*) цикл, но не узел

Рис. 8.3. Примеры узла и цикла в графе потребляемых ресурсов процесса

Теперь рассмотрим процесс, в котором потоки используют только потребляемые ресурсы. Пусть G — граф распределения ресурсов такого процесса. Введем понятие узла. Подграф H графа G называется *узлом*, если выполняются следующие условия:

- через все вершины подграфа H проходит цикл;
- для каждой дуги, связывающей вершину из графа G, не входящую в подграф H, с вершиной из подграфа H, вершина из подграфа H является началом этой дуги.

На рис. 8.3 приведены примеры узла и цикла в графе потребляемых ресурсов процесса.

Теперь можно изложить критерий тупика для процесса, который использует только потребляемые ресурсы.

Теорема 2

Процесс, потоки которого используют только потребляемые ресурсы, находится в тупике тогда и только тогда, когда граф распределения ресурсов этого процесса содержит узел.

Доказывается эта теорема по аналогии с *теоремой 1.*

Из двух приведенных теорем следует, что для обнаружения тупика нужно построить граф распределения ресурсов процесса и проверить, содержит ли этот граф цикл для случая повторно используемых ресурсов или узел, для случая потребляемых ресурсов.

8.4. Восстановление заблокированного процесса

После обнаружения тупика должно быть выполнено *восстановление процесса*, которое заключается в разблокировании потоков этого процесса. Существуют три подхода к восстановлению заблокированного процесса:

- прекращение работы заблокированных потоков (termination);
- перераспределение ресурсов (preemption);
- откат на контрольную точку (rollback).

Рассмотрим каждый из этих подходов подробнее.

Самым простым решением проблемы тупика является прекращение работы всех заблокированных потоков процесса и освобождение всех ресурсов, захваченных этими потоками. Этот подход к разблокированию процесса является самым плохим, т. к. в этом случае нарушается целостность обрабатываемой информации. Это происходит потому, что очень сложно обеспечить

корректность состояния ресурсов в случае аварийного завершения работы одного потока другим потоком. Однако этот подход может применяться, если потоки обрабатывают только локальные данные и аварийное завершение этих потоков не вызовет нарушения целостности и непротиворечивости глобальных данных.

Если это возможно, то лучше всего разблокировать процесс путем перераспределения ресурсов между потоками этого процесса. То есть забрать ресурс у одного из заблокированных потоков и передать его другому заблокированному потоку, который ждет этот ресурс. В этом случае все потоки, за исключением одного, продолжат свою работу. Кроме того, поток, у которого забрали ресурс, также продолжит свою работу, после того как этот ресурс станет свободным. Этот подход является самым лучшим с концептуальной точки зрения, но он не всегда может быть реализован на практике по той причине, что не каждый ресурс является перераспределяемым.

Третий подход, использующий откат на контрольную точку, используется наиболее часто и является компромиссом между двумя первыми подходами. При использовании отката на контрольную точку должны быть решены две проблемы:

- корректное освобождение ресурсов, захваченных заблокированными потоками;
- восстановление контекста потока на момент, предшествующий тупику.

Для реализации этого подхода в программе устанавливаются такие точки, в которых запоминается состояние контекста потока. Эти точки называются *контрольными точками потока*. Изменение контекста потока между двумя контрольными точками называется *транзакцией* (tranzaction). Транзакция может быть зафиксирована или отменена. Отмена транзакции называется *откатом* (rollback). При откате контекст потока возвращается на контрольную точку, в которой не было тупика. При этом выполняется освобождение ресурсов, захваченных после прохождения контрольной точки, и восстановление контекста потока на момент прохождения контрольной точки. Если транзакция зафиксирована, то откат невозможен. Очевидно, что после выполнения отката одного из заблокированных потоков, остальные заблокированные потоки получат доступ к освобожденным этими потоками ресурсам и, следовательно, будут разблокированы.

Следует отметить, что при использовании откатов на контрольную точку нужно также решить, для какого из заблокированных потоков выполнить откат. Подход к решению этого вопроса зависит от работы, которую выполняют потоки. Например, можно выполнить откат заблокированного потока, который меньше всего захватил ресурсов после прохождения последней контрольной точки.

8.5. Предотвращение тупиков

Для обнаружения тупиков необходимо исследовать графы распределения ресурсов процесса, а затем в случае обнаружения тупика выполнить восстановление потоков, находящихся в тупике. Эта задача является довольно трудоемкой, поэтому лучше предпринимать некоторые стратегии при захвате ресурсов, которые позволяют избегать появления тупиков. Рассмотрим некоторые из этих стратегий и отметим недостатки каждой из них.

Первая стратегия, направленная на предотвращение тупиков, заключается в следующем: поток должен захватывать сразу все необходимые ему для работы ресурсы и только потом начинать свою работу. Очевидно, что в данном случае возникновение тупика невозможно, т. к. поток не будет запрашивать дополнительных ресурсов, а после окончания своей работы освободит все захваченные им ресурсы. Но данная стратегия имеет и свои недостатки:

- неэффективное использование ресурсов компьютера, т. к. они блокируются для использования другими потоками. То есть может возникнуть такая ситуация, что поток захватывает все необходимые ему для работы ресурсы, но не все эти ресурсы он использует одновременно. В этом случае ресурсы простаивают, хотя могли бы использоваться другими потоками;
- не всегда возможно определить полную потребность в ресурсах, необходимых для работы потока.

Вторая стратегия, направленная на предотвращение тупиков, заключается в следующем: если в процессе работы потоку требуется дополнительный ресурс, но он получает отказ на захват этого ресурса, то поток должен освободить все принадлежащие ему ресурсы. Очевидно, что в этом случае возникновение тупиков также невозможно, т. к. все ресурсы могут быть освобождены. Эта стратегия более предпочтительна, чем первая, т. к. не мешает эффективному использованию ресурсов. Но и она имеет один недостаток, а именно — системные затраты на откат потока и его повторный рестарт могут быть очень большими. Нет никакой гарантии того, что поток, запросивший некоторый ресурс и не получивший его сразу или по истечении некоторого конечного интервала времени, находится в тупике. Поэтому могут быть фальшивые откаты и рестарты потоков и процессов, что также снижает производительность системы в целом.

Третья стратегия, направленная на предотвращение тупиков, заключается в следующем: все ресурсы линейно упорядочиваются по типам, а в процессе своей работы поток может захватывать только те ресурсы, тип которых больше типа уже используемых им ресурсов. Эта стратегия также позволяет избегать тупиков, т. к. однотипные ресурсы захватываются все сразу, что исключает блокировку по каждому из типов ресурсов. Использование этой стратегии обычно не снижает эффективность использования ресурсов и позволяет избежать откатов и рестартов потоков. Но она также имеет свой

недостаток. Эта стратегия хорошо подходит потокам, для которых потребность в ресурсах может быть определена до запуска этого потока. Но она совершенно не пригодна для потоков, которые определяют потребность в ресурсах динамически в процессе своей работы. К сожалению, в системном программировании на практике чаще всего встречается именно такая ситуация, когда поток заранее не знает, сколько ресурсов определенного типа ему потребуется. В противном случае это, как правило, не системная, а прикладная программа, которая не обладает универсальностью.

Как видно из вышесказанного, не существует общей стратегии для предотвращения тупиков. Каждая из стратегий имеет свои достоинства и недостатки. Кроме того, слепое следование этим стратегиям приводит к неэффективной работе системы. При разработке систем, как правило, используют все из этих стратегий, но не в ущерб производительности системы. Поэтому в современных системах также возможны тупики и с этим приходится мириться.

8.6. Безопасное завершение потоков в Windows

После обнаружения тупика должно быть выполнено восстановление процесса, которое заключается в разблокировании потоков этого процесса. Как уже было сказано, восстановление невозможно без рестарта, по крайней мере, одного из заблокированных потоков. При этом возникает следующая задача оповещения потока об освобождении захваченных им ресурсов. Рассмотрим, как эту задачу можно решить средствами операционных систем Windows. Для этого в программе нужно определить дополнительные события, которые сообщают потоку о том, что ему нужно сделать: освободить ресурсы и закончить свою работу или продолжить свое исполнение.

Для иллюстрации этого подхода приведем пример программы, в которой выполняется разблокирование потоков. Прежде чем привести текст этой программы, кратко изложим суть ее работы. В рамках процесса выполняются два потока `main` и `marker`. Поток `marker` заполняет целыми числами элементы целочисленного массива `a[size]` при условии, что элемент массива является пустым. Мы предполагаем, что вначале все элементы массива `a[size]` являются пустыми и инициализированы нулями. Заполнение элементов происходит следующим образом: поток `marker` генерирует положительное случайное число и находит остаток от деления этого числа на `size`, а затем заполняет элемент, индекс которого равен этому остатку. В качестве заполнителя используется единица. Ясно, что когда-нибудь поток `marker` войдет в тупик, т. к. может сгенерировать индекс уже заполненного элемента. Задача потока `main` — обнаружить этот тупик и разблокировать поток `marker`. Разблокирование потока `marker` выполняется следующим образом: или

потоку `marker` разрешается продолжить свою работу, или он должен безопасно завершить свою работу. Под *безопасным завершением работы потока* мы понимаем корректное освобождение всех захваченных этим потоком ресурсов и завершение этого потока посредством вызова функции `ExitThread`. В листинге 8.1 приведен текст программы, выполняющей указанные действия.

Листинг 8.1. Восстановление работы процесса после обнаружения тупика

```
#include <windows.h>
#include <iostream.h>
#include <math.h>

const int size = 10;   // размерность массива
int    a[size];        // обрабатываемый массив
HANDLE  hDeadlock;     // сигнал о тупике
HANDLE  hAnswer[2];    // для обработки тупика

DWORD WINAPI marker(LPVOID)
{
  int    i;
  DWORD  dwValue;

  for (;;)
  {
    // вычисляем случайный индекс
    i = abs(rand()) % size;
    // проверяем, занят ли элемент
    if (!a[i])
      // нет, заполняем элемент
      a[i] = 1;
    else
    {
      // да, сигнализируем о тупике
      SetEvent(hDeadlock);
      // ждем ответа
      dwValue = WaitForMultipleObjects(
              2, hAnswer, FALSE, INFINITE);
      if (dwValue == WAIT_FAILED)
      {
```

```
        cerr << "Wait function failed." << endl;
        cerr << "Press any key to exit." << endl;
        cin.get();

        return GetLastError();
      }
      // вычисляем индекс сигнального объекта
      dwValue -= WAIT_OBJECT_0;
      switch (dwValue)
      {
      case 0:    // продолжаем работу
        continue;
      case 1:    // завершаем работу
        ExitThread(1);
        break;
      default:
        ExitThread(2);
        break;
      }
    }
  }
}

int main()
{
  HANDLE  hMarker;
  DWORD  idMarker;

  // создаем событие, оповещающее о тупике
  hDeadlock = CreateEvent(NULL, FALSE, FALSE, NULL);
  // создаем события для обработки тупика
  hAnswer[0] = CreateEvent(NULL, FALSE, FALSE, NULL);
  hAnswer[1] = CreateEvent(NULL, FALSE, FALSE, NULL);
  // запускаем поток marker
  hMarker = CreateThread(NULL, 0, marker,NULL, 0, &idMarker);
  if (hMarker == NULL)
    return GetLastError();
  for (;;)
  {
```

```
    char c;
    // ждем сигнал о тупике
    WaitForSingleObject(hDeadlock, INFINITE);
    // выводим на консоль текущее состояние массива
    cout << "Current state of the array: ";
    for (int i = 0; i < size; ++i)
      cout << a[i] << ' ';
    cout << endl;
    // завершать или нет поток marker?
    cout << "Input 'y' to continue: ";
    cin >> c;
    if (c == 'y')
      SetEvent(hAnswer[0]);  // продолжаем работу
    else
    {
      SetEvent(hAnswer[1]);  // завершаем работу
      break;
    }
  }

  WaitForSingleObject(hMarker, INFINITE);
  CloseHandle(hMarker);

  return 0;
}
```

Обратим внимание на обработку в этой программе ситуации, когда возникает тупик. В этом случае мы используем для обработки тупика два события, которые сигнализируют потоку `marker` о завершении или продолжении работы. Второй момент касается заполнения элементов массива целыми числами. В нашем случае у нас фактически работает только один поток `marker`, а поток `main` просто ждет до тех пор, пока поток `marker` войдет в тупик. Поэтому для проверки значений элементов массива `a[size]` поток `marker` просто выполняет сравнения значений элементов этого массива с нулем. Если бы параллельно работало несколько потоков `marker`, то для выполнения этих действий пришлось бы использовать функцию `InterlockedCompareExchange`.

Часть III

Программирование консольных приложений

Глава 9

Структура консольного приложения

9.1. Структура консоли

Консолью называется интерфейс, который используется приложением для ввода-вывода текстовой информации. В этом случае приложение, которое использует консоль для обмена данными с пользователем, называется *консольным приложением*. Консольные приложения применяются главным образом в системном программировании для разработки различных сервисов и для обработки неустранимых ошибок, возникающих при работе графического приложения.

Консоль состоит из одного входного буфера и одного или нескольких буферов экрана. *Входной буфер* содержит информацию о событиях ввода. Каждое событие ввода описывается записью. Все записи упорядочены в очередь, которая хранится в буфере ввода. *Буфер экрана* содержит информацию для вывода в окно приложения и является двумерным массивом, который содержит символы и данные о цвете.

Консоль обеспечивает два уровня ввода-вывода текстовой информации: высокий и низкий. Функции высокого уровня обеспечивают ввод-вывод символов с консоли, игнорируя остальные события. Функции низкого уровня обеспечивают обработку всех событий, связанных с консольным приложением.

9.2. Входной буфер консоли

Входной буфер консоли содержит очередь записей, которые описывают события ввода. События ввода подразделяются на следующие категории:

- ввод с клавиатуры;
- ввод с мыши;
- изменение размеров окна;

- изменение фокуса ввода;
- события, связанные с меню.

Два последних события ввода, связанные с фокусом ввода и меню, обрабатываются системой и должны игнорироваться приложением. Каждому событию ввода в очереди сообщений соответствует запись типа `INPUT_RECORD`, которая имеет следующую структуру:

```
typedef  struct  _INPUT_RECORD {
  WORD EventType;
  union {
    KEY_EVENT_RECORD      KeyEvent;
    MOUSE_EVENT_RECORD    MouseEvent;
    WINDOW_BUFFER_SIZE_RECORD  WindowBufferSizeEvent;
    MENU_EVENT_RECORD     MenuEvent;
    FOCUS_EVENT_RECORD    FocusEvent;
  }  Event;
}  INPUT_RECORD;
```

где поле `EventType` определяет тип события ввода и может принимать одно из следующих значений:

- `KEY_EVENT` — ввод с клавиатуры;
- `MOUSE_EVENT` — ввод с мыши;
- `WINDOW_BUFFER_SIZE_EVENT` — изменение размеров окна;
- `MENU_EVENT` — событие, связанное с меню;
- `FOCUS_EVENT` — изменение фокуса ввода.

В свою очередь объединение `Event` содержит запись только одного из перечисленных типов в зависимости от значения поля `EventType`.

Рассмотрим структуру записи для каждого события ввода.

Запись для ввода с клавиатуры имеет следующую структуру:

```
typedef  struct  _KEY_EVENT_RECORD {
  BOOL    bKeyDown;
  WORD    wRepeatCount;
  WORD    wVirtualKeyCode;
  WORD    wVirtualScanCode;
  union {
    WCHAR  UnicodeChar;
    CHAR   AsciiChar;
  }  uChar;
```

```
    DWORD  dwControlKeyState;
} KEY_EVENT_RECORD;
```

Кратко опишем поля этой структуры:

- значение поля `bKeyDown` равно `TRUE`, если клавиша нажата, в противном случае значение этого поля равно `FALSE`;
- поле `wRepeatCount` содержит счетчик задержки при нажатии клавиши;
- поле `wVirtualKeyCode` содержит код клавиши, который не зависит от типа клавиатуры;
- поле `wVirtualScanCode` содержит код клавиши, который генерируется клавиатурой. Значение этого поля зависит от клавиатуры, т. к. клавиатуры разных производителей могут генерировать различные коды при нажатии одинаковых клавиш;
- объединение `uChar` содержит символ, который соответствует нажатой клавише. Этот символ может быть представлен как в коде Unicode, так и в коде ASCII;
- поле `dwControlKeyState` содержит комбинацию флагов, которые описывают состояние управляющих клавиш. Эти флаги могут принимать следующие значения:
  - `CAPSLOCK_ON` — включен индикатор <Caps Lock>;
  - `ENHANCED_KEY` — дополнительные клавиши;
  - `LEFT_ALT_PRESSED` — нажата левая клавиша <Alt>;
  - `LEFT_CTRL_PRESSED` — нажата левая клавиша <Ctrl>;
  - `NUMLOCK_ON` — включен индикатор <Num Lock>;
  - `RIGHT_ALT_PRESSED` — нажата правая клавиша <Alt>;
  - `RIGHT_CTRL_PRESSED` — нажата правая клавиша <Ctrl>;
  - `SCROLLLOCK_ON` — включен индикатор <Scoll Lock>;
  - `SHIFT_PRESSED` — нажата клавиша <Shift>.

Замечание

К дополнительным клавишам относится самый правый блок клавиш на клавиатуре, который обычно находится под индикаторами состояния клавиатуры. Кроме того, отметим, что автономное нажатие клавиши <Alt> имеет специальный смысл для системы и поэтому не передается приложению для обработки.

Теперь перейдем к описанию событий, связанных с мышью. Запись для ввода с мыши имеет следующую структуру:

```
typedef  struct  _MOUS_EVENT_RECORD {
  COORD  dwMousePosition;
```

```
    DWORD  dwButtonState;
    DWORD  dwControlKeyState;
    DWORD  dwEventFlags;
  }  MOUSE_EVENT_RECORD;
```

поля которой имеют следующее назначение:

- поле `dwMousePosition` определяет координаты курсора относительно буфера экрана;
- поле `dwButtonState` содержит флаги, которые определяют состояние кнопок мыши. Эти флаги описываются следующими символическими константами в порядке их следования по битам, начиная с младшего бита:
  - `FROM_LEFT_1ST_BUTTON_PRESSED` — нажата самая левая кнопка мыши;
  - `RIGHTMOST_BUTTON_PRESSED` — нажата самая правая кнопка мыши;
  - `FROM_LEFT_2ND_BUTTON_PRESSED` — нажата вторая слева кнопка мыши;
  - `FROM_LEFT_3RD_BUTTON_PRESSED` — нажата третья слева кнопка мыши;
  - `FROM_LEFT_4TH_BUTTON_PRESSED` — нажата четвертая слева кнопка мыши;
- поле `dwControlKeyState` определяет состояние управляющих клавиш и может принимать те же значения, что и соответствующее поле в записи `KEY_EVENT_RECORD`;
- поле `dwEventFlags` отмечает тип события и может принимать одно из следующих значений:
  - `0` — кнопка мыши нажата или отпущена;
  - `DOUBLE_CLICK` — кнопка мыши нажата второй раз, первое нажатие отмечается `0`;
  - `MOUSE_MOVED` — изменение позиции мыши;
  - `MOUSE_WHEELED` — крутится колесо мыши.

Замечание

Значение `MOUSE_WHEELED` используется, начиная только с версии Windows 2000.

В заключение этого раздела рассмотрим события, которые связаны с изменением размеров окна. Записи, описывающие такие события, имеют следующую структуру:

```
typedef  struct  _WINDOW_BUFFER_SIZE_RECORD {
  COODR    dwSize;
}  WINDOW_BUFFER_SIZE_RECORD;
```

где поле `dwSize` определяет новый размер буфера экрана в символах.

9.3. Буфер экрана

Буфер экрана является двумерным массивом, элементы которого представляют собой записи типа:

```
typedef struct _CHAR_INFO {
  union {
    WCHAR UnicodeChar;
    CHAR  AsciiChar;
  } Char;
  WORD   Attributes;
} CHAR_INFO, *PCHAR_INFO;
```

где объединение `Char` содержит символ, представленный в коде Unicode или ASCII, а поле `Attributes` определяет цвет фона и цвет текста, которыми выводятся символы на экран дисплея. Это значение может быть равно `0`, что обозначает фон — черный, а цвет — белый, или любой комбинации из следующих констант:

- `BACKGROUND_BLUE` — фон синий;
- `BACKGROUND_GREEN` — фон зеленый;
- `BACKGROUND_RED` — фон красный;
- `BACKGROUND_INTENSITY` — фон яркий;
- `FOREGROUND_BLUE` — текст синий;
- `FOREGROUND_GREEN` — текст зеленый;
- `FOREGROUND_RED` — текст красный;
- `FOREGROUND_INTENSITY` — текст яркий.

Цвет фона и цвет текста будем называть атрибутами текста. Сделаем несколько замечаний относительно использования этих констант. Цвет фона и цвет текста выбираются как комбинация базовых цветов синего, зеленого и красного. То есть в этом случае используется цветовая модель RGB. Можно подсчитать, что всего существует семь возможных комбинаций из трех цветов. Белый цвет определяется комбинацией всех трех цветов. Если сюда добавить черный цвет, который определяется как побитовое отрицание белого цвета, то всего существует восемь возможных вариантов, как для цвета фона, так и для цвета текста.

Теперь обсудим использование констант `BACKGROUND_INTENSITY` и `FOREGROUND_INTENSITY`, задающих яркость. Обычно если окно консольного приложения имеет фокус ввода, то фон и текст выводятся на экран яркими цветами, что и задается этими константами. Если же окно не имеет фокус ввода, то его фон и текст выводятся на экран приглушенными цветами.

Глава 10

Работа с консолью

10.1. Создание консоли

Процесс может быть связан только с одной консолью. Новая консоль может создаваться одним из следующих двух способов.

Первый способ заключается в том, что при создании консольного процесса командой `CreateProcess` нужно установить флаг `CREATE_NEW_CONSOLE`. Отметим, что в этом случае, если консольный процесс создается из консольного приложения, а указанный флаг не установлен, то новый процесс присоединяется к консоли родительского процесса. Более подробно функция `CreateProcess` рассмотрена в *разд. 4.2*.

Второй способ заключается в использовании функции `AllocConsole`, которая имеет следующий прототип:

```
BOOL  AllocConsole(VOID);
```

Эта функция возвращает ненулевое значение, если консоль создана успешно, и `FALSE` — в противном случае.

В обоих этих случаях заголовок окна консоли, его параметры, а также цвет фона и цвет текста задаются следующими полями структуры `STARTUPINFO`:

- `lpTitle` — заголовок окна консоли;
- `dwX`, `dwY` — позиция левого угла окна консоли;
- `dwXSize`, `dwYSize` — размеры окна;
- `dwXCountChars`, `dwYCountChars` — размеры буфера экрана;
- `dwFillAttributes` — цвет фона и цвет текста;
- `wShowWindow` — способ отображения окна при запуске приложения.

Все эти параметры используются при запуске процесса только в случае, если в поле `dwFlags` этой же структуры установлены следующие управляющие флаги:

- `STARTF_USEPOSITION` — использовать поля `dwX`, `dwY`;

- `STARTF_USESIZE` — использовать поля `dwXSize`, `dwYSize`;
- `STARTF_USECOUNTCHARS` — использовать поля `dwXCountChars`, `dwYCountChars`;
- `STARTF_USEFILLATTRIBUTE` — использовать поле `dwFillAttributes`;
- `STARTF_USESHOWWINDOW` — использовать поле `wShowWindow`.

В противном случае значения этих полей игнорируются.

Отметим, что в операционной системе Windows 98 при создании новой консоли используется только значение поля `lpTitle`, а остальные атрибуты консоли используются по умолчанию (например, черный цвет фона и белый цвет текста).

В листингах 10.1, 10 2 приведены тексты программ, которые создают новую консоль.

Листинг 10.1. Установка атрибутов консоли для дочернего консольного процесса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char lpszAppName1[] = "C:\\ConsoleProcess.exe";
  STARTUPINFO si;
  PROCESS_INFORMATION pi;

  // заполняем поля структуры STARTUPINFO
  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb = sizeof(STARTUPINFO);
  si.lpTitle = "This is a new title";
  si.dwX = 200;            // позиция левого угла окна консоли
  si.dwY = 200;
  si.dwXSize = 300;        // размеры окна
  si.dwYSize = 200;
  si.dwXCountChars = 100;  // размер буфера экрана по горизонтали
  si.dwYCountChars = 100;  // размер буфера экрана по вертикали
  // красные буквы на синем фоне
  si.dwFillAttribute = FOREGROUND_RED|FOREGROUND_INTENSITY|
    BACKGROUND_INTENSITY|BACKGROUND_BLUE;
  // используем все параметры, что установили
  si.dwFlags = STARTF_USECOUNTCHARS|STARTF_USEFILLATTRIBUTE|
```

```
    STARTF_USEPOSITION|STARTF_USESHOWWINDOW|
    STARTF_USESIZE;
  si.wShowWindow = SW_SHOWNORMAL;

  // запускаем процесс с новой цветной консолью
  if (!CreateProcess(lpszAppName1, NULL, NULL, NULL, FALSE,
      CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi))
  {
    cout << "The new process is not created." << endl
      << "Check a name of the process." << endl
      << "Press any key to finish." << endl;
    cin.get();

    return 0;
  }
  // закрываем дескрипторы процесса
  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);

  return 0;
}
```

Листинг 10.2. Дочерний консольный процесс

```
#include <iostream.h>

int main()
{
  cout << "I am created." << endl;
  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
```

Теперь рассмотрим программу, которая создает консоль, используя функцию `AllocConsole`. Параметры этой консоли берутся из структуры `STARTUPINFO`, которую заполняет процесс, запускающий эту программу.

В листингах 10.3, 10.4 приведены программы, которые демонстрируют создание новой консоли при помощи функции `AllocConsole`.

Листинг 10.3. Создание нового процесса без консоли

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char lpszAppName[] = "C:\\ConsoleProcess.exe";

  STARTUPINFO si;
  PROCESS_INFORMATION pi;

  // заполняем поля структуры STARTUPINFO
  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb = sizeof(STARTUPINFO);
  si.lpTitle = "This is a new title";
  si.dwX = 200;            // позиция левого угла окна консоли
  si.dwY = 200;
  si.dwXSize = 200;        // размеры окна
  si.dwYSize = 100;
  si.dwXCountChars = 150;  // размер буфера экрана по горизонтали
  si.dwYCountChars = 50;   // размер буфера экрана по вертикали
  // красные буквы на синем фоне
  si.dwFillAttribute = FOREGROUND_RED|FOREGROUND_INTENSITY|
  BACKGROUND_BLUE|BACKGROUND_INTENSITY;
  // используем все параметры, что установили
  si.dwFlags = STARTF_USECOUNTCHARS|STARTF_USEFILLATTRIBUTE|
  STARTF_USEPOSITION|STARTF_USESIZE;
  si.wShowWindow = SW_SHOWNORMAL;

  // запускаем процесс, который сам распределяет консоль
  if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
    DETACHED_PROCESS, NULL, NULL, &si, &pi))
  {
```

```
    cout << "The new process is not created." << endl
      << "Check a name of the process." << endl
      << "Press any key to finish." << endl;
    cin.get();

    return 0;
  }
  // закрываем дескрипторы процесса
  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);

  return 0;
}
```

Листинг 10.4. Распределение консоли (`AllocateConsole`) дочерним процессом

```
#include <windows.h>
#include <conio.h>

int main()
{
  if (!AllocConsole())
  {
    MessageBox(NULL,
      "Console allocation failed",
      "Ошибка Win32 API",
      MB_OK | MB_ICONINFORMATION
    );
    return 0;
  }

  _cputs("I am created.");
  _cputs("\nPress any char to exit.\n");
  _getch();

  return 0;
}
```

10.2. Освобождение консоли

Приложение освобождает консоль посредством вызова функции `FreeConsole`, которая имеет следующий прототип:

```
BOOL  FreeConsole(VOID);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в противном случае — `FALSE`.

В листинге 10.5 приведена программа процесса, который запускается без консоли. В начале своей работы этот процесс распределяет консоль и затем выводит несколько сообщений. После этого процесс освобождает консоль посредством вызова функции `FreeConsole`.

Листинг 10.5. Освобождение консоли функцией `FreeConsole`

```
#include <windows.h>
#include <conio.h>

int main()
{
  // распределяем консоль
  if (!AllocConsole())
  {
    MessageBox(NULL,
      "Console allocation failed", "Ошибка Win32 API"
      MB_OK | MB_ICONINFORMATION
    );
    return 0;
  }

  _cputs("I am created.\n");
  _cputs("Press any char to exit.\n");
  _getch();

  // освобождаем консоль
  if (!FreeConsole())
  {
    _cputs("Free console failed.\n");
```

```
    _cputs("Press any key to exit.\n");
    _getch();
  }

  return 0;
}
```

10.3. Стандартные дескрипторы ввода-вывода

При создании новой консоли система создает три дескриптора, которые обозначаются `STDIN`, `STDOUT`, `STDERR` и называются соответственно стандартными дескрипторами ввода, вывода и ошибки. Дескриптор `STDIN` связывается с буфером ввода, а дескрипторы `STDOUT` и `STDERR` связываются с буфером экрана. Эти дескрипторы используются в функциях, предназначенных для работы с консолью.

Значения стандартных дескрипторов ввода-вывода можно получить, используя функцию `GetStdHandle`, которая имеет следующий прототип:

```
HANDLE  GetStdHandle(DWORD  dwStdHandle);
```

где параметр `dwStdHandle` может принимать одно из следующих значений:

- `STD_INPUT_HANDLE` — дескриптор стандартного ввода;
- `STD_OUTPUT_HANDLE` — дескриптор стандартного вывода;
- `STD_ERROR_HANDLE` — дескриптор стандартной ошибки.

При успешном завершении функция `GetStdHandle` возвращает требуемый дескриптор, а в случае неудачи — значение `INVALID_HANDLE_VALUE`.

Установить значения стандартных дескрипторов ввода-вывода можно при помощи функции `SetStdHandle`, которая имеет следующий прототип:

```
BOOL  SetStdHandle(
  DWORD   dwStdHandle,     // тип дескриптора
  HANDLE  hHandle          // новое значение дескриптора
);
```

где параметр `dwStdHandle` принимает те же значения, что и в функции `GetStdHandle`, а параметр `hHandle` равен новому значению для стандартного дескриптора. При успешном завершении функция `SetStdHandle` возвращает ненулевое значение, а в случае неудачи — `FALSE`.

Обычно функция `SetStdHandle` используется для перенаправления потоков стандартного ввода-вывода. В этом случае дескрипторы стандартных пото-

ков ввода-вывода могут быть определены вызовом функции `CreateFile`, которая в качестве имени файла (параметр `lpNameFile`) принимает одно из значений: `CONIN$` или `CONOUT$`. Остальные параметры устанавливаются следующим образом.

Для входного потока `CONIN$`:

- `dwDesiredAccess` — устанавливается в `GENERIC_READ`;
- `dwShareMode` — устанавливается в `FILE_SHARE_READ`, если консоль наследуется;
- `dwCreationDisposition` — устанавливается в `OPEN_EXISTING`.

Для выходного потока `CONOUT$`:

- `dwDesiredAccess` — устанавливается в `GENERIC_WRITE`;
- `dwShareMode` — устанавливается в `FILE_SHARE_WRITE`, если консоль наследуется;
- `dwCreationDisposition` — устанавливается в `OPEN_EXISTING`.

В обоих случаях параметры `dwFlagsAndAttributes` и `hTemplateFile` игнорируются, а параметр `lpSecurityAttributes` используется только для того, чтобы сделать консоль наследуемой.

Более подробно перенаправление стандартного ввода/вывода будет рассмотрено в *гл. 13*, посвященной передаче данных между процессами при помощи анонимных каналов.

Глава 11

Работа с окном консоли

11.1. Получение дескриптора окна консоли

Дескриптор окна консоли можно получить, вызвав функцию `GetConsoleWindow`, которая имеет следующий прототип:

```
HWND  GetConsoleWindow(VOID);
```

В случае успеха эта функция возвращает дескриптор окна консоли, а в случае неудачи — `NULL`. Неудача означает, что консоли у приложения нет. Отметим, что функция `GetConsoleWindow` работает только на платформе операционных систем Windows 2000/XP.

В листинге 11.1 приведена программа, которая получает дескриптор окна консоли, используя для этого функцию `GetConsoleWindow`.

Листинг 11.1. Получение дескриптора окна консоли

```
#include <windows.h>
#include <stdio.h>

extern "C" WINBASEAPI HWND WINAPI GetConsoleWindow ();

int main()
{
  HWND hWindow = NULL;      // дескриптор окна
  HDC hDeviceContext;       // контекст устройства
  HPEN hPen;                // дескриптор пера
  HGDIOBJ hObject;          // дескриптор GDI объекта

  // получаем дескриптор окна
```

```
  hWindow = GetConsoleWindow();

  if (hWindow == NULL)
  {
    printf("Get console window failed.\n");

    return 1;
  }
  else
    printf("Cet console window is done.\n");

  // получаем контекст устройства
  hDeviceContext = GetDC(hWindow);
  // создаем перо
  hPen = CreatePen(PS_SOLID, 10, RGB(0, 255, 0));
  // устанавливает перо
  hObject = SelectObject(hDeviceContext, hPen);

  // рисуем линию
  MoveToEx(hDeviceContext, 100, 100, NULL);
  LineTo(hDeviceContext, 500, 100);

  // восстанавливает старый объект
  SelectObject(hDeviceContext, hObject);

  // освобождаем объекты
  DeleteObject(hPen);
  ReleaseDC(hWindow, hDeviceContext);

  return 0;
}
```

11.2. Получение и изменение заголовка консоли

Для чтения заголовка окна консоли используется функция `GetConsoleTitle`, которая имеет следующий прототип:

```
  DWORD  GetConsoleTitle(
    LPTSTR  lpConsoleTitle,  // адрес буфера для заголовка
```

```
    DWORD   nSize            // размер буфера для заголовка в символах
  );
```

В случае успеха эта функция возвращает длину строки с заголовком в символах, а в случае неудачи — 0.

Для установки заголовка окна консоли используется функция `SetConsoleTitle`, которая имеет следующий прототип:

```
  BOOL  SetConsoleTitle(
    LPCTSTR  lpConsoleTitle  // указатель на строку с заголовком
  );
```

В случае успеха эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`.

В листинге 11.2 приведена программа, которая сначала читает заголовок окна консоли, используя функцию `GetConsoleTitle`, а затем изменяет его, используя функцию `SetConsoleTitle`.

Листинг 11.2. Чтение и изменение заголовка окна консоли

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char  ConsoleTitleBuffer[80];  // указатель на буфер с заголовком
  DWORD  dwBufferSize = 80;      // размер буфера для заголовка
  DWORD  dwTitleSize;            // длина заголовка

  // читаем заголовок консоли
  dwTitleSize = GetConsoleTitle(ConsoleTitleBuffer, dwBufferSize);

  // выводим на консоль результат
  cout << "Title length = " << dwTitleSize << endl;
  cout << "The window title = " << ConsoleTitleBuffer << endl;

  cout << "Input new title: ";
  cin.getline(ConsoleTitleBuffer, 80);
  // устанавливаем новый заголовок консоли
  if (!SetConsoleTitle(ConsoleTitleBuffer))
```

```
    cout << "Set console title failed." << endl;

  cout << "The title was changed." << endl;
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

11.3. Определение максимального размера окна

Максимальный размер окна консоли можно определить, вызвав функцию `GetLargestConsoleWindowSize`, которая имеет следующий прототип:

```
COORD  GetLargestConsoleWindowSize(
  HANDLE  hConsoleOutput     // дескриптор буфера экрана
);
```

В случае успеха эта функция возвращает в структуру следующего типа:

```
typedef  struct  _COORD {
  SHORT    X,
  SHORT    Y,
}  COORD;
```

Поле `X` содержит максимальное количество столбцов, а поле `Y` — максимальное количество строк окна консоли в символах. Следует отметить, что `X` и `Y` зависят от текущего размера шрифта и величины экрана. В случае неудачи функция `GetLargestConsoleWindowSize` возвращает в полях структуры `COORD` нули.

В листинге 13.3 приведена программа, которая, используя функцию `GetLargestConsoleWindowSize`, определяет максимальные размеры окна консоли.

Листинг 11.3. Определение максимальных размеров окна консоли

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hConsoleOutput;  // дескриптор буфера экрана
```

```
  COORD  coord;            // координаты окна консоли

  // получаем дескриптор окна консоли
  hConsoleOutput = GetStdHandle(STD_OUTPUT_HANDLE);
  if (hConsoleOutput == INVALID_HANDLE_VALUE)
  {
    cout << "Get standard handle failed." << endl;
    return GetLastError();
  }
  // получаем максимальные размеры окна консоли
  coord = GetLargestConsoleWindowSize(hConsoleOutput);
  if (coord.X == 0 && coord.Y == 0)
  {
    cout << "Get largest console window size failed." << endl;
    return GetLastError();
  }

  cout << "Coordinate X = " << coord.X << endl;
  cout << "Coordinate Y = " << coord.Y << endl;

  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

11.4. Установка координат окна

Установить окно консоли относительно буфера экрана можно вызовом функции `SetConsoleWindowInfo`, которая имеет следующий прототип:

```
BOOL  SetConsoleWindowInfo(
  HANDLE  hConsoleOutput,    // дескриптор буфера экрана
  BOOL    bAbsolut,          // тип координат
  CONST  SMALL_RECT  *lpConsoleWindow  // углы окна
);
```

Параметры этой функции могут принимать следующие значения:

- параметр `hConsoleOutput` должен содержать дескриптор буфера экрана, который должен быть открыт в режиме `GENERIC_WRITE` (*см. гл. 12*);

- в параметре `bAbsolute` может быть установлено значение `FALSE` или `TRUE`. Если значение этого параметра равно `FALSE`, то поля структуры `SMALL_RECT` определяют сдвиг буфера экрана относительно левого верхнего и правого нижнего углов текущего окна консоли. Если же значение параметра `bAbsolute` равно `TRUE`, то поля структуры `SMALL_RECT` задают абсолютное положение левого верхнего и правого нижнего углов окна консоли относительно буфера экрана;
- для задания левого верхнего и правого нижнего углов окна используются поля структуры `SMALL_RECT`, которая имеет следующий тип:

```
typedef  struct  _SMALL_RECT {
  SHORT  Left;     // x - координата левого верхнего угла окна консоли
  SHORT  Top;      // y - координата левого верхнего угла окна консоли
  SHORT  Right;    // x - координата правого нижнего угла окна консоли
  SHORT  Bottom;   // y - координата правого нижнего угла окна консоли
} SMALL_RECT;
```

Координаты в структуре `SMALL_RECT` определяют ячейки буфера экрана.

В случае успеха функция `SetConsoleWindowInfo` возвращает ненулевое значение, а в противном случае — `FALSE`. Неудачное завершение функции может быть вызвано неправильным заданием координат окна консоли. Например, координаты окна консоли превышают размер буфера экрана, или не выполняется условие `Left > 0`, или не выполняется условие `Left < Right` и т. д.

Отметим, что функция `SetConsoleWindowInfo` используется для прокрутки буфера экрана в окне консоли в операционной системе Windows 98.

В листинге 11.4 приведена программа, в которой устанавливаются абсолютные размеры окна консоли относительно буфера экрана.

Листинг 11.4. Установка абсолютных размеров окна консоли

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hStdOut;   // дескриптор стандартного вывода
  SMALL_RECT  sr;    // прямоугольник окна

  // читаем дескриптор стандартного вывода
  hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);

  cout << "Set new window rectangle in characters." << endl << endl;
```

```
  // устанавливаем прямоугольник окна в символах
  cout << "Input left coordinate (0-79): ";
  cin >> sr.Left;
  cout << "Input top coordinate (0-24): ";
  cin >> sr.Top;
  cout << "Input right coordinate (0-79): ";
  cin >> sr.Right;
  cout << "Input bottom coordinate (0-24): ";
  cin >> sr.Bottom;
  // устанавливаем новое окно
  if (!SetConsoleWindowInfo(hStdOut, TRUE, &sr))
    cout << "Set console window info failed." << endl;

  cin.get();
  cout << endl << "Press any key to exit.";
  cin.get();

  return 0;
}
```

Теперь, в листинге 11.5, приведем программу, в которой устанавливаются относительные размеры окна. Отметим, что в этой программе числовые пределы координат окна установлены для работы в операционной системе Windows 2000. В операционной системе Windows 98 координаты правого нижнего угла могут принимать только отрицательные значения.

Листинг 11.5. Установка относительных размеров окна консоли

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hStdOut;   // дескриптор стандартного вывода
  SMALL_RECT  sr;    // прямоугольник окна

  // читаем дескриптор стандартного вывода
  hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);

  cout << "Set new window rectangle in characters."<<endl<<endl;
```

```
  // устанавливаем прямоугольник окна в символах
  cout << "Input left coordinate (0-79): ";
  cin >> sr.Left;
  cout << "Input top coordinate (0-24): ";
  cin >> sr.Top;
  cout << "Input right coordinate (" << (sr.Left - 79) << ','
       << sr.Left << "): ";
  cin >> sr.Right;
  cout << "Input bottom coordinate (" << (sr.Top - 24) << ','
    << sr.Top << "): ";
  cin >> sr.Bottom;
  // устанавливаем новое окно в относительных координатах
  if (!SetConsoleWindowInfo(hStdOut, FALSE, &sr))
    cout << "Set console window info failed." << endl;

  cin.get();
  cout << endl << "Press key to exit.";
  cin.get();

  return 0;
}
```

Глава 12

Работа с буфером экрана

12.1. Создание и активация буфера экрана

Буфер экрана может быть создан посредством вызова функции `CreateConsoleScreenBuffer`, которая имеет следующий прототип:

```
HANDLE  CreateConsoleScreenBuffer(
  DWORD  dwDesiredAccess,     // режимы доступа
  DWORD  dwShareMode,         // режимы разделения доступа
  CONST  SECURITY_ATTRIBUTES  *lpSecurityAttributes, // атрибуты защиты
  DWORD  dwFlags,             // тип буфера экрана
  LPVOID lpScreenBufferData   // зарезервировано
);
```

В случае успешного завершения эта функция возвращает дескриптор нового буфера экрана, а в случае неудачи значение `INVALID_HANDLE_VALUE`. Параметры этой функции имеют следующее назначение.

В параметре `dwDesiredAccess` задаются флаги, которые определяет способ доступа к буферу экрана. Этот параметр может принимать любую комбинацию следующих значений:

- `GENERIC_READ` — процессу разрешается чтение данных из буфера экрана;
- `GENERIC_WRITE` — процессу разрешается запись в буфер экрана.

Параметр `dwShareMode` определяет, может ли буфер экрана использоваться несколькими процессами одновременно. Если этот параметр равен 0, то создаваемый буфер экрана не может использоваться несколькими процессами одновременно. Кроме того, этот параметр может принимать любую комбинацию следующих значений:

- `FILE_SHARE_READ` — буфер экрана допускает совместное чтение;
- `FILE_SHARE_WRITE` — буфер экрана допускает совместную запись.

Параметр `lpSecurityAttributes` задает атрибуты защиты. Пока он будет устанавливаться нами в `NULL`, что задает атрибуты защиты по умолчанию.

Параметр `dwFlags` задает тип буфера экрана и может принимать только одно значение `CONSOLE_TEXTMODE_BUFFER`.

Параметр `lpScreenBufferData` зарезервирован для дальнейшего использования и поэтому должен быть установлен в `NULL`.

После создания буфер экрана содержит пробелы, а курсор установлен в позицию с координатами (0, 0). Чтобы сделать буфер экрана активным, т. е. выводить его содержимое на экран, нужно вызвать функцию `SetConsoleActiveScreenBuffer`, которая имеет следующий прототип:

```
BOOL  SetConsoleActiveScreenBuffer(HANDLE  hConsoleOutput);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Единственным параметром этой функции является дескриптор буфера экрана.

В листинге 12.1 приведена программа, которая сначала создает второй буфер экрана, а затем этот буфер экрана делает активным. Отметим, что в этой программе используется функция `WriteConsole` для вывода в новый буфер экрана. Более подробно работа этой функции будет рассмотрена в *разд. 13.1*, посвященном выводу на высоком уровне.

Листинг 12.1. Создание и активация буфера экрана

```
#include <windows.h>
#include <conio.h>

int main()
{
  HANDLE  hStdOutOld, hStdOutNew;  // дескрипторы буфера экрана
  DWORD  dwWritten;                // для количества выведенных символов

  // создаем буфер экрана
  hStdOutNew = CreateConsoleScreenBuffer(
    GENERIC_READ | GENERIC_WRITE,  // чтение и запись
    0,          // не разделяемый
    NULL,       // защита по умолчанию
    CONSOLE_TEXTMODE_BUFFER,       / текстовый режим
    NULL);      // не используется

  if (hStdOutNew == INVALID_HANDLE_VALUE)
  {
```

```
  _cputs("Create console screen buffer failed.\n");
  return GetLastError();
}
// сохраняем старый буфер экрана
hStdOutOld = GetStdHandle(STD_OUTPUT_HANDLE);
  // ждем команду на переход к новому буферу экрана
_cputs("Press any key to set new screen buffer active.\n");
_getch();

// делаем активным новый буфер экрана
if (!SetConsoleActiveScreenBuffer(hStdOutNew))
{
  _cputs("Set new console active screen buffer failed.\n");
  return GetLastError();
}

// выводим текст в новый буфер экрана
char  text[] = "This is a new screen buffer.";
if (!WriteConsole(
    hStdOutNew,    // дескриптор буфера экрана
    text,          // символы, которые выводим
    sizeof(text),  // длина текста
    &dwWritten,    // количество выведенных символов
    NULL))         // не используется
  _cputs("Write console output character failed.\n");

// выводим сообщение о вводе символа
char  str[] = "\nPress any key to set old screen buffer.";
if (!WriteConsole(
    hStdOutNew,    // дескриптор буфера экрана
    str,           // символы, которые выводим
    sizeof(str),   // длина текста
    &dwWritten,    // количество выведенных символов
    NULL))         // не используется
  _cputs("Write console output character failed.\n");

_getch();

// восстанавливаем старый буфер экрана
```

```
  if (!SetConsoleActiveScreenBuffer(hStdOutOld))
  {
    _cputs("Set old console active screen buffer failed.\n");
    return GetLastError();
  }
  // пишем в старый буфер экрана
  _cputs("This is an old console screen buffer.\n");

  // закрываем новый буфер экрана
  CloseHandle(hStdOutNew);
  // ждем команду на завершение программы
  _cputs("Press any key to finish.\n");
  _getch();

  return 0;
}
```

12.2. Определение и установка параметров буфера экрана

Параметры буфера экрана можно определить с помощью функции `GetConsoleScreenBufferInfo`, которая имеет следующий прототип:

```
BOOL  GetConsoleScreenBufferInfo(
  HANDLE  hConsoleOutput,    // дескриптор буфера экрана
  // указатель на строку параметров буфера экрана
  PCONSOLE_SCREEN_BUFFER_INFO  lpConsoleScreenBufferInfo
);
```

В случае успеха эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. При успешном завершении функция `GetConsoleScreenBufferInfo` возвращает в параметре `lpConsoleScreenBufferInfo` указатель на структуру следующего типа:

```
typedef  struct  _CONSOLE_SCREEN_BUFFER_INFO {
  COORD  dwSize;   // размер буфера экрана в символах (столбцы, строки)
  COORD  dwCursorPosition;     // координаты курсора в символах
  WORD   wAttributes;          // цвет фона и цвет текста
  SMALL_RECT;  SrWindow;       // левый верхний и правый нижний углы
                               // окна относительно буфера экрана
  COORD  dwMaximumWindowSize;  // максимальный размер окна
}  CONSOLE_SCREEN_BUFFER_INFO;
```

которая содержит параметры буфера экрана, заданного параметром `hConsoleOutput`.

В листинге 12.2 приведена программа, которая выводит на экран информацию о буфере экрана, используя функцию `GetConsoleScreenBufferInfo`.

Листинг 12.2. Вывод на экран информации о буфере экрана

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE    hStdOut;     // дескриптор стандартного вывода
  CONSOLE_SCREEN_BUFFER_INFO  csbi;  // для параметров буфера экрана

  // читаем стандартный дескриптор вывода
  hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
  // читаем параметры буфера экрана
  if (!GetConsoleScreenBufferInfo(hStdOut,&csbi))
    cout << "Console screen buffer info failed." << endl;

  cout << "Console screen buffer info: " << endl << endl;
  // выводим на консоль параметры выходного буфера
  cout << "A number of columns = " << csbi.dwSize.X << endl;
  cout << "A number of rows = " << csbi.dwSize.Y << endl;
  cout << "X cursor coordinate = " << csbi.dwCursorPosition.X << endl;
  cout << "Y cursor coordinate = " << csbi.dwCursorPosition.Y << endl;
  cout << "Attributes = " << hex << csbi.wAttributes << dec << endl;
  cout << "Window upper corner = "
       << csbi.srWindow.Left << ","
       << csbi.srWindow.Top << endl;
  cout << "Window lower corner = "
       << csbi.srWindow.Right << ","
       << csbi.srWindow.Bottom << endl;
  cout << "Maximum number of columns = "
       << csbi.dwMaximumWindowSize.X << endl;
  cout << "Maximum number of rows = "
       << csbi.dwMaximumWindowSize.Y << endl << endl;

  return 0;
}
```

Размер буфера экрана можно изменить, вызвав функцию `SetConsoleScreenBufferSize`, которая имеет следующий прототип:

```
BOOL  SetConsoleScreenBufferSize (
  HANDLE  hConsoleOutput,    // дескриптор буфера экрана
  COORD   dwSize             // новый размер буфера экрана
);
```

В случае успеха эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. Размер буфера экрана задается в символах и не может быть меньше размера окна консоли. Минимальные размеры буфера экрана также ограничены системой и зависят от размера используемого шрифта и системных метрик `SM_CXMIN` и `SM_CYMIN`.

В листинге 12.3 приведена программа, которая изменяет размер буфера экрана.

Листинг 12.3. Установка новых размеров буфера экрана

```
#include <windows.h>
#include <iostream.h>

int main()
{
  COORD  coord;      // для размера буфера экрана
  HANDLE  hStdOut;   // дескриптор стандартного вывода

  // читаем дескриптор стандартного вывода
  hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);

  // вводим новый размер буфера экрана
  cout << "Enter new screen buffer size." << endl;
  cout << "A number of columns: ";
  cin >> coord.X;
  cout << "A number of rows: ";
  cin >> coord.Y;
  // устанавливаем новый размер буфера экрана
  if (!SetConsoleScreenBufferSize(hStdOut, coord))
  {
    cout << "Set console screen buffer size failed." << endl;
    return GetLastError();
  }

  return 0;
}
```

12.3. Функции для работы с курсором

Информацию о положении и видимости курсора можно получить, используя функцию GetConsoleCursorInfo, которая имеет следующий прототип:

```
BOLL  GetConsoleCursorInfo(
  HANDLE  hConsoleOutput,    // дескриптор буфера экрана
  PCONSOLE_CURSOR_INFO  lpConsoleCursorInfo  // информация о курсоре
);
```

В случае успеха эта функция возвращает ненулевое значение, а в случае неудачи — FALSE. При успешном завершении функция GetConsoleCursorInfo возвращает в структуре типа CONSOLE_CURSOR_INFO, адрес которой должен быть задан в параметре lpConsoleCursorInfo, информацию о курсоре. Эта структура имеет следующий формат:

```
typedef  struct  _CONSOLE_CURSOR_INFO {
  DWORD  dwSize;
  BOOL   bVisible;
}  CONSOLE_CURSOR_INFO,  *PCONSOLE_CURSOR_INFO;
```

Поле dwSize изменяется в интервале от 1 до 100 и определяет размер курсора в процентах от размера клетки для символа, а поле bVisible определяет видимость курсора. Если значение поля bVisible равно TRUE, то курсор видим, в противном случае — невидим.

Размер и видимость курсора устанавливаются функцией SetConsoleCursorInfo, которая имеет следующий прототип:

```
BOOL  SetConsoleCursorInfo (
  HANDLE  hConsoleOutput,    // дескриптор буфера экрана
  // информация о курсоре
  CONST  CONSOLE_CURSOR_INFO  *lpConsoleCursorInfo
);
```

В случае успеха эта функция возвращает ненулевое значение, а в случае неудачи — значение FALSE. При этом нужные значения размера и видимости курсора передаются через параметр lpConsoleCursorInfo, который указывает на структуру типа CONSOLE_CURSOR_INFO, содержащую эти значения.

В листинге 12.4 приведена программа, которая использует функции GetConsoleCursorInfo и SetConsoleCursorInfo для чтения и установки параметров курсора.

Листинг 12.4. Чтение и установка параметров курсора

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char  c;
  HANDLE  hStdOut;           // дескриптор стандартного вывода
  CONSOLE_CURSOR_INFO cci;   // информация о курсоре

  // читаем дескриптор стандартного вывода
  hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
  // читаем информацию о курсоре
  if (!GetConsoleCursorInfo(hStdOut, &cci))
    cout << "Get console cursor info failed." << endl;
  // выводим информацию о курсоре
  cout << "Size of cursor in procents of char= " << cci.dwSize << endl;
  cout << "Visibility of cursor = " << cci.bVisible << endl;

  // читаем новый размер курсора
  cout << "Input a new size of cursor (1-100): ";
  cin >> cci.dwSize;
  // устанавливаем новый размер курсора
  if (!SetConsoleCursorInfo(hStdOut, &cci))
    cout << "Set console cursor info failed." << endl;

  cout << "Input any char to make the cursor invisible: ";
  cin >> c;
  // делаем курсор невидимым
  cci.bVisible = FALSE;
  // устанавливаем невидимый курсор
  if (!SetConsoleCursorInfo(hStdOut, &cci))
    cout << "Set console cursor info failed." << endl;

  cout << "Input any char to make the cursor visible: ";
  cin >> c;
  // делаем курсор невидимым
```

```
  cci.bVisible = TRUE;
  // устанавливаем видимый курсор
  if (!SetConsoleCursorInfo(hStdOut, &cci))
    cout << "Set console cursor info failed." << endl;

  return 0;
}
```

Позиция курсора в буфере экрана устанавливается функцией `SetConsoleCursorPosition`, которая имеет следующий прототип:

```
BOOL  SetConsoleCursorPosition(
  HANDLE  hConsoleOutput,    // дескриптор буфера экрана
  COORD   dwCursorPosition   // новая позиция курсора
);
```

В случае успеха эта функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Функция `SetConsoleCursorPosition` устанавливает курсор в позицию, заданную параметром `dwCursorPosition`.

В листинге 12.5 приведена программа, которая демонстрирует использование функции `SetConsoleCursorPosition`.

Листинг 12.5. Установка курсора в новую позицию

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char  c;
  HANDLE  hStdOut;   // дескриптор стандартного вывода
  COORD  coord;      // для позиции курсора

  // читаем дескриптор стандартного вывода
  hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);

  cout << "Input new cursor position." << endl;
  cout << "X = ";
  cin >> coord.X;
  cout << "Y = ";
```

```
  cin >> coord.Y;

  // установить курсор в новую позицию
  if (!SetConsoleCursorPosition(hStdOut, coord))
  {
    cout << "Set cursor position failed." << endl;
    return GetLastError();
  }

  cout << "This is a new position." << endl;
  cout << "Input any char to exit: ";
  cin >> c;

  return 0;
}
```

12.4. Чтение и установка атрибутов консоли

Для установки атрибутов консоли используется функция `FillConsoleOutputAttribute`, которая имеет следующий прототип:

```
BOOL  FillConsoleOutputAttribute(
  HANDLE  hConsoleOutput,      // дескриптор буфера экрана
  WORD    wAttributes,         // цвет фона и цвет текста
  DWORD   nLength,             // количество заполняемых клеток
  COORD   dwWriteCoord,        // координаты первой клетки
  LPDWORD lpNumberOfAttrsWritten  // количество заполненных клеток
);
```

В случае успешного завершения возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Функция `FillConsoleOutputAttribute` заполняет клетки экрана, количество которых задано параметром `nLength`, атрибутами, которые заданы параметром `wAttribute`. Параметр `dwWriteCoord` задает координаты первой заполняемой клетки, а параметр `lpNumberOfAttrsWritten` должен указывать на переменную типа `DWORD`, в которую функция `FillConsoleOutputAttribute` помещает количество заполненных клеток.

Фактически функция `FillConsoleOutputAttribute` используется для заполнения новыми атрибутами прямоугольных областей консоли. Программа из листинга 12.6 показывает, как заполнить всю консоль новыми атрибутами.

Листинг 12.6. Заполнение консоли новыми атрибутами

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char c;
  HANDLE  hStdOut;     // дескриптор стандартного вывода
  WORD    wAttribute;  // цвет фона и текста
  DWORD  dwLength;     // количество заполняемых клеток
  DWORD  dwWritten;    // для количества заполенных клеток
  COORD    coord;      // координаты первой клетки
  CONSOLE_SCREEN_BUFFER_INFO csbi;   // для параметров буфера экрана

  cout << "In order to fill console attributes, input any char: ";
  cin >> c;
  // читаем стандартный дескриптор вывода
  hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
  if (hStdOut == INVALID_HANDLE_VALUE)
  {
    cout << "Get standard handle failed." << endl;
    return GetLastError();
  }
  // читаем параметры выходного буфера
  if (!GetConsoleScreenBufferInfo(hStdOut, &csbi))
  {
    cout << "Console screen buffer info failed." << endl;
    return GetLastError();
  }

  // вычисляем размер буфера экрана в символах
  dwLength = csbi.dwSize.X * csbi.dwSize.Y;
    // начинаем заполнять буфер с первой клетки
  coord.X = 0;
  coord.Y = 0;
  // устанавливаем цвет фона голубым, а цвет символов желтым
  wAttribute = BACKGROUND_BLUE | BACKGROUND_INTENSITY |
```

```
    FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_INTENSITY;
  // заполняем буфер атрибутами
  if (!FillConsoleOutputAttribute(
    hStdOut,     // стандартный дескриптор вывода
    wAttribute,  // цвет фона и текста
    dwLength,    // длина буфера в символах
    coord,       // индекс первой клетки
    &dwWritten)) // количество заполненных клеток
  {
    cout << «Fill console output attribute failed.» << endl;
    return GetLastError();
  }

  cout << «The fill attributes was changed.» << endl;

  return 0;
}
```

Для того чтобы установить атрибуты символов, которые пишутся в буфер экрана функциями `WriteFile` и `WriteConsole`, а также отображаются на экране при чтении функциями `ReadFile` и `ReadConsole`, используется функция `SetConsoleTextAttributes`, которая имеет следующий прототип:

```
BOOL  SetConsoleTextAttributes (
  HANDLE  hConsoleOutput,    // дескриптор буфера экрана
  WORD    wAttribute         // цвет фона и цвет текста
);
```

В случае успеха эта функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Параметр `wAttribute` содержит новые цвета фона и текста, которыми будут выводиться символы.

Программа из листинга 12.7 показывает, как устанавливаются атрибуты текста консоли.

Листинг 12.7. Установка атрибутов текста консоли

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char c;
```

```
  HANDLE  hStdout;   // дескриптор стандартного вывода
  WORD  wAttribute;  // цвет фона и текста

  cout << "In order to set text attributes, input any char: ";
  cin >> c;

  // читаем стандартный дескриптор вывода
  hStdout = GetStdHandle(STD_OUTPUT_HANDLE);

  // задаем цвет фона зеленым, а цвет символов красным
  wAttribute = BACKGROUND_GREEN | BACKGROUND_INTENSITY |
    FOREGROUND_RED | FOREGROUND_INTENSITY;
  // устанавливаем новые атрибуты
  if (!SetConsoleTextAttribute(hStdout,wAttribute))
  {
    cout << "Set console text attribute failed." << endl;
    return GetLastError();
  }

  cout << "The text attributes was changed." << endl;

  return 0;
}
```

Для установки атрибутов в последовательные клетки буфера экрана используется функция `WriteConsoleOutputAttribute`, которая имеет следующий прототип:

```
BOOL  WriteConsoleOutputAttribute (
  HANDLE  hConsoleOutput,      // дескриптор буфера экрана
  CONST  WORD  *lpAttribute,   // указатель на атрибуты
  DWORD  nLength,              // количество заполняемых клеток
  COORD  dwWriteCoord,         // координаты первой клетки
  LPDWORD  lpNumberOfAttrsWritten  // количество заполненных клеток
);
```

При успешном завершении эта функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Все параметры этой функции, за исключением параметра `lpAttribute`, имеют тот же смысл, что и соответствующие параметры функции `SetConsoleTextAttributes`. Параметр `lpAttribute` указывает на массив атрибутов, которые будут заноситься в последовательные клетки буфера экрана.

В отличие от функции `SetConsoleTextAttributes` функция `WriteConsoleOutputAttribute` позволяет заполнять последовательные клетки буфера экрана не одним, а разными атрибутами.

Для чтения атрибутов текста буфера экрана из последовательных ячеек используется функция `ReadConsoleOutputAttribute`, которая имеет следующий прототип:

```
BOOL  ReadConsoleOutputAttribute(
  HANDLE  hConsoleOutput,      // дескриптор буфера экрана
  LPWORD  lpAttribute,         // указатель на атрибуты
  DWORD   nLength,             // количество читаемых клеток
  COORD   dwWriteCoord,        // координаты первой клетки
  LPDWORD lpNumberOfAttrsRead  // количество прочитанных клеток
);
```

При успешном завершении эта функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Назначение параметров `hConsoleOutput` и `dwWriteCoord` этой функции совпадает с назначением соответствующих параметров функции `WriteConsoleOutputAttribute`. Параметр `lpAttribute` указывает на область памяти, в которую будут читаться атрибуты. Параметр `nLength` должен содержать количество клеток, из которых будут читаться атрибуты, а параметр `lpNumberOfAttrsRead` — указывать на область памяти, в которую функция поместит количество прочитанных атрибутов.

В листинге 12.8 приведена программа, в которой используются функции `WriteConsoleOutputAttribute` и `ReadConsoleOutputAttribute` для установки и чтения атрибутов текста консоли из последовательных клеток.

Листинг 12.8. Установка и чтение атрибутов текста консоли

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hStdOut;              // дескриптор буфера экрана
  WORD    lpAttribute[4];       // массив клеток с атрибутами
  DWORD  nLength = 4;           // количество клеток
  COORD    dwCoord = {8, 0};    // координата первой клетки
  DWORD  NumberOfAttrs;         // количество обработанных клеток

  // читаем стандартный дескриптор вывода
  hStdOut=GetStdHandle(STD_OUTPUT_HANDLE);
```

```
  // выводим демо-текст
  cout << "Console text attributes." << endl;
  // ждем команды на изменение атрибутов слова "текст"
  cout << "Press any key to change attributes.";
  cin.get();
  // устанавливаем новые атрибуты
  lpAttribute[0] = BACKGROUND_BLUE | BACKGROUND_INTENSITY |
                   FOREGROUND_GREEN | FOREGROUND_INTENSITY;
  lpAttribute[1] = BACKGROUND_GREEN | BACKGROUND_INTENSITY |
                   FOREGROUND_BLUE | FOREGROUND_INTENSITY;
  lpAttribute[2] = BACKGROUND_RED | BACKGROUND_INTENSITY |
                   FOREGROUND_GREEN | FOREGROUND_INTENSITY;
  lpAttribute[3] = BACKGROUND_GREEN | BACKGROUND_INTENSITY |
                   FOREGROUND_RED | FOREGROUND_INTENSITY;
  // записываем новые атрибуты в буфер экрана
  if (!WriteConsoleOutputAttribute(hStdOut, lpAttribute,
      nLength, dwCoord, &NumberOfAttrs))
  {
    cout << "Read console output attribute failed." << endl;
    return GetLastError();
  }
  // читаем атрибуты слова "текст"
  if (!ReadConsoleOutputAttribute(hStdOut, lpAttribute,
      nLength, dwCoord, &NumberOfAttrs))
  {
    cout << "Read console output attribute failed." << endl;
    return GetLastError();
  }
  // распечатываем атрибуты слова "текст"
  cout << hex;
  cout << "Attribute[0] = " << lpAttribute[0] << endl;
  cout << "Attribute[1] = " << lpAttribute[1] << endl;
  cout << "Attribute[2] = " << lpAttribute[2] << endl;
  cout << "Attribute[3] = " << lpAttribute[3] << endl;
  // ждем команду на завершение программы
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

Глава 13

Ввод-вывод на консоль

13.1. Ввод-вывод высокого уровня

К функциям ввода-вывода высокого уровня относятся следующие функции: `WriteConsole`, `ReadConsole`, `WriteFile` и `ReadFile`. Сейчас подробно рассмотрим работу с функциями `ReadConsole` и `WriteConsole`. Затем рассмотрим функции `WriteFile` и `ReadFile`.

Для чтения строки символов из входного буфера консоли экрана используется функция `ReadConsole`, которая имеет следующий прототип:

```
BOOL  ReadConsole(
  HANDLE   hConsoleInput,          // дескриптор буфера экрана
  LPVOID   lpBuffer,               // массив для ввода символов
  DWORD    nNumberOfCharsToRead,   // количество читаемых символов
  LPDWORD  lpNumberOfCharsRead,    // количество прочитанных символов
  LPVOID   lpReserved              // зарезервировано
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`.

Назначение параметров этой функции очевидно. Отметим лишь, что последний параметр всегда должен быть установлен в `NULL`. Функция `ReadConsole` вводит символы последовательно друг за другом, при этом курсор передвигается в следующую свободную позицию. Если включен режим отображения введенных символов, а он включен по умолчанию, то при вводе символы отображаются на экране. Подробно режимы управления вводом-выводом рассмотрены в *разд. 13.3*. Кроме того, в процессе ввода символов из входного буфера консоли эта функция игнорирует все события ввода, которые отличаются от ввода символа.

Для записи строки символов в буфер экрана используется функция `WriteConsole`, которая имеет следующий прототип:

```
BOOL  WriteConsole (
  HANDLE  hConsoleOutput,           // дескриптор буфера экрана
  CONST  VOID  *lpBuffer,           // массив с символами для вывода
  DWORD   nNumberOfCharsToWrite,    // количество записываемых символов
  LPDWORD lpNumberOfCharsWritten,   // количество записанных символов
  LPVOID  lpReserved                // зарезервировано
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Как и в случае с функцией `ReadConsole` параметр `lpReserved` всегда должен быть установлен в `NULL`.

В листинге 13.1 приведена программа, которая выводит и вводит на консоль информацию, используя функции `WriteConsole` и `ReadConsole`.

Листинг 13.1. Чтение и запись на консоль посредством функций `ReadConsole` и `WriteConsole`

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hStdOut, hStdIn;      // дескрипторы консоли
  DWORD  dwWritten, dwRead;     // для количества символов
  char  buffer[80];             // для ввода символов
  char  str[] = "Input any string:";
  char  c;

  // читаем дескрипторы консоли
  hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
  hStdIn = GetStdHandle(STD_INPUT_HANDLE);
  if (hStdOut == INVALID_HANDLE_VALUE || hStdIn == INVALID_HANDLE_VALUE)
  {
    cout << "Get standard handle failed." << endl;
    return GetLastError();
  }
  // выводим сообщения о вводе строки
  if(!WriteConsole(hStdOut, &str, sizeof(str), &dwWritten, NULL))
```

```
  {
    cout << "Write console failed." << endl;
    return GetLastError();
  }
  // вводим строку
  if(!ReadConsole(hStdIn, &buffer, sizeof(buffer), &dwRead, NULL))
  {
    cout << "Read console failed." << endl;
    return GetLastError();
  }
  // ждем команду на завершение работы
  cout << "Input any char to exit: ";
  cin >> c;

  return 0;
}
```

Теперь скажем о том, что ввод-вывод на консоль при помощи функций `ReadFile` и `WriteFile` выполняется аналогично функциям `ReadConsole` и `WriteConsole`. То есть эти функции имеют аналогичные списки параметров. Единственное отличие заключается в том, что последний параметр в функциях `ReadFile` и `WriteFile` указывает на синхронный или асинхронный ввод-вывод. Более подробно эти функции будут рассмотрены в *гл. 24*, посвященной вводу-выводу из файлов. В листинге 13.2 приведен текст программы, в которой эти функции используются для ввода-вывода на консоль.

Листинг 13.2. Чтение и запись на консоль посредством функций `ReadFile` и `WriteFile`

```
#include <windows.h>

HANDLE hStdOut, hStdIn;

int main(void)
{
  LPSTR lpszPrompt1 = "Input 'q' and press Enter to exit.\n";
  LPSTR lpszPrompt2 = "Input string and press Enter:\n";
  CHAR  chBuffer[80];
  DWORD cRead, cWritten;

  // читаем дескрипторы стандартного ввода и вывода
```

```
hStdIn = GetStdHandle(STD_INPUT_HANDLE);
hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
if (hStdIn == INVALID_HANDLE_VALUE || hStdOut == INVALID_HANDLE_VALUE)
{
  MessageBox(NULL, "Get standard handle failed", "Win32 API error",
    MB_OK | MB_ICONINFORMATION);
  return GetLastError();
}
// по умолчанию установлены режимы ввода: ENABLE_LINE_INPUT,
// ENABLE_ECHO_INPUT, ENABLE_PROCESSED_INPUT

// выводим сообщение о том, как выйти из цикла чтения
if (!WriteFile(
    hStdOut,       // дескриптор стандартного вывода
    lpszPrompt1,   // строка, которую выводим
    lstrlen(lpszPrompt1),  // длина строки
    &cWritten,     // количество записанных байтов
    NULL))         // синхронный вывод
  {
    MessageBox(NULL, "Write file failed", "Win32 API error",
      MB_OK | MB_ICONINFORMATION);
    return GetLastError();
  }
// цикл чтения
for (;;)
{
  // выводим сообщение о вводе строки
  if (!WriteFile(
      hStdOut,       // дескриптор стандартного вывода
      lpszPrompt2,   // строка, которую выводим
      lstrlen(lpszPrompt2),  // длина строки
      &cWritten,     // количество записанных байтов
      NULL))         // синхронный вывод
    {
      MessageBox(NULL, "Write file failed", "Win32 API error",
        MB_OK | MB_ICONINFORMATION);
      return GetLastError();
    }
  // вводим строку с клавиатуры и дублируем ее на экран
```

```
    if (!ReadFile(
        hStdIn,    // дескриптор стандартного ввода
        chBuffer,  // буфер для чтения
        80,        // длина буфера
        &cRead,    // количество прочитанных байтов
        NULL))     // синхронный ввод
      {
        MessageBox(NULL, "Write file failed", "Win32 API error",
          MB_OK | MB_ICONINFORMATION);
        return GetLastError();
      }
    // выход из программы
    if (chBuffer[0] == 'q')
      return 1;
  }

  return 0;
}
```

В заключение данного параграфа сделаем следующие замечания. В случае с консолью функции `WriteFile`, `ReadFile`, `WriteConsole` и `ReadConsole` читают и записывают символы потоком. Функции `ReadFile` и `WriteFile` работают только с символами, заданными в кодировке ASCII. Функции `ReadConsole` и `WriteConsole` отличаются от файловых функций только тем, что работают также с символами, заданными в кодировке Unicode. Кроме того, эти функции не обрабатывают управляющие символы при вводе-выводе на консоль.

13.2. Ввод низкого уровня

Функции ввода низкого уровня работают непосредственно с записями входного буфера консоли. К ним относятся функции `ReadConsoleInput`, `PeekConsoleInput`, `WriteConsoleInput`, `GetNumberOfConsoleInputEvents` и `FlushConsoleInputBuffer`, работу с которыми мы сейчас рассмотрим.

Для чтения записей из входного буфера используется функция `ReadConsoleInput`, которая имеет следующий прототип:

```
BOOL  ReadConsoleInput(
  HANDLE   hConsoleInput,        // дескриптор входного буфера консоли
  PINPUT_RECORD  lpBuffer,       // буфер данных
  DWORD    nLength,              // количество читаемых записей
```

```
    LPDWORD  lpNumberOfEventsRead  // количество прочитанных записей
  );
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Отметим, что после чтения записи из входного буфера консоли функция `ReadConsoleInput` удаляет ее оттуда. Теперь перейдем к описанию параметров этой функции.

Параметр `hConsoleInput` должен содержать дескриптор входного буфера консоли.

Параметр `lpBuffer` должен указывать на область памяти, в которую будут читаться записи из входного буфера консоли.

Параметр `nLength` должен содержать количество записей, которые пользователь хочет прочитать из входного буфера консоли. А по адресу, заданному параметром `lpNumberOfEventsRead`, функция запишет количество прочитанных записей из буфера консоли.

Для чтения записей из входного буфера консоли, не удаляя их оттуда, используется функция `PeekConsoleInput`. Параметры этой функции полностью совпадают с параметрами функции `ReadConsoleInput`.

В листинге 13.3 приведен текст программы, которая читает по одной записи из входного буфера и просто распечатывает их содержимое на экран.

Листинг 13.3. Чтение записей входного буфера консоли

```
#include <windows.h>
#include <iostream.h>

HANDLE  hStdIn, hStdOut; // для дескрипторов стандартного ввода и вывода
BOOL  bRead = TRUE;      // для цикла обработки событий

// функция обработки сообщений от клавиатуры
VOID KeyEventProc(KEY_EVENT_RECORD kir)
{
  cout << "\tKey event record:" << endl;
    // просто выводим на консоль содержимое записи
  cout << "bKeyDown = " << hex << kir.bKeyDown << endl;
  cout << "wRepeatCount = " << dec << kir.wRepeatCount << endl;
  cout << "wVirtualKeyCode = " << hex << kir.wVirtualKeyCode << endl;
  cout << "wVirtualScanCode = " << kir.wVirtualScanCode << endl;
  cout << "uChar.AsciiChar = " << kir.uChar.AsciiChar << endl;
```

```
  cout << "dwControlKeyState = " << kir.dwControlKeyState << endl;

  // если ввели букву 'q', то выходим из цикла обработки событий
  if (kir.uChar.AsciiChar == 'q')
    bRead = FALSE;
}

// функция обработки сообщений от мыши
VOID MouseEventProc(MOUSE_EVENT_RECORD mer)
{
  cout << "\tMouse event record:" << endl << dec;
    // просто выводим на консоль содержимое записи
  cout << "dwMousePosition.X = " << mer.dwMousePosition.X << endl;
  cout << "dwMousePosition.Y = " << mer.dwMousePosition.Y << endl;
  cout << "dwButtonState = " << hex << mer.dwButtonState << endl;
  cout << "dwControlKeyState = " << mer.dwControlKeyState << endl;
  cout << "dwEventFlags = " << mer.dwEventFlags << endl;
}

// функция обработки сообщения об изменении размеров окна
VOID ResizeEventProc(WINDOW_BUFFER_SIZE_RECORD wbsr)
{
  // изменяем размеры буфера вывода
  SetConsoleScreenBufferSize(hStdOut, wbsr.dwSize);
}


int main()
{
  INPUT_RECORD  ir;     // входная запись
  DWORD  cNumRead;      // для количества прочитанных записей

  // получить дескрипторы стандартного ввода и вывода
  hStdIn = GetStdHandle(STD_INPUT_HANDLE);
  if (hStdIn == INVALID_HANDLE_VALUE)
  {
    cout << "Get standard input handle failed." << endl;
    return GetLastError();
  }
```

```
hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
if (hStdOut == INVALID_HANDLE_VALUE)
{
  cout << "Get standard output handle failed." << endl;
  return GetLastError();
}
// начинаем обработку событий ввода
cout << "Begin input event queue processing." << endl;
cout << "Input 'q' to quit."<< endl << endl;
// цикл обработки событий ввода
while (bRead)
{
  // ждем событие ввода
  WaitForSingleObject(hStdIn, INFINITE);

  // читаем запись ввода
  if (!ReadConsoleInput(
      hStdIn,      // дескриптор ввода
      &ir,         // буфер для записи
      1,           // читаем одну запись
      &cNumRead))  // количество прочитанных записей
  {
    cout << "Read console input failed." << endl;
    break;
  }

  // вызываем соответствующий обработчик
  switch(ir.EventType)
  {
    case KEY_EVENT:        // событие ввода с клавиатуры
      KeyEventProc(ir.Event.KeyEvent);
      break;

    case MOUSE_EVENT:      // событие ввода с мыши
      MouseEventProc(ir.Event.MouseEvent);
      break;

    case WINDOW_BUFFER_SIZE_EVENT:   // изменения размеров окна
      ResizeEventProc(
```

```
          ir.Event.WindowBufferSizeEvent);
        break;

      case FOCUS_EVENT:       // события фокуса ввода игнорируем
        break;

      case MENU_EVENT:        // события меню игнорируем
        break;

      default:                // неизвестное событие
        cout << "Unknown event type.";
        break;
    }
  }

  return 0;
}
```

Замечание

Отметим, что в операционной системе Windows 2000 программа из листинга 13.3 работает только в полноэкранном режиме. Для того чтобы эта программа работала в графическом режиме нужно поступить следующим образом (для русской локализации). Правой кнопкой мыши щелкнуть по пиктограмме консольного приложения. Затем в появившемся диалоговом окне **Свойства** перейти на вкладку **Общие**. На этой вкладке сбросить в группе **Редактирование** флаг **Выделение мышью**.

Теперь рассмотрим функции `WriteConsoleInput`, `GetNumberOfConsoleInputEvents` и `FlushConsoleInputBuffer`.

Функция `WriteConsoleInput` предназначена для записи событий ввода во входной буфер консоли. Эта функция имеет следующий прототип:

```
BOOL  WriteConsoleInput(
  HANDLE   hConsoleInput,       // дескриптор входного буфера консоли
  CONST  INPUT_RECORD  *lpBuffer,    // указатель на буфер с записями
  DWORD    nLength,                  // количество записываемых записей
  LPDWORD  lpNumberOfEventsWritten  // количество записанных записей
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. Кратко опишем назначение параметров этой функции.

Параметр `hConsoleInput` должен содержать дескриптор входного буфера консоли.

Параметр `lpBuffer` должен указывать на область памяти, из которой будут записываться записи во входной буфер консоли.

Параметр `nLength` должен содержать количество записей, которые пользователь хочет записать во входной буфер консоли. А по адресу, заданному параметром `lpNumberOfEventsWritten`, функция запишет количество записей во входном буфере консоли.

Для чтения количества записей, находящихся во входном буфере консоли используется функция `GetNumberOfConsoleInputEvents`, которая имеет следующий прототип:

```
BOOL  GetNumberOfConsoleInputEvents(
  HANDLE   hConsoleInput,      // дескриптор входного буфера консоли
  LPDWORD  lpNumberOfEvents    // указатель на количество записей
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. В случае успешного завершения адресу, заданному параметром `lpNumberOfEvents`, эта функция записывает количество записей, находящихся в буфере ввода консоли.

Для очистки буфера ввода консоли используется функция `FlushConsoleInputBuffer`, которая имеет следующий прототип:

```
BOOL  FlushConsoleInputBuffer(
  HANDLE  hConsoleInput,       // дескриптор входного буфера консоли
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — `FALSE`.

В листинге 13.4 приведена программа, в которой показаны примеры использования функций `WriteConsoleInput`, `GetNumberOfConsoleInputEvents` и `FlushConsoleInputBuffer`.

Листинг 13.4. Пример записи в буфер ввода, подсчет количества записей и очистка буфера ввода

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hStdIn;      // для дескриптора стандартного ввода
```

```
INPUT_RECORD  ir;    // входная запись
DWORD  dwNumberWritten;  // количество записанных записей
DWORD  dwNumber;     // для количества записей в буфере ввода

// получить дескриптор стандартного ввода
hStdIn = GetStdHandle(STD_INPUT_HANDLE);
if (hStdIn == INVALID_HANDLE_VALUE)
{
  cout << "Get standard input handle failed." << endl;
  return GetLastError();
}
// подсчитываем записи в буфере ввода
if (!GetNumberOfConsoleInputEvents(hStdIn, &dwNumber))
{
  cout << "Get number of console input events failed." << endl;
  return GetLastError();
}
// печатаем количество событий ввода
cout << "Number of console input events = " << dwNumber << endl;
// инициализируем запись события ввода
ir.EventType = KEY_EVENT;
ir.Event.KeyEvent.bKeyDown = 0x1;
ir.Event.KeyEvent.wRepeatCount = 1;
ir.Event.KeyEvent.wVirtualKeyCode = 0x43;
ir.Event.KeyEvent.wVirtualScanCode = 0x2e;
ir.Event.KeyEvent.uChar.AsciiChar = 'c';
ir.Event.KeyEvent.dwControlKeyState =0x20;
// записываем запись в буфер ввода
if (!WriteConsoleInput(hStdIn, &ir, 1, &dwNumberWritten))
{
  cout << "Write console input failed." << endl;
  return GetLastError();
}
cout << "Write one record into the input buffer." << endl;
// подсчитываем записи в буфере ввода
if (!GetNumberOfConsoleInputEvents(hStdIn, &dwNumber))
{
  cout << "Get number of console input events failed." << endl;
  return GetLastError();
```

```
  }
  // печатаем количество событий ввода
  cout << "Number of console input events = " << dwNumber << endl;
  // очищаем входной буфер
  cout << "Flush console input buffer." << endl;
  if (!FlushConsoleInputBuffer(hStdIn))
  {
    cout << "Flush console input buffer failed." << endl;
    return GetLastError();
  }
  // подсчитываем записи в буфере ввода
  if (!GetNumberOfConsoleInputEvents(hStdIn, &dwNumber))
  {
    cout << "Get number of console input events failed." << endl;
    return GetLastError();
  }
  // печатаем количество событий ввода
  cout << "Number of console input events = " << dwNumber << endl;

  return 0;
}
```

Для определения количества кнопок у мыши используется функция `GetNumberOfConsoleMouseButton`, которая имеет следующий прототип:

```
BOOL  GetNumberOfConsoleMouseButton(
  LPDWORD  lpNumberOfMouseButtons  // количество кнопок у мыши
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. При успешном завершении эта функция записывает по адресу, заданному параметром `lpNumberOfMouseButtons`, количество кнопок у мыши.

В листинге 13.5 приведена программа, которая показывает пример использования функции `GetNumberOfConsoleMouseButton`.

Листинг 13.5. Определение количества кнопок у мыши

```
#include <windows.h>
#include <iostream.h>

int main()
```

```
{
  DWORD  dwNumber;      // для количества кнопок у мыши

  // подсчитываем количество кнопок у мыши
  if (!GetNumberOfConsoleMouseButtons(&dwNumber))
  {
    cout << "Get number of console mouse buttons failed." << endl;
    return GetLastError();
  }
  // выводим количество кнопок у мыши
  cout << "Number of console mouse buttons = " << dwNumber << endl;

  return 0;
}
```

13.3. Вывод низкого уровня

Функции вывода низкого уровня работают непосредственно с элементами буфера экрана. Эти функции можно разбить на три группы.

- Чтение и запись последовательности символов:
  - `ReadConsoleOutputCharacter` — чтение последовательности символов из буфера экрана;
  - `WriteConsoleOutputCharacter` — запись последовательности символов в буфер экрана.
- Заполнение буфера экрана заданным символом:
  - `FillConsoleOutputCharacter` — заполнение буфера экрана.
- Чтение и запись прямоугольных областей символов:
  - `ReadConsoleOutput` — чтение прямоугольной области символов из буфера экрана;
  - `WriteConsoleOutput` — запись прямоугольной области символов в буфер экрана.

Рассмотрим работу с этими функциями подробнее. Сначала рассмотрим функции из первой группы, которые предназначены для работы с последовательностями символов.

Для ввода последовательности символов из буфера экрана используется функция `ReadConsoleOutputCharacter`, которая имеет следующий прототип:

```
BOOL  ReadConsoleOutputCharacter(
  HANDLE  hConsoleOutput,    // дескриптор буфера экрана
```

```
    LPTSTR  lpCharacter,       // указатель на буфер с символами
    DWORD   nLength,           // количество читаемых символов
    COORD   dwReadCoord,       // координаты первого читаемого символа
    LPDWORD lpNumberOfCharsRead   // количество прочитанных символов
  );
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. Кратко опишем назначение параметров этой функции.

Параметр `hConsoleOutput` этой функции должен содержать дескриптор буфера экрана.

Параметр `lpCharacter` должен указывать на область памяти, в которую будет читаться содержимое буфера экрана.

Параметр `nLength` должен содержать количество читаемых символов из буфера экрана.

Параметр `dwReadCoord` должен содержать координаты первого символа, который будет читаться из буфера экрана. При этом поля `X` и `Y` структуры `COORD` задают соответственно индексы строки и столбца первого читаемого элемента.

По адресу, заданному параметром `lpNumberOfCharsRead`, функция `ReadConsoleOutputCharacter` возвратит количество прочитанных символов из буфера экрана.

В листинге 13.6 приведена программа, которая демонстрирует использование функции `ReadConsoleOutputCharacter` для чтения содержимого прямоугольной области буфера экрана.

Листинг 13.6. Чтение последовательности символов из буфера экрана

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hConsoleOutput;     // для дескриптора буфера экрана
  CHAR    lpBuffer[80];     // буфер для ввода
  COORD    dwReadCoord = {0, 0};  // координаты первого элемента в буфере
  DWORD  nNumberOfCharsRead;  // количество прочитанных символов

  // получаем дескриптор буфера экрана
  hConsoleOutput = GetStdHandle(STD_OUTPUT_HANDLE);
```

```
  if (hConsoleOutput == INVALID_HANDLE_VALUE)
  {
    cout << "Get standard handle failed." << endl;
    return GetLastError();
  }
  // выводим те символы, которые будем читать
  cout << 'a' << 'b' << endl;
  // читаем эти символы в буфер
  if (!ReadConsoleOutputCharacter(
      hConsoleOutput,  // дескриптор буфера экрана
      lpBuffer,        // буфер для ввода символов
      2,               // количество читаемых символов
      dwReadCoord,     // координата первого символа
      &nNumberOfCharsRead))  // количество прочитанных символов
  {
    cout << "Read consoleoutput character failed." << endl;
    return GetLastError();
  }
    // выводим количество прочитанных символов и сами символы
  cout << "Number of chars read: " << nNumberOfCharsRead << endl;
  cout << "Read chars: " << lpBuffer[0] << lpBuffer[1] << endl;

  return 0;
}
```

Для записи последовательности символов в буфер экрана используется функция `WriteConsoleOutputCharacter`, которая имеет следующий прототип:

```
  BOOL  WriteConsoleOutputCharacter(
    HANDLE  hConsoleOutput, // дескриптор буфера экрана
    LPCTSTR lpCharacter,    // указатель на массив символов
    DWORD   nLength,        // количество записываемых символов
    COORD   dwWriteCoord,   // координаты первого символа в буфере экрана
    LPDWORD lpNumberOfCharsWritten   // количество символов,
                                     // записанных в буфер экрана
  );
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Эта функция записывает последовательность символов, на которую указывает параметр `lpCharacter`, в буфер

экрана, начиная с позиции, на которую указывает параметр `dwWriteCoord`. Количество записываемых символов задается параметром `nLength`, а количество фактически записанных символов функция `WriteConsoleOutputCharacter` возвращает по адресу, заданному параметром `lpNumberOfCharsWritten`.

В листинге 13.7 приведена программа, в которой показано, как можно вывести последовательность символов с заданной позиции в буфер экрана, используя функцию `WriteConsoleInputCharacter`.

Листинг 13.7. Запись последовательности символов в буфер экрана

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hConsoleOutput;         // для дескриптора буфера экрана
  CHAR    lpBuffer[] = "abcd";    // буфер с символами для вывода
  COORD   dwWriteCoord = {10, 10};// координаты первого элемента в буфере
  DWORD   nNumberOfCharsWritten;  // количество записанных символов

  // получаем дескриптор буфера экрана
  hConsoleOutput = GetStdHandle(STD_OUTPUT_HANDLE);
  if (hConsoleOutput == INVALID_HANDLE_VALUE)
  {
    cout << "Get standard handle failed." << endl;
    return GetLastError();
  }
  // записываем символы в буфер экрана
  if (!WriteConsoleOutputCharacter(
      hConsoleOutput,       // дескриптор буфера экрана
      lpBuffer,       // буфер для ввода символов
      sizeof(lpBuffer),       // количество записываемых символов
      dwWriteCoord,       // координата первого символа
      &nNumberOfCharsWritten))  // количество записанных символов
  {
    cout << "Read console output character failed." << endl;
    return GetLastError();
  }
  // выводим количество записанных символов
```

```
  cout << "Number of chars written: " << nNumberOfCharsWritten << endl;

  return 0;
}
```

Теперь рассмотрим функцию из второй группы нашей классификации. Для заполнения всего или части буфера экрана определенным символом используется функция `FillConsoleOutputCharacter`, которая имеет следующий прототип:

```
BOOL  FillConsoleOutputCharacter(
  HANDLE  hConsoleOutput,  // дескриптор буфера экрана
  TCHAR   cCharacter,      // символ-заполнитель
  DWORD   nLength,         // длина заполняемой области
  COORD   dwWriteCoord,    // координаты первой клетки буфера экрана
  LPDWORD lpNumberOfCharsWritten   // количество заполненных клеток
);
```

В случае успеха функция возвращает ненулевое значение, а случае неудачи — значение `FALSE`. Назначение параметров этой функции очевидно. Назначение параметров этой функции очевидно, принимая во внимание рассмотренные нами выше функции.

В листинге 13.8 приведен пример программы, в которой эта функция используется для заполнения буфера экрана заданным символом, а затем для очистки буфера экрана.

Листинг 13.8. Заполнение буфера экрана заданным символом

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char c;
  HANDLE  hStdOut;   // дескриптор стандартного вывода
  DWORD  dwLength;   // количество заполняемых клеток
  DWORD  dwWritten;  // для количества заполенных клеток
  COORD  coord;      // координаты первой клетки
  CONSOLE_SCREEN_BUFFER_INFO csbi;   // для параметров буфера экрана

  // читаем дескриптор стандартного вывода
  hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
```

```
if (hStdOut == INVALID_HANDLE_VALUE)
{
  cout << "Get standard handle failed." << endl;
  return GetLastError();
}
// читаем параметры выходного буфера
if (!GetConsoleScreenBufferInfo(hStdOut, &csbi))
{
  cout << "Console screen buffer info failed." << endl;
  return GetLastError();
}
// вычисляем размер буфера экрана в символах
dwLength = csbi.dwSize.X * csbi.dwSize.Y;
// устанавливаем координаты первой клетки
coord.X = 0;
coord.Y = 0;
// вводим символ-заполнитель
cout << "Input any char to fill screen buffer: ";
cin >> c;
// заполняем буфер экрана символом-заполнителем
if (!FillConsoleOutputCharacter(
    hStdOut,     // стандартный дескриптор вывода
    c,           // символ заполнения
    dwLength,    // длина буфера в символах
    coord,       // индекс первой клетки
    &dwWritten)) // количество заполненных клеток
{
  cout << "Fill console output character failed." << endl;
  return GetLastError();
}
// ждем команды на очищение буфера экрана
cout << "In order to clear screen buffer, press any char: ";
cin >> c;
// очищаем буфер экрана пробелами
if (!FillConsoleOutputCharacter(
    hStdOut,     // стандартный дескриптор вывода
    ' ',         // символ заполнения
    dwLength,    // длина буфера в символах
    coord,       // индекс первой клетки
```

```
      &dwWritten)) // количество заполненных клеток
  {
    cout << "Fill console output character failed." << endl;
    return GetLastError();
  }

  return 0;
}
```

Теперь перейдем к функциям из третьей группы, которые предназначены для чтения и записи прямоугольных областей символов. Функция `ReadConsoleOutput` предназначена для чтения прямоугольных областей символов и их атрибутов из буфера экрана. Эта функция имеет следующий прототип:

```
BOOL  ReadConsoleOutput(
  HANDLE  hConsoleOutput,  // дескриптор буфера экрана
  PCHAR_INFO  lpBuffer,    // указатель на буфер с данными
  COORD   dwBufferSize,    // размер буфера с данными
  COORD   dwBufferCoord,   // координаты для первого элемента в буфере
  PSMALL_RECT  lpReadRegion    // область ввода в буфере экрана
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Кратко опишем назначение параметров этой функции.

Параметр `hConsoleOutput` этой функции должен содержать дескриптор буфера экрана.

Параметр `lpBuffer` должен указывать на область памяти, в которую будет читаться содержимое буфера экрана. Причем эта область памяти рассматривается функцией `ReadConsoleOutput` как двумерный массив, элементами которого являются струкутры типа `CHAR_INFO`. Размерность этого массива задается параметром `dwBufferSize`. При этом поля `X` и `Y` структуры `COORD` задают соответственно количество столбцов и строк этого массива.

Параметр `lpReadRegion` является указателем на структуру типа `SMALL_RECT`, которая содержит координаты левого верхнего и правого нижнего углов прямоугольника в буфере экрана, содержимое которого будет прочитано в область памяти, заданную параметром `lpBuffer`.

В листинге 13.9 приведена программа, которая демонстрирует использование функции `ReadConsoleOutput` для чтения содержимого прямоугольной области буфера экрана.

Листинг 13.9. Чтение прямоугольной области из буфера экрана

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE    hConsoleOutput;    // для дескриптора буфера экрана
  CHAR_INFO  lpBuffer[4];      // буфер для ввода
  COORD    dwBufferSize = {2, 2};    // размеры буфера
  COORD    dwBufferCoord = {0, 0}; // координаты первого элемента в буфере
  SMALL_RECT  ReadRegion = {0, 0, 1, 1}; // прямоугольник, который читаем

  // выводим символы, которые будем читать
  cout << 'a' << 'b' << endl << 'c' << 'd' << endl;
  // получаем дескриптор ввода
  hConsoleOutput = GetStdHandle(STD_OUTPUT_HANDLE);
  if (hConsoleOutput == INVALID_HANDLE_VALUE)
  {
    cout << "Get standard handle failed." << endl;
    return GetLastError();
  }
  // читаем символы
  if (!ReadConsoleOutput(hConsoleOutput, lpBuffer, dwBufferSize,
  dwBufferCoord, &ReadRegion))
  {
    cout << "Read console input failed." << endl;
    return GetLastError();
  }
  // распечатываем прочитанные символы
  cout << "Read cells." << hex << endl;
  for (int i = 0; i < 4; ++i)
    cout << lpBuffer[i].Attributes << ' ' << lpBuffer[i].Char.AsciiChar
<< endl;

  return 0;
}
```

Для записи прямоугольных областей символов и их атрибутов в буфер экрана используется функция `WriteConsoleOutput`, которая имеет следующий прототип:

```
BOOL  WriteConsoleOutput(
  HANDLE  hConsoleOutput,    // дескриптор буфера экрана
  CONST  CHAR_INFO  *lpBuffer,   // указатель на буфер с данными
  COORD   dwBufferSize,      // размер буфера с данными
  COORD   dwBufferCoord,     // координаты первого элемента в буфере
  PSMALL_RECT  lpWriteRegion     // область вывода в буфере экрана
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Эта функция записывает символы и их атрибуты из буфера, на который указывает параметр `lpBuffer`, в область вывода буфера экрана, на которую указывает параметр `lpWriteRegion`. Остальные параметры этой функции имеют тот же смысл, что и параметры функции `ReadConsoleOutput`.

В листинге 13.10 приведена программа, которая показывает, как можно заполнить прямоугольник в буфере экрана пробелами, используя функцию `WriteConsoleInput`.

Листинг 13.10. Заполнение прямоугольной области в буфере экрана

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hStdOut;       // дескриптор стандартного вывода
  CHAR_INFO  ci[80*25];  // прямоугольник, из которого будем выводить
  COORD    size;         // размеры этого прямоугольника
  // координаты левого угла прямоугольника, из которого выводим
  COORD    coord;
  // координаты левого угла прямоугольника, в который пишем
  SMALL_RECT  sr;

  // читаем стандартный дескриптор вывода
  hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
  // заполняем прямоугольник, который будем выводить, пробелами
  for (int i = 0; i < 80*25; ++i)
```

```
  {
    ci[i].Char.AsciiChar = ' ';
    ci[i].Attributes = BACKGROUND_BLUE | BACKGROUND_INTENSITY;
  }
  // устанавливаем левый угол многоугольника, из которого пишем
  coord.X = 0;
  coord.Y = 0;
  // устанавливаем размеры прямоугольника, который пишем
  size.X = 80;
  size.Y = 25;
  // вводим координаты левого верхнего угла многоугольника,
  // в который пишем
  cout << "Input left coordinate to write: ";
  cin >> sr.Left;
  cout << "Input top coordinate to write: ";
  cin >> sr.Top;
  // вводим координаты правого нижнего угла прямоугольника,
  // в который пишем
  cout << "Input right coordinate to write: ";
  cin >> sr.Right;
  cout << "Input down coordinate to write: ";
  cin >> sr.Bottom;

  // пишем прямоугольник в буфер экрана
  if (!WriteConsoleOutput(
      hStdOut,    // дескриптор буфера экрана
      ci,         // прямоугольник, из которого пишем
      size,       // размеры этого прямоугольника
      coord,      // и его левый угол
      &sr))       // прямоугольник, в который пишем
  {
    cout << "Write console output failed." << endl;
    return GetLastError();
  }

  return 0;
}
```

13.4. Режимы ввода-вывода консоли

Функции ввода-вывода на консоль могут работать в нескольких режимах, которые устанавливаются функцией SetConsoleMode, которая имеет следующий прототип:

```
BOOL  SetConsoleMode(
  HANDLE  hConsoleHandle,  // дескриптор ввода или вывода
  DWORD   dwMode           // режим ввода или вывода
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение FALSE.

Если параметр hConsoleHandle является дескриптором входного буфера, то режим ввода, который описывается параметром dwMode, может быть произвольной комбинацией следующих флагов:

- ENABLE_LINE_INPUT — функции ReadFile и ReadConsole читают символы до тех пор, пока не встретят символ ENTER. В противном случае читается доступное количество символов во входном буфере;
- ENABLE_ECHO_INPUT — символы, прочитанные функциями ReadFile и ReadConsole, выводятся на экран. Этот режим может использоваться только совместно с режимом ENABLE_LINE_INPUT;
- ENABLE_PROCESSED_INPUT — комбинация клавиш <Ctrl>+<C> обрабатывается системой. Если этот режим используется совместно с режимом ENABLE_LINE_INPUT, то управляющие символы \b, \r и \n также обрабатываются системой;
- ENABLE_WINDOW_INPUT — изменение размеров окна обрабатывается приложением;
- ENABLE_MOUSE_INPUT — сообщения от мыши обрабатываются приложением.

Отметим следующий момент: как видно из описания режимов, значения ENABLE_LINE_INPUT, ENABLE_ECHO_INPUT и ENABLE_PROCESSED_INPUT влияют на работу функций ввода-вывода высокого уровня, а значения ENABLE_WINDOW_INPUT и ENABLE_MOUSE_INPUT — на работу функций низкого уровня ввода-вывода.

Если параметр hConsoleHandle является дескриптором буфера экрана, то режим ввода, который задается параметром dwMode, может быть комбинацией следующих значений:

- ENABLE_PROCESSED_OUTPUT — функции WriteFile и WriteConsole обрабатывают во входном буфере управляющие символы: \n, \t, \b, \v и \a. Эти же действия выполняются при выводе на экран ввода функций ReadFile и ReadConsole;

- ENABLE_WRAP_AT_EOL_OUTPUT — разрешает прокручивать буфер экрана при выводе данных на консоль.

Для чтения установленных режимов ввода-вывода используется функция GetConsoleMode, которая имеет следующий прототип:

```
BOOL  GetConsoleMode(
  HANDLE  hConsoleHandle,    // входной или выходной дескриптор консоли
  LPDWORD lpMode             // указатель на флаги режимов консоли
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — значение FALSE. Если функция GetConsoleMode выполнилась успешно, то в двойном слове, на которое ссылается параметр LpMode, будут установлены флаги режимов консоли.

В листинге 13.11 в качестве примера приведена программа, в которой отключается режим эхо-вывода на экран введенных с консоли символов.

Листинг 13.11. Отключение режима эхо-вывода

```
#include <windows.h>

HANDLE hStdOut, hStdIn;

int main(void)
{
  LPSTR lpszPrompt1  = "Input ESC and press Enter to exit.\n";
  LPSTR lpszPrompt2 = "Input string and press Enter:\n";
  CHAR  chBuffer[80];
  DWORD cRead, cWritten;
  DWORD dwOldMode, dwNewMode;

  // читаем дескрипторы стандартного ввода и вывода
  hStdIn = GetStdHandle(STD_INPUT_HANDLE);
  hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
  if (hStdIn == INVALID_HANDLE_VALUE || hStdOut == INVALID_HANDLE_VALUE)
  {
    MessageBox(NULL, "Get standard handle failed.", "Win32 API error",
      MB_OK | MB_ICONINFORMATION);
    return GetLastError();
  }
```

```
// читаем режимы, установленные по умолчанию
if (!GetConsoleMode(hStdIn, &dwOldMode))
{
  MessageBox(NULL, "Get console mode failed.", "Win32 API error",
    MB_OK | MB_ICONINFORMATION);
  return GetLastError();
}
// отключаем режим ENABLE_ECHO_INPUT
dwNewMode = dwOldMode & ~ENABLE_ECHO_INPUT;
// устанавливаем новый режим
if (!SetConsoleMode(hStdIn, dwNewMode))
{
  MessageBox(NULL, "Set console mode failed.", "Win32 API error",
    MB_OK | MB_ICONINFORMATION);
  return GetLastError();
}
// выводим сообщение о том, как выйти из цикла чтения
if (!WriteConsole(
    hStdOut,       // дескриптор стандартного вывода
    lpszPrompt1,   // строка, которую выводим
    lstrlen(lpszPrompt1),  // длина строки
    &cWritten,     // количество записанных байтов
    NULL))         // не используется
{
  MessageBox(NULL, "Write file failed.", "Win32 API error",
    MB_OK | MB_ICONINFORMATION);
  return GetLastError();
}
// цикл чтения
for (;;)
{
  // выводим сообщение о вводе строки
  if (!WriteConsole(
      hStdOut,       // дескриптор стандартного вывода
      lpszPrompt2,   // строка, которую выводим
      lstrlen(lpszPrompt2),  // длина строки
```

```
        &cWritten,     // количество записанных байтов
        NULL))         // не используется
    {
      MessageBox(NULL, "Write file failed.", "Win32 API error",
        MB_OK | MB_ICONINFORMATION);
      return GetLastError();
    }
    // вводим строку с клавиатуры
    if (!ReadConsole(
        hStdIn,    // дескриптор стандартного ввода
        chBuffer,  // буфер для чтения
        80,        // длина буфера
        &cRead,    // количество прочитанных байт
        NULL))     // не используется
    {
      MessageBox(NULL, "Read file failed.", "Win32 API error",
        MB_OK | MB_ICONINFORMATION);
      return GetLastError();
    }
    // выход из программы
    if (chBuffer[0] == '\033')
      return 1;
    // дублируем строку на экране
    if (!WriteConsole(
        hStdOut,   // дескриптор стандартного вывода
        chBuffer,  // строка, которую выводим
        cRead,     // длина строки
        &cWritten, // количество записанных байтов
        NULL))     // не используется
    {
      MessageBox(NULL, "Write file failed.", "Win32 API error",
        MB_OK | MB_ICONINFORMATION);
      return GetLastError();
    }
  }

  return 0;
}
```

13.5. Прокрутка буфера экрана

Для перемещения блоков данных внутри буфера экрана используется функция `ScrollConsoleScreenBuffer`, которая имеет следующий прототип:

```
BOOL  ScrollConsoleScreenBuffer(
  HANDLE hConsoleOutput,       // дескриптор буфера экрана
  CONST  SMALL_RECT  *lpScrollRectangle, // исходный прямоугольник
  CONST  SMALL_RECT  *lpClipRectangle,   // прямоугольник отсечения
  COORD  dwDestinationOrigin,  // целевой прямоугольник
  CONST  CHAR_INFO  *lpFill    // символ-заполнитель
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. Кратко опишем назначение параметров этой функции.

В параметре `hConsoleOutput` должен быть установлен дескриптор буфера экрана.

Параметр `lpScrollRectangle` должен указывать на прямоугольник, который будет перемещаться внутри буфера экрана.

Параметр `lpClipRectangle` должен указывать на прямоугольник, внутри которого выполняется изменение буфера экрана при перемещении исходного прямоугольника. За пределами этого прямоугольника изменения в буфере экрана производиться не будут. Такой процесс называется отсечением изменений вне прямоугольника, а сам прямоугольник называется прямоугольником отсечения.

Параметр `dwDestinationOrigin` должен указывать на прямоугольник, в который будет переписано содержимое исходного прямоугольника.

Параметр `lpFill` должен указывать на символ-заполнитель. Этим символом будут заполняться те области целевого прямоугольника, которые не перекрываются исходным прямоугольником.

Главным образом функция `ScrollConsoleScreenBuffer` используется для прокрутки буфера экрана. Как это можно сделать, показано в программе из листинга 13.12.

Листинг 13.12. Прокрутка буфера экрана

```
#include <windows.h>

HANDLE hStdOut, hStdIn;

// функция перехода на новую строку в буфере экрана
```

```
int GoToNewLine(void)
{
  CONSOLE_SCREEN_BUFFER_INFO csbi;   // информация о буфере экрана
  SMALL_RECT    srScroll;      // перемещаемый прямоугольник
  SMALL_RECT    srClip;        // рассматриваемая область
  COORD      coord;            // новое положение
  CHAR_INFO    ci;             // символ-заполнитель

  // читаем информацию о буфере экрана
  if (!GetConsoleScreenBufferInfo(hStdOut, &csbi))
  {
    MessageBox(NULL, "Get console screen buffer info failed.", "Win32 API error",
      MB_OK | MB_ICONINFORMATION);
    return 0;
  }
  // переходим на первый столбец
  csbi.dwCursorPosition.X = 0;
  // если это не последняя строка,
  if ((csbi.dwCursorPosition.Y+1) < csbi.dwSize.Y)
    // то переводим курсор на следующую строку
    csbi.dwCursorPosition.Y += 1;
  // иначе прокручиваем буфер экрана
  else
  {
    // координаты прямоугольника, который прокручиваем
    srScroll.Left = 0;
    srScroll.Top = 1;
    srScroll.Right = csbi.dwSize.X;
    srScroll.Bottom = csbi.dwSize.Y;
    // координаты прямоугольника буфера экрана
    srClip.Left = 0;
    srClip.Top = 0;
    srClip.Right = csbi.dwSize.X;
    srClip.Bottom = csbi.dwSize.Y;
    // устанавливаем новые координаты левого угла прямоугольника srScroll
    coord.X = 0;
    coord.Y = 0;
    // устанавливаем атрибуты и символ-заполнитель для последней строки
    ci.Attributes = csbi.wAttributes;
```

```
    ci.Char.AsciiChar = ' ';
    // прокручиваем прямоугольник srScroll
    if (!ScrollConsoleScreenBuffer(
        hStdOut,     // дескриптор стандартного вывода
        &srScroll,   // прокручиваемый прямоугольник
        &srClip,     // буфер экрана
        coord,       // начало буфера экрана
        &ci))        // атрибуты и символ-заполнитель
    {
      MessageBox(NULL, "Set console window info failed.", "Win32 API error",
        MB_OK | MB_ICONINFORMATION);
      return 0;
    }
  }
  // теперь устанавливаем курсор
  if (!SetConsoleCursorPosition(hStdOut, csbi.dwCursorPosition))
  {
    MessageBox(NULL, "Set console cursor position failed.", "Win32 API error",
      MB_OK | MB_ICONINFORMATION);
    return 0;
  }

  return 0;
}

int main(void)
{
  LPSTR  lpszPrompt = "Press ESC to exit.\n";
  CHAR  c;
  DWORD  cRead, cWritten;
  DWORD  dwOldMode, dwNewMode;

  // читаем дескрипторы стандартного ввода и вывода
  hStdIn = GetStdHandle(STD_INPUT_HANDLE);
  hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
  if (hStdIn == INVALID_HANDLE_VALUE || hStdOut == INVALID_HANDLE_VALUE)
  {
    MessageBox(NULL, "Get standard handle failed.", "Win32 API error",
      MB_OK | MB_ICONINFORMATION);
```

```
  return GetLastError();
}
// читаем режимы, установленные по умолчанию
if (!GetConsoleMode(hStdIn, &dwOldMode))
{
  MessageBox(NULL, "Get console mode failed.", "Win32 API error",
    MB_OK | MB_ICONINFORMATION);
  return GetLastError();
}
// отключаем режиы ENABLE_LINE_INPUT и ENABLE_ECHO_INPUT
dwNewMode=dwOldMode & ~(ENABLE_LINE_INPUT | ENABLE_ECHO_INPUT);
// устанавливаем новый режим
if (!SetConsoleMode(hStdIn, dwNewMode))
{
  MessageBox(NULL, "Set console mode failed.", "Win32 API error",
    MB_OK | MB_ICONINFORMATION);
  return GetLastError();
}
// выводим сообщение о том, как выйти из цикла чтения
if (!WriteFile(
    hStdOut,      // дескриптор стандартного вывода
    lpszPrompt,    // строка, которую выводим
    lstrlen(lpszPrompt),  // длина строки
    &cWritten,    // количество записанных байт
    NULL))      // синхронный вывод
{
  MessageBox(NULL, "Write file failed.", "Win32 API error",
    MB_OK | MB_ICONINFORMATION);
  return GetLastError();
}
// цикл чтения
for ( ; ; )
{
  // читаем следующий символ
  if (!ReadFile(hStdIn, &c, 1, &cRead, NULL))
  {
    MessageBox(NULL, «Write file failed.», «Win32 API error»,
      MB_OK | MB_ICONINFORMATION);
    return GetLastError();
```

```
    }
    // выбор действия
    switch (c)
    {
    // переход на новую строку
    case '\r':
      if (!GoToNewLine())
        {
          MessageBox(NULL, «Go to a new line failed.», «Win32 API error»,
            MB_OK | MB_ICONINFORMATION);
          return GetLastError();
        }
      break;
    // выход из программы
    case '\033':
      return 1;
    // распечатываем введенный символ
    default:
      if (!WriteFile(hStdOut, &c, cRead, &cWritten, NULL))
      {
        MessageBox(NULL, «Write file failed.», «Win32 API error»,
          MB_OK | MB_ICONINFORMATION);
        return GetLastError();
      }
    }
  }

  return 0;
}
```

Часть IV

Обмен данными между параллельными процессами

Глава 14

Передача данных

14.1. Способы передачи данных между процессами

Под *обменом данными между параллельными процессами* понимается пересылка данных от одного потока к другому потоку, предполагая, что эти потоки выполняются в контекстах разных процессов. Поток, который посылает данные другому потоку, называется *отправителем*. Поток, который получает данные от другого потока, называется *адресатом* или *получателем*.

Если потоки выполняются в одном процессе, то для обмена данными между ними можно использовать глобальные переменные и средства синхронизации потоков. Дело обстоит сложнее в том случае, если потоки выполняются в разных процессах — потоки не могут обращаться к общим переменным и для обмена данными между ними существуют специальные средства операционной системы. Если говорить концептуально, то для обмена данными между процессами создается *канал передачи данных*, организация которого схематически показана на рис. 14.1.

Кратко поясним устройство и работу канала передачи данных. Канал данных включает входной и выходной буферы памяти, потоки ядра операционной системы и общую память, доступ к которой имеют оба потока ядра. Работает канал передачи данных следующим образом:

- первый поток ядра операционной системы читает данные из входного буфера B_1 и записывает их в общую память M;
- второй поток ядра читает данные из общей памяти M и записывает их в буфер B_2.

Пользовательские потоки T_1 и T_2 посредством вызова функций ядра операционной системы имеют доступ к буферам B_1 и B_2, соответственно. Поэтому пересылка данных из потока T_1 в поток T_2 происходит следующим образом:

- пользовательский поток T_1 записывает данные в буфер B_1, используя специальную функцию ядра операционной системы;

- поток K_1 ядра операционной системы читает данные из буфера B_1 и записывает их в общую память M;
- поток K_2 ядра операционной системы читает данные из общей памяти M и записывает их в буфер B_2;
- пользовательский поток T_2 читает данные из буфера B_2.

Рис. 14.1. Схема канала передачи данных

Отсюда видно, что в любом случае обмен данными может быть организован только через цепочку взаимодействующих потоков, которые обмениваются между собой данными через общую, только для них, память.

На схему канала, показанную на рис. 14.1, можно взглянуть и шире. Она также подходит для организации канала передачи данных по сети. Только в этом случае общая память *M*, может рассматриваться как передающая среда, устройство которой аналогично устройству канала передачи данных.

Теперь скажем о том, как организовать канал передачи данных между процессами программным образом. Для этого нужно потоки ядра операционной системы и общую память, используемую для обмена данными, заменить файлом. И мы получим простейший канал передачи данных между потоками, выполняющимися в контекстах разных процессов. Таким образом, обмен данными между процессами через общий файл представляет собой организацию простейшего канала передачи данных между процессами. Иногда операционная система может упростить доступ к разделяемому файлу, чтобы ускорить обмен данными. Например, операционные системы Windows для этих целей могут отображать или, другими словами, проецировать файл на адресное пространство процесса.

При обмене данными между параллельными процессами различают два способа передачи данных:

- потоком;
- сообщением.

Если данные передаются непрерывной последовательностью байтов, то такая пересылка данных называется *передача данных потоком*. В этом случае общая память M, доступная потокам ядра операционной системы, может и отсутствовать, а пересылка данных выполняется одним потоком ядра непосредственно из буфера B_1 в буфер B_2.

Если данные пересылаются группами байтов, то такая группа байтов называется *сообщением*, а сама пересылка данных называется *передачей данных сообщениями*.

14.2. Связи между процессами

Прежде чем передавать данные между процессами, нужно установить между этими процессами связь. Связь между процессами может устанавливаться как на физическом (или аппаратном), так и на логическом (или программном) уровнях. С точки зрения направления передачи данных различают следующие виды связей:

- *полудуплексная связь*, т. е. данные по этой связи могут передаваться только в одном направлении;
- *дуплексная связь*, т. е. данные по этой связи могут передаваться в обоих направлениях.

Теперь, предполагая, что рассматриваются только полудуплексные связи, определим возможные топологии связей. Под *топологией связи* будем понимать конфигурацию связей между процессами-отправителями и адресатами. С точки зрения топологии различают следующие виды связей:

- $1 \to 1$ — между собой связаны только два процесса;
- $1 \to N$ — один процесс связан с N процессами;
- $N \to 1$ — каждый из N процессов связан с одним процессом;
- $N \to M$ — каждый из N процессов связан с каждым из M процессов.

Эти виды связей схематически показаны на рис. 14.2, где процессы изображены кругами, связи дугами, а прямоугольники обозначают средства передачи данных. Здесь нужно было бы сказать не средства, а каналы передачи данных, но для названия каналов передачи данных, которые обеспечивают различные виды связей, используются специальные термины. Причем эти термины могут отличаться в различных системах и, кроме того, часто один канал передачи данных может поддерживать различные виды связей между процессами.

При разработке систем с обменом данными между процессами, прежде всего, должна быть выбрана топология связей и направления передачи данных по этим связям. После этого в программах реализуются выбранные связи между процессами с использованием специальных функций операционной сис-

темы, которые предназначены для установки связи между процессами. Эти функции, а также функции для обмена данными между процессами обеспечивает система передачи данных, которая обычно является частью ядра операционной системы.

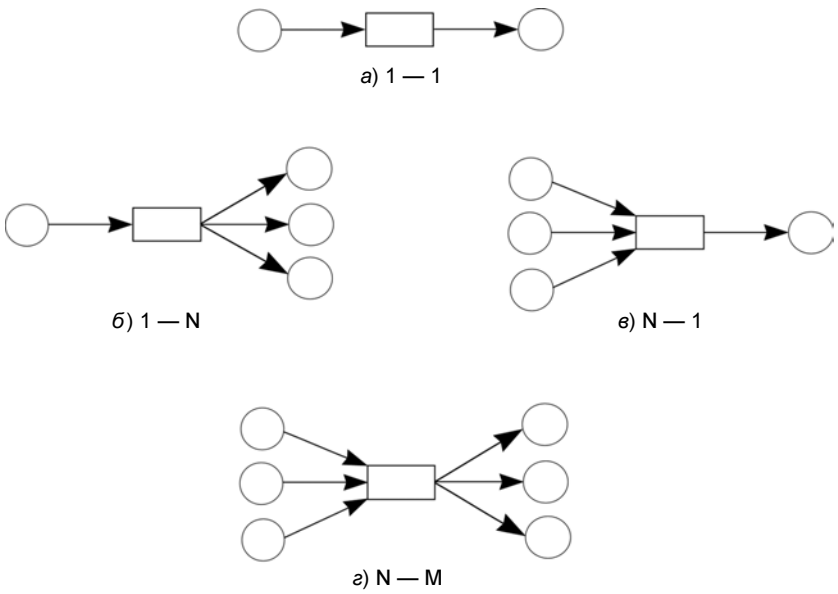


Рис. 14.2. Топология связей между процессами

14.3. Передача сообщений

Концептуально обмен сообщениями между процессами выполняется при помощи двух функций:

- `send` — послать сообщение;
- `receive` — получить сообщение.

Само сообщение состоит из двух частей: заголовка и тела сообщения. В заголовке сообщения находится такая служебная информация, как:

- тип сообщения;
- имя адресата сообщения;
- имя отправителя сообщения;
- длина сообщения (контрольная информация).

Тело сообщения содержит само сообщение.

При передаче сообщений может использоваться прямая или косвенная адресация процессов. При *прямой адресации* процессов в функциях `send` и `receive` явно указываются процессы отправитель и адресат. В этом случае функции обмена данными имеют следующий вид:

```
send(Process P, сообщение);      // послать сообщение процессу P
receive(Process Q, сообщение);   // получить сообщение от процесса Q
```

При *косвенной адресации* в функциях `send` и `receive` указываются не адреса, а имя связи, по которой передается сообщение. В этом случае функции обмена данными имеют следующий вид:

```
send(Connection S, сообщение);      // послать сообщение по связи S
receive(Connection R, сообщение);   // получить сообщение по связи R
```

Адресация процессов может быть симметричной и асимметричной. Если при обмене сообщениями между процессами используется только прямая или только косвенная адресация, то такая адресация процессов называется *симметричной*. Если же при обмене сообщениями между процессами используется как прямая, так и косвенная адресация, то такая адресация процессов называется *асимметричной*. Асимметричная адресация процессов используется в системах "клиент-сервер". В этом случае клиенты знают адрес сервера и посылают ему сообщения, используя функцию:

```
send(Process Сервер, сообщение);
```

Сервер "слушает" канал связи и принимает сообщения от всех клиентов, используя функцию:

```
receive(Connection S, сообщение);
```

Часто эта функция так и называется `listen` (слушать).

В заключение этого раздела дадим важное определение. Набор правил, по которым устанавливаются связи и передаются данные между процессами, называется *протоколом*. Можно немного пояснить, в чем заключаются правила протокола с точки зрения программирования передачи данных. Эти правила включают набор процедур или функций, которые используются для передачи данных, и порядок использования этих функций. То есть при использовании функций нужно учитывать определенную последовательность их вызовов. Например, нельзя начинать передачу данных, не установив связь между отправителем и адресатом.

14.4. Синхронный и асинхронный обмен данными

При передаче данных различают синхронный и асинхронный обмен данными. Если поток-отправитель, отправив сообщение функцией `send`, блокируется до получения этого сообщения потоком-адресатом, то такое отправление

сообщения называется *синхронным*. В противном случае отправление сообщения называется *асинхронным*. Если поток-адресат, вызвавший функцию `receive`, блокируется до тех пор, пока не получит сообщение, то такое получение сообщения называется *синхронным*. В противном случае получе-ние сообщения называется *асинхронным*.

Обмен сообщениями называется *синхронным*, если поток-отправитель синхронно передает сообщения, а поток-адресат синхронно принимает эти сообщения. В противном случае обмен сообщениями называется *асинхронным*. Предполагается, что обмен данными потоком всегда происходит синхронным образом, т. к. в этом случае между отправителем и адресатом устанавливается непосредственная связь.

Синхронный обмен данными в случае прямой адресации процессов называется *рандеву* (*rendezvous*, франц. "встреча"). Такой механизм обмена сообщениями используется в языке программирования Ада.

Учитывая возможности синхронного обмена данными, этот механизм может использоваться для решения задач синхронизации процессов. При этом следует отметить, что в операционных системах Windows синхронизация процессов, выполняющихся на разных компьютерах в локальной сети, может осуществляться только при помощи механизма синхронного обмена данными.

14.5. Буферизация

Буфером называется *вместимость связи между процессами*, т. е. количество сообщений, которые могут одновременно пересылаться по этой связи. Различаются три типа буферизации:

- нулевая вместимость связи (нет буфера) — в этом случае возможен только синхронный обмен данными между процессами;
- ограниченная вместимость связи (ограниченный буфер) — в этом случае, если буфер полон, то отправитель сообщения должен ждать очистки буфера хотя бы от одного сообщения;
- неограниченная вместимость связи (неограниченный буфер) — в этом случае отправитель никогда не ждет при отправке сообщения.

Как видно из этих определений, типы буферизации тесно связаны с синхронизацией передачи данных и поэтому также должны учитываться при разработке систем, которые содержат обмен данными между процессами.

Глава 15

Работа с анонимными каналами в Windows

Так как работа с анонимными каналами требует совместного использования целого ряда функций, то данная глава организована следующим образом. Сначала дано определение анонимных каналов и рассмотрены все функции, которые предназначены для работы с ними. Потом приведены примеры, иллюстрирующие использование этих функций. В последнем разделе показано, как при помощи анонимных каналов можно перенаправить стандартный ввод-вывод.

15.1. Анонимные каналы

Анонимным каналом называется объект ядра операционной системы, который обеспечивает передачу данных между процессами, выполняющимися на одном компьютере. Процесс, который создает анонимный канал, называется *сервером анонимного канала*. Процессы, которые связываются с анонимным каналом, называются *клиентами анонимного канала*. Другими словами можно сказать, что анонимный — это такой канал передачи данных между процессами, который не имеет имени. Следовательно, доступ к такому каналу имеют только родительский процесс-сервер и дочерние процессы-клиенты этого канала.

Перечислим характеристики анонимных каналов, которые необходимо учитывать при их использовании для обмена данными между параллельными процессами:

- ❒ не имеют имени;
- ❒ полудуплексные;
- ❒ передача данных потоком;
- ❒ синхронный обмен данными;
- ❒ возможность моделирования любой топологии связей.

Немного поясним характеристику направления передачи данных по анонимному каналу. Так как анонимный канал полудуплексный, то может создаться впечатление, что по этому каналу можно передавать данные от процесса к процессу только в одном направлении. На самом деле это не так. Да, действительно, по анонимному каналу можно передавать данные только в одном направлении. Но каждый конец анонимного канала имеет свой дескриптор, который можно передать любому дочернему процессу. Поэтому каждый процесс может как записывать данные в анонимный канал, так и читать данные оттуда.

Замечание

В операционных системах Windows NT анонимный канал реализован при помощи именованного канала с уникальным именем. Отсюда следует, что в этих операционных системах функции, которые работают с именованными каналами, могут работать также и с анонимными каналами.

Теперь приведем порядок работы с анонимными каналами, которого и будем придерживаться в дальнейшем:

- создание анонимного канала сервером;
- соединение клиентов с каналом;
- обмен данными по каналу;
- закрытие канала.

Подробно эти пункты работы с анонимными каналами будут рассмотрены в следующих разделах этой главы.

15.2. Создание анонимных каналов

Анонимные каналы создаются процессом-сервером при помощи функции `CreatePipe`, которая имеет следующий прототип:

```
BOOL  CreatePipe(
  PHANDLE  hReadHandle,      // дескриптор для чтения из канала
  PHANDLE  hWriteHandle,     // дескриптор для записи в канал
  LPSECURITY_ATTRIBUTES  lpPipeAttributes,   // атрибуты защиты
  DWORD    dwSize            // размер буфера в байтах
);
```

При удачном завершении функция `CreatePipe` возвращает ненулевое значение, а в случае неудачи — `FALSE`. Рассмотрим кратко назначение параметров этой функции.

В случае успешного завершения функция `CreatePipe` создает два дескриптора анонимного канала — один для чтения данных из канала, а второй для записи данных в канал. Эти дескрипторы возвращаются в переменных,

на которые указывают параметры hReadPipe и hWritePipe. Дескриптор, на который указывает параметр hReadPipe, в дальнейшем используется в функциях чтения данных из канала, а дескриптор hWritePipe — в функциях записи данных в канал.

Параметр lpPipeAttributes определяет атрибуты защиты анонимного канала. Установку значения этого параметра lpPipeAttributes мы рассмотрим в *разд. 15.3*.

Параметр dwSize определяет размер буфера ввода-вывода анонимного канала. Отметим, что операционные системы Windows автоматически определяют размер буфера и поэтому значение параметра dwSize является только пожеланием операционной системе при выборе размера буфера. Значение этого параметра можно установить равным 0, тогда операционная система выберет размер буфера по умолчанию.

15.3. Соединение клиентов с анонимным каналом

Так как анонимные каналы не имеют имени, то для соединения процесса-клиента с таким каналом необходимо передать ему один из дескрипторов анонимного канала. При этом передаваемый дескриптор должен быть наследуемым, а сам процесс-клиент должен быть дочерним процессом процесса сервера анонимного канала и наследовать наследуемые дескрипторы процесса-сервера.

Наследование дескрипторов анонимного канала определяется значением поля bInheritHandle в структуре типа SECURITY_ATTRIBUTES, на которую указывает параметр lpPipeAttributes функции CreatePipe. Если значение этого поля, которое имеет тип BOOL, равно TRUE, то дескрипторы анонимного канала создаются наследуемыми, в противном случае — дескрипторы создаются ненаследуемыми. Если процессу-клиенту передаются два дескриптора анонимного канала, то естественно, что оба они должны быть сделаны наследуемыми при создании анонимного канала процессом-сервером. Если же процессу-клиенту передается только один из дескрипторов, то возможны два варианта действий. Либо создать наследуемые дескрипторы анонимного канала, а затем тот дескриптор, который не передается клиенту, сделать ненаследуемым. Либо, наоборот, создать ненаследуемые дескрипторы анонимного канала, а затем тот дескриптор, который передается клиенту, сделать наследуемым. В операционной системе Windows 98 обе эти задачи решаются путем создания соответственно ненаследуемого или наследуемого дубликата исходного дескриптора, используя функцию DuplicateHandle. После этого исходный дескриптор закрывается. В операционной системе Windows 2000 эта задача также может быть решена с помощью функции SetHandleInformation, которая изменяет свойство наследования дескриптора.

Для того чтобы процесс-клиент наследовал дескрипторы анонимного канала он должен быть создан функцией `CreateProcess` в процессе-сервере анонимного канала и параметр `bInheritHandles` этой функции должен быть установлен в `TRUE`.

Явная передача наследуемого дескриптора процессу-клиенту анонимного канала может выполняться одним из следующих способов:

- через командную строку;
- через поля `hStdInput`, `hStdOutput` и `hStdError` структуры `STARTUPINFO`;
- посредством сообщения `WM_COPYDATA`;
- через файл.

В данной главе мы будем использовать только первые два способа передачи дескрипторов процессу-клиенту, что наиболее естественно для анонимных каналов.

В завершение этого раздела скажем, что более подробно вопросы наследования и дублирования дескрипторов были рассмотрены в *гл. 4*.

15.4. Обмен данными по анонимному каналу

Для обмена данными по анонимному каналу в операционных системах Windows используются те же функции, что для записи/чтения данных в файл.

Для записи данных в анонимный канал используется функция `WriteFile`, которая имеет следующий прототип:

```
BOOL  WriteFile(
  HANDLE   hAnonymousPipe,    // дескриптор анонимного канала
  LPCVOID  lpBuffer,          // буфер данных
  DWORD    dwNumberOfBytesToWrite,    // количество байтов для записи
  LPDWORD  lpNumberOfBytesWritten,    // количество записанных байтов
  LPOVERLAPPED  lpOverlapped          // асинхронный ввод
);
```

Функция `WriteFile` записывает в анонимный канал количество байтов, заданных параметром `dwNumberOfBytesToWrite`, из буфера данных, на который указывает параметр `lpBuffer`. Дескриптор вывода этого анонимного канала должен быть задан первым параметром функции `WriteFile`. При успешном завершении функция `WriteFile` возвращает ненулевое значение, а в случае неудачи — `FALSE`. Количество байт, записанных функцией `WriteFile` в анонимный канал, возвращается в переменной, на которую указывает параметр `lpNumberOfBytesWritten`. Параметр `lpOverlapped` предназначен для выполнения асинхронной операции вывода, т. к. анонимные каналы поддерживают только синхронную передачу данных, то в нашем случае этот параметр всегда будет равен `NULL`.

Для чтения данных из анонимного канала используется функция ReadFile, которая имеет следующий прототип:

```
BOOL  ReadFile(
  HANDLE   hAnonymousPipe,   // дескриптор анонимного канала
  LPCVOID  lpBuffer,         // буфер данных
  DWORD    dwNumberOfBytesToRead,  // количество байт для записи
  LPDWORD  lpNumberOfBytesRead,    // количество записанных байтов
  LPOVERLAPPED  lpOverlapped       // асинхронный ввод
);
```

Функция ReadFile читает из анонимного канала количество байт, заданных параметром dwNumberOfBytesToRead, в буфер данных, на который указывает параметр lpBuffer. Дескриптор ввода этого анонимного канала должен быть задан первым параметром функции ReadFile. При успешном завершении функция ReadFile возвращает ненулевое значение, а в случае неудачи — FALSE. Количество байт, прочитанных функцией WriteFile из анонимного канала, возвращается в переменной, на которую указывает параметр lpNumberOfBytesRead. Также как и в случае записи в анонимный канал параметр lpOverlapped должен быть равен NULL.

Отметим, что обмен данными по анонимному каналу осуществляется только в соответствии с назначением дескриптора этого канала. Дескриптор для записи в анонимный канал должен быть параметром функции WriteFile, а дескриптор для чтения из анонимного канала должен быть параметром функции ReadFile. В этом и состоит смысл передачи данных по анонимному каналу только в одном направлении. Однако это не означает, что один процесс может использовать анонимный канал только для записи или только для чтения. Один и тот же процесс может как писать в анонимный канал, так и читать данные из него, должным образом используя дескрипторы этого канала.

В завершение этого раздела отметим, что после завершения обмена данными по анонимному каналу потоки должны закрыть дескрипторы записи и чтения анонимного канала, используя функцию CloseHandle.

15.5. Примеры работы с анонимными каналами

Сначала рассмотрим пример, в котором процесс-сервер создает анонимный канал и дочерний процесс, которому передает один из дескрипторов этого анонимного канала. Для передачи дескриптора используется командная строка и, в этом случае, дочерний процесс является клиентом анонимного канала. Для определенности передадим клиенту дескриптор для записи в анонимный канал и оставим серверу дескриптор для чтения.

В листинге 15.1 приведена программа процесса-клиента анонимного канала.

Листинг 15.1. Программа процесса-клиента анонимного канала

```
#include <windows.h>
#include <conio.h>

int main(int argc, char *argv[])
{
  HANDLE hWritePipe;

  // преобразуем символьное представление дескриптора в число
  hWritePipe = (HANDLE)atoi(argv[1]);
  // ждем команды о начале записи в анонимный канал
  _cputs("Press any key to start communication.\n");
  _getch();
  // пишем в анонимный канал
  for (int i = 0; i < 10; i++)
  {
    DWORD dwBytesWritten;
    if (!WriteFile(
        hWritePipe,
        &i,
        sizeof(i),
        &dwBytesWritten,
        NULL))
      {
        _cputs("Write to file failed.\n");
        _cputs("Press any key to finish.\n");
        _getch();
        return GetLastError();
      }
    _cprintf("The number %d is written to the pipe.\n", i);
    Sleep(500);
  }
  // закрываем дескриптор канала
  CloseHandle(hWritePipe);

  _cputs("The process finished writing to the pipe.\n");
```

```
  _cputs("Press any key to exit.\n");
  _getch();

  return 0;
}
```

В листинге 15.2 приведена программа процесса-сервера анонимного канала, который запускает клиента и передает ему дескриптор записи в анонимный канал через командную строку.

Листинг 15.2. Программа процесса-сервера анонимного канала

```
#include <windows.h>
#include <conio.h>

int main()
{
  char lpszComLine[80];  // для командной строки

  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  HANDLE hWritePipe, hReadPipe, hInheritWritePipe;

  // создаем анонимный канал
  if(!CreatePipe(
      &hReadPipe,    // дескриптор для чтения
      &hWritePipe,   // дескриптор для записи
      NULL,          // атрибуты защиты по умолчанию, в этом случае
                     // дескрипторы hReadPipe и hWritePipe ненаследуемые
      0))            // размер буфера по умолчанию

  {
    _cputs("Create pipe failed.\n");
    _cputs("Press any key to finish.\n");
    _getch();
    return GetLastError();
  }
  // делаем наследуемый дубликат дескриптора hWritePipe
  if(!DuplicateHandle(
      GetCurrentProcess(),   // дескриптор текущего процесса
```

```
    hWritePipe,            // исходный дескриптор канала
    GetCurrentProcess(),   // дескриптор текущего процесса
    &hInheritWritePipe,    // новый дескриптор канала
    0,                     // этот параметр игнорируется
    TRUE,                  // новый дескриптор наследуемый
    DUPLICATE_SAME_ACCESS ))  // доступ не изменяем
{
  _cputs("Duplicate handle failed.\n");
  _cputs("Press any key to finish.\n");
  _getch();
  return GetLastError();
}
// закрываем ненужный дескриптор
CloseHandle(hWritePipe);
// устанавливаем атрибуты нового процесса
ZeroMemory(&si, sizeof(STARTUPINFO));
si.cb = sizeof(STARTUPINFO);
// формируем командную строку
wsprintf(lpszComLine, "C:\\Client.exe %d", (int)hInheritWritePipe);
// запускаем новый консольный процесс
if (!CreateProcess(
    NULL,    // имя процесса
    lpszComLine,   // командная строка
    NULL,    // атрибуты защиты процесса по умолчанию
    NULL,    // атрибуты защиты первичного потока по умолчанию
    TRUE,    // наследуемые дескрипторы текущего процесса
             // наследуются новым процессом
    CREATE_NEW_CONSOLE,  // новая консоль
    NULL,    // используем среду окружения процесса-предка
    NULL,    // текущий диск и каталог, как и в процессе-предке
    &si,     // вид главного окна - по умолчанию
    &pi      // здесь будут дескрипторы и идентификаторы
             // нового процесса и его первичного потока
    ))
  {
    _cputs("Create process failed.\n");
    _cputs("Press any key to finish.\n");
    _getch();
    return GetLastError();
```

```
    }
  // закрываем дескрипторы нового процесса
  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);
  // закрываем ненужный дескриптор канала
  CloseHandle(hInheritWritePipe);
  // читаем из анонимного канала
  for (int i = 0; i < 10; i++)
  {
    int nData;
    DWORD dwBytesRead;
    if (!ReadFile(
        hReadPipe,
        &nData,
        sizeof(nData),
        &dwBytesRead,
        NULL))
      {
        _cputs("Read from the pipe failed.\n");
        _cputs("Press any key to finish.\n");
        _getch();
        return GetLastError();
      }
    _cprintf("The number %d is read from the pipe.\n", nData);
  }
  // закрываем дескриптор канала
  CloseHandle(hReadPipe);

  _cputs("The process finished reading from the pipe.\n");
  _cputs("Press any key to exit.\n");
  _getch();

  return 0;
}
```

В программах, приведенных в листингах 15.3 и 15.4, показывается, как организовать двусторонний обмен данными по анонимному каналу между клиентом и сервером. Для этого дескрипторы чтения и записи анонимного канала используются как сервером, так и клиентом этого анонимного канала.

В этих листингах сначала приведена программа процесса-клиента анонимного канала, а затем программа процесса-сервера анонимного канала. Как и в листингах 15.1, 15.2, дескрипторы анонимного канала передаются через командную строку.

Листинг 15.3. Процесс-клиент анонимного канала

```
#include <windows.h>
#include <conio.h>

int main(int argc, char *argv[])
{
  HANDLE hWritePipe, hReadPipe;
  HANDLE hEnableRead;    // для синхронизации обмена данными
  char lpszEnableRead[] = "EnableRead";

  // открываем событие, разрешающее чтение
  hEnableRead = OpenEvent(EVENT_ALL_ACCESS, FALSE, lpszEnableRead);

  // преобразуем символьное представление дескрипторов в число
  hWritePipe = (HANDLE)atoi(argv[1]);
  hReadPipe = (HANDLE)atoi(argv[2]);
  // ждем команды о начале записи в анонимный канал
  _cputs("Press any key to start communication.\n");
  _getch();
  // пишем в анонимный канал
  for (int i = 0; i < 10; i++)
  {
    DWORD dwBytesWritten;
    if (!WriteFile(
        hWritePipe,
        &i,
        sizeof(i),
        &dwBytesWritten,
        NULL))
      {
        _cputs("Write to file failed.\n");
        _cputs("Press any key to finish.\n");
        _getch();
```

```
        return GetLastError();
      }
    _cprintf("The number %d is written to the pipe.\n", i);
  }
  _cputs("The process finished writing to the pipe.\n");

  // ждем разрешения на чтение
  WaitForSingleObject(hEnableRead, INFINITE);
  // читаем ответ из анонимного канала
  for (int j = 0; j < 10; j++)
  {
    int nData;
    DWORD dwBytesRead;
    if (!ReadFile(
        hReadPipe,
        &nData,
        sizeof(nData),
        &dwBytesRead,
        NULL))
      {
        _cputs("Read from the pipe failed.\n");
        _cputs("Press any key to finish.\n");
        _getch();
        return GetLastError();
      }
    _cprintf("The number %d is read from the pipe.\n", nData);
  }
  _cputs("The process finished reading from the pipe.\n");
  _cputs("Press any key to exit.\n");
  _getch();

  // закрываем дескрипторы канала
  CloseHandle(hWritePipe);
  CloseHandle(hReadPipe);
  CloseHandle(hEnableRead);

  return 0;
}
```

В листинге 15.4 приведена программа процесса-сервера анонимного канала, который запускает клиента и передает ему дескрипторы анонимного канала через командную строку.

Листинг 15.4. Программа процесса-сервера анонимного канала

```
#include <windows.h>
#include <conio.h>

int main()
{
  char lpszComLine[80];  // для командной строки

  HANDLE hEnableRead;    // для синхронизации обмена данными
  char lpszEnableRead[] = "EnableRead";

  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  HANDLE hWritePipe, hReadPipe;
  SECURITY_ATTRIBUTES sa;

  // создаем событие для синхронизации обмена данными
  hEnableRead = CreateEvent(NULL, FALSE, FALSE, lpszEnableRead);

  // устанавливает атрибуты защиты канала
  sa.nLength = sizeof(SECURITY_ATTRIBUTES);
  sa.lpSecurityDescriptor = NULL;     // защита по умолчанию
  sa.bInheritHandle = TRUE;           // дескрипторы наследуемые

  // создаем анонимный канал
  if(!CreatePipe(
      &hReadPipe,  // дескриптор для чтения
      &hWritePipe, // дескриптор для записи
      &sa,   // атрибуты защиты по умолчанию, дескрипторы наследуемые
      0))    // размер буфера по умолчанию

    {
      _cputs("Create pipe failed.\n");
      _cputs("Press any key to finish.\n");
```

```
    _getch();
    return GetLastError();
  }
// устанавливаем атрибуты нового процесса
ZeroMemory(&si, sizeof(STARTUPINFO));
si.cb = sizeof(STARTUPINFO);
// формируем командную строку
wsprintf(lpszComLine, "C:\\Client.exe %d %d",
        (int)hWritePipe, (int)hReadPipe);
// запускаем новый консольный процесс
if (!CreateProcess(
    NULL,    // имя процесса
    lpszComLine,   // командная строка
    NULL,    // атрибуты защиты процесса по умолчанию
    NULL,    // атрибуты защиты первичного потока по умолчанию
    TRUE,    // наследуемые дескрипторы текущего процесса
             // наследуются новым процессом
    CREATE_NEW_CONSOLE,  // новая консоль
    NULL,    // используем среду окружения процесса-предка
    NULL,    // текущий диск и каталог, как и в процессе-предке
    &si,     // вид главного окна - по умолчанию
    &pi      // здесь будут дескрипторы и идентификаторы
             // нового процесса и его первичного потока
    ))
  {
    _cputs("Create process failed.\n");
    _cputs("Press any key to finish.\n");
    _getch();
    return GetLastError();
  }
// закрываем дескрипторы нового процесса
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
// читаем из анонимного канала
for (int i = 0; i < 10; i++)
{
  int nData;
  DWORD dwBytesRead;
  if (!ReadFile(
```

```
      hReadPipe,
      &nData,
      sizeof(nData),
      &dwBytesRead,
      NULL))
    {
      _cputs("Read from the pipe failed.\n");
      _cputs("Press any key to finish.\n");
      _getch();
      return GetLastError();
    }
  _cprintf("The number %d is read from the pipe.\n", nData);
}
_cputs("The process finished reading from the pipe.\n");

// даем сигнал на разрешение чтения клиентом
SetEvent(hEnableRead);

// пишем ответ в анонимный канал
for (int j = 10; j < 20; j++)
{
  DWORD dwBytesWritten;
  if (!WriteFile(
      hWritePipe,
      &j,
      sizeof(j),
      &dwBytesWritten,
      NULL))
    {
      _cputs("Write to file failed.\n");
      _cputs("Press any key to finish.\n");
      _getch();
      return GetLastError();
    }
  _cprintf("The number %d is written to the pipe.\n", j);
}
// закрываем дескрипторы канала
CloseHandle(hReadPipe);
CloseHandle(hWritePipe);
```

```
  CloseHandle(hEnableRead);

  _cputs("The process finished writing to the pipe.\n");
  _cputs("Press any key to exit.\n");
  _getch();

  return 0;
}
```

Отметим в листинге 15.4 следующий момент: для организации двустороннего обмена данными по анонимному каналу сервер и клиенты канала должны синхронизировать доступ к этому каналу. То есть для организации передачи данных необходимо разработать протокол передачи данных или использовать объекты синхронизации, которые исключают одновременный неконтролируемый доступ параллельных потоков к анонимному каналу. В приведенном листинге событие `hEnableRead` сигнализирует клиенту о том, что сервер закончил чтение данных и теперь данные из канала может читать клиент. При отсутствии такой синхронизации возможно одновременное чтение данных сервером и клиентом, т. к. они работают параллельно, и это вызовет неправильную работу программы и ее зависание.

15.6. Перенаправление стандартного ввода-вывода

Анонимные каналы часто используются для перенаправления стандартного ввода-вывода. Чтобы подробнее разобраться с этим вопросом, сначала кратко рассмотрим стандартные средства ввода-вывода, используемые в языке программирования C++. Компилятор языка программирования C++ фирмы Microsoft содержит стандартную библиотеку, которая поддерживает три варианта функций стандартного ввода-вывода. Эти функции описываются в заголовочных файлах stdio.h, iostream.h и conio.h.

Функции, которые описаны в файле stdio.h, обеспечивают ввод-вывод в следующие стандартные файлы:

- `stdin` — стандартный файл ввода;
- `stdout` — стандартный файл вывода;
- `stderr` — файл вывода сообщений об ошибках.

Эти функции составляют стандартную библиотеку ввода-вывода языка программирования C.

Функции и операторы, которые описаны в заголовке iostream.h, обеспечивают ввод-вывод в стандартные потоки. Эти функции составляют стандарт-

ную библиотеку ввода-вывода в языке программирования C++. При создании консольного процесса или при распределении консоли с помощью функции `AllocConsole` стандартные файлы и стандартные потоки ввода-вывода связываются с дескрипторами, которые заданы в полях `hStdInput`, `hStdOutput` и `hStdError` структуры типа `STARTUPINFO`. Поэтому если в эти поля будут записаны соответствующие дескрипторы анонимного канала, то для передачи данных по анонимному каналу можно использовать функции стандартного ввода-вывода. Такая процедура называется *перенаправлением стандартного ввода-вывода*.

Функции ввода-вывода из заголовочного файла conio.h отличаются от функций ввода-вывода из заголовочного файла stdio.h стандартной библиотеки языка программирования C тем, что они всегда работают с консолью. Поэтому эти функции можно использовать для ввода-вывода на консоль даже в случае перенаправления стандартного ввода-вывода.

В листингах 15.5, 15.6, 15.7 приведены программы, в которых стандартный ввод-вывод перенаправляется в анонимный канал, а для обмена данными по анонимному каналу используются перегруженные операторы ввода-вывода языка программирования C++. Пример включает программы следующих процессов: два процесса-клиента, которые обмениваются данными по анонимному каналу, и процесс-сервер, который создает клиентов и передает им дескрипторы анонимного канала через поля структуры `STARTUPINFO`. В листингах 15.5, 15.6 приведены программы процессов-клиентов.

Листинг 15.5. Процесс-клиент, записывающий данные в анонимный канал

```
#include <windows.h>
#include <conio.h>
#include <iostream.h>

int main()
{
  // события для синхронизации обмена данными
  HANDLE hReadFloat, hReadText;
  char lpszReadFloat[] = "ReadFloat";
  char lpszReadText[] = "ReadText";

  // открываем события
  hReadFloat = CreateEvent(NULL, FALSE, FALSE, lpszReadFloat);
  hReadText = CreateEvent(NULL, FALSE, FALSE, lpszReadText);

  // ждем команды о начале записи в анонимный канал
```

```
  _cputs("Press any key to start communication.\n");
  _getch();
  // пишем целые числа в анонимный канал
    for (int i = 0; i < 5; ++i)
  {
    Sleep(500);
    cout << i << endl;
  }

  // ждем разрешение на чтение плавающих чисел из канала
  WaitForSingleObject(hReadFloat, INFINITE);
  // читаем плавающие числа из анонимного канала
    for (int j = 0; j < 5; ++j)
  {
    float nData;
    cin >> nData;
    _cprintf("The number %2.1f is read from the pipe.\n", nData);
    }

  // отмечаем, что можно читать текст из анонимного канала
  SetEvent(hReadText);
  // теперь передаем текст
  cout << "This is a demo sentence." << endl;
  // отмечаем конец передачи
  cout << '\0' << endl;

  _cputs("The process finished transmission of data.\n");
  _cputs("Press any key to exit.\n");
  _getch();

  CloseHandle(hReadFloat);
  CloseHandle(hReadText);

  return 0;
}
```

Листинг 15.6. Процесс-клиент, читающий данные из анонимного канала

```
#include <windows.h>
#include <conio.h>
#include <iostream.h>

int main()
{
  // события для синхронизации обмена данными
  HANDLE hReadFloat, hReadText;
  char lpszReadFloat[] = "ReadFloat";
  char lpszReadText[] = "ReadText";

  // открываем события
  hReadFloat = CreateEvent(NULL, FALSE, FALSE, lpszReadFloat);
  hReadText = CreateEvent(NULL, FALSE, FALSE, lpszReadText);

  // читаем целые числа из анонимного канала
  for (int i = 0; i < 5; ++i)
  {
    int nData;
    cin >> nData;
    _cprintf("The number %d is read from the pipe.\n", nData);
  }

  // разрешаем читать плавающие числа из анонимного канала
  SetEvent(hReadFloat);
  // пишем плавающие числа в анонимный канал
  for (int j = 0; j < 5; ++j)
  {
    Sleep(500);
    cout << (j*0.1) << endl;
  }

  // ждем разрешения на чтение текста
  WaitForSingleObject(hReadText, INFINITE);
  _cputs("The process read the text: ");
  // теперь читаем текст
  char lpszInput[80];
```

```
  do
  {
    Sleep(500);
    cin >> lpszInput;
    _cputs(lpszInput);
    _cputs(" ");
  }
  while (*lpszInput != '\0');

  _cputs("\nThe process finished transmission of data.\n");
  _cputs("Press any key to exit.\n");
  _getch();

  CloseHandle(hReadFloat);
  CloseHandle(hReadText);

  return 0;
}
```

Теперь приведем, в листинге 15.7, программу процесса-сервера анонимного канала. Эта программа просто создает двух клиентов анонимного канала и прекращает свою работу.

Листинг 15.7. Процесс-сервер анонимного канала

```
#include <windows.h>
#include <conio.h>

int main()
{
  char lpszComLine1[80] = "C:\\Client1.exe";   // имя первого клиента
  char lpszComLine2[80] = "C:\\Client2.exe";   // имя второго клиента

  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  HANDLE hWritePipe, hReadPipe;
  SECURITY_ATTRIBUTES sa;

  // устанавливает атрибуты защиты канала
  sa.nLength = sizeof(SECURITY_ATTRIBUTES);
```

```
sa.lpSecurityDescriptor = NULL;  // защита по умолчанию
sa.bInheritHandle = TRUE;        // дескрипторы наследуемые

// создаем анонимный канал
if(!CreatePipe(
    &hReadPipe,  // дескриптор для чтения
    &hWritePipe, // дескриптор для записи
    &sa,   // атрибуты защиты по умолчанию, дескрипторы наследуемые
    0))    // размер буфера по умолчанию

{
  _cputs("Create pipe failed.\n");
  _cputs("Press any key to finish.\n");
  _getch();
  return GetLastError();
}
// устанавливаем атрибуты нового процесса
ZeroMemory(&si, sizeof(STARTUPINFO));
si.cb = sizeof(STARTUPINFO);
// использовать стандартные дескрипторы
si.dwFlags = STARTF_USESTDHANDLES;
// устанавливаем стандартные дескрипторы
si.hStdInput = hReadPipe;
si.hStdOutput = hWritePipe;
si.hStdError = hWritePipe;
// запускаем первого клиента
if (!CreateProcess(
    NULL,    // имя процесса
    lpszComLine1,  // командная строка
    NULL,    // атрибуты защиты процесса по умолчанию
    NULL,    // атрибуты защиты первичного потока по умолчанию
    TRUE,    // наследуемые дескрипторы текущего процесса
             // наследуются новым процессом
    CREATE_NEW_CONSOLE,  // создаем новую консоль
    NULL,    // используем среду окружения процесса-предка
    NULL,    // текущий диск и каталог, как и в процессе-предке
```

```
    &si,      // вид главного окна - по умолчанию
    &pi       // здесь будут дескрипторы и идентификаторы
              // нового процесса и его первичного потока
    )
  )
{
  _cputs("Create process failed.\n");
  _cputs("Press any key to finish.\n");
  _getch();
  return GetLastError();
}
// закрываем дескрипторы первого клиента
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);

// запускаем второго клиента
if (!CreateProcess(
    NULL,     // имя процесса
    lpszComLine2,  // командная строка
    NULL,     // атрибуты защиты процесса по умолчанию
    NULL,     // атрибуты защиты первичного потока по умолчанию
    TRUE,     // наследуемые дескрипторы текущего процесса
        // наследуются новым процессом
    CREATE_NEW_CONSOLE,  // создаем новую консоль
    NULL,     // используем среду окружения процесса-предка
    NULL,     // текущий диск и каталог, как и в процессе-предке
    &si,      // вид главного окна - по умолчанию
    &pi       // здесь будут дескрипторы и идентификаторы
              // нового процесса и его первичного потока
    )
  )
{
  _cputs("Create process failed.\n");
  _cputs("Press any key to finish.\n");
  _getch();
  return GetLastError();
```

```
  }
  // закрываем дескрипторы второго клиента
  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);

  // закрываем дескрипторы канала
  CloseHandle(hReadPipe);
  CloseHandle(hWritePipe);

  _cputs("The clients are created.\n");
  _cputs("Press any key to exit.\n");
  _getch();

  return 0;
}
```

Глава 16

Работа с именованными каналами в Windows

Работа с именованными каналами, также как и работа с анонимными каналами, требует совместного использования целого ряда функций. Поэтому сначала будут даны характеристики именованных каналов и рассмотрены все функции, которые предназначены для передачи данных по именованным каналам. Затем будут приведены несколько примеров, которые проиллюстрируют использование этих функций. После этого будут рассмотрены служебные функции для работы с именованными каналами.

16.1. Именованные каналы

Именованным каналом называется объект ядра операционной системы, который обеспечивает передачу данных между процессами, выполняющимися на компьютерах в одной локальной сети. Процесс, который создает именованный канал, называется *сервером именованного канала*. Процессы, которые связываются с именованным каналом, называются *клиентами именованного канала*. Перечислим характеристики именованных каналов:

- имеют имя, которое используется клиентами для связи с именованным каналом;
- могут быть как полудуплексные, так и дуплексные;
- передача данных может осуществляться как потоком, так и сообщениями;
- обмен данными может быть как синхронным, так и асинхронным;
- возможность моделирования любой топологии связей.

Теперь приведем порядок работы с именованными каналами, который и будет использоваться в дальнейшем:

- создание именованного канала сервером:
- соединение сервера с экземпляром именованного канала;
- соединение клиента с экземпляром именованного канала;

- обмен данными по именованному каналу;
- отсоединение сервера от экземпляра именованного канала;
- закрытие именованного канала клиентом и сервером.

Подробно эти пункты будут рассмотрены в следующих разделах.

16.2. Создание именованных каналов

Именованные каналы создаются процессом-сервером при помощи функции `CreateNamedPipe`, которая имеет следующий прототип:

```
HANDLE  CreateNamedPipe(
  LPCTSTR  lpName,        // имя канала
  DWORD  dwOpenMode,      // атрибуты канала
  DWORD  dwPipeMode,      // режим передачи данных
  DWORD  nMaxInstances,   // максимальное количество экземпляров канала
  DWORD  nOutBufferSize,   // размер выходного буфера
  DWORD  nInBufferSize,    // размер входного буфера
  DWORD  nDefaultTimeOut,  // время ожидания связи с клиентом
  LPSECURITY_ATTRIBUTES  lpPipeAttributes    // атрибуты безопасности
);
```

При удачном завершении функция `CreateNamedPipe` возвращает дескриптор именованного канала, а в случае неудачи — одно из двух значений:

- `INVALID_HANDLE_VALUE` — неудачное завершение;
- `ERROR_INVALID_PARAMETR` — значение параметра `nMaxInstances` больше, чем величина `PIPE_UNLIMITED_INSTANCES`.

Опишем параметры этой функции.

Параметр `lpName` указывает на строку, которая должна иметь вид:

```
\\.\pipe\pipe_name
```

Здесь "." обозначает локальную машину, т. к. новый именованный канал всегда создается на локальной машине, слово `pipe` — фиксировано, а `pipe_name` обозначает имя канала, которое задается пользователем и нечувствительно к регистру.

Параметр `dwOpenMode` задает флаги, которые определяют направление передачи данных, буферизацию, синхронизацию обмена данными и права доступа к именованному каналу. Для определения направления передачи данных используются флаги:

- `PIPE_ACCESS_DUPLEX` — чтение и запись в канал;
- `PIPE_ACCESS_INBOUND` — клиент пишет, а сервер читает данные;
- `PIPE_ACCESS_OUTBOUND` — сервер пишет, а клиент читает данные.

Кроме того, каждый из этих флагов позволяет использовать дескриптор канала для синхронизации.

Флаг, определяющий направление передачи данных по именованному каналу, должен совпадать для всех экземпляров одного и того же именованного канала. Для определения способа буферизации и синхронизации используются флаги:

- `FILE_FLAG_WRITE_THROUGH` — запрещает буферизацию при передаче данных по сети;
- `FILE_FLAG_OVERLAPPED` — разрешает асинхронную передачу данных по каналу.

Эти флаги могут быть разными для каждого экземпляра одного и того же именованного канала.

В этом же параметре могут быть заданы другие режимы доступа к именованному каналу. Флаги, используемые для определения атрибутов безопасности, подробно рассмотрены в *разд. 36.7, 41.3*. Здесь же только скажем, что рассмотренные выше флаги `PIPE_ACCESS_DUPLEX`, `PIPE_ACCESS_INBOUND` и `PIPE_ACCESS_OUTBOUND` определяют специфические права доступа к именованному каналу, которые включают следующие родовые права доступа:

- `PIPE_ACCESS_DUPLEX` — включает родовые права доступа `GENERIC_READ`, `GENERIC_WRITE` и `SYNCHRONIZE`;
- `PIPE_ACCESS_INBOUND` — включает родовые права доступа `GENERIC_READ` и `SYNCHRONIZE`;
- `PIPE_ACCESS_OUTBOUND` — включает родовые права доступа `GENERIC_WRITE` и `SYNCHRONIZE`.

Подробнее о правах доступа рассказано в *разд. 36.7, 41.3*.

Параметр `dwPipeMode` задает флаги, определяющие способ передачи данных по именованному каналу. Для определения способов чтения и записи данных в именованный канал используются флаги:

- `PIPE_TYPE_BYTE` — запись данных потоком;
- `PIPE_TYPE_MESSAGE` — запись данных сообщениями;
- `PYPE_READMODE_BYTE` — чтение данных потоком;
- `PYPE_READMODE_MESSAGE` — чтение данных сообщениями.

По умолчанию данные по именованному каналу передаются потоком. Флаги, определяющие способ чтения и записи данных в именованный канал, должны совпадать для всех экземпляров одного и того же канала. Для определения синхронизации доступа к именованному каналу используются флаги:

- `PIPE_WAIT` — синхронная связь с каналом и обмен данными по каналу;
- `PIPE_NOWAIT` — асинхронная связь с каналом и обмен данными по каналу.

Эти флаги могут быть разными для каждого экземпляра именованного канала.

Параметр `nMaxInstances` определяет максимальное число экземпляров именованного канала, которое может находиться в пределах от 1 до `PIPE_UNLIMITED_INSTANCES`. Каждый экземпляр канала предназначен для обмена данными по каналу между сервером и отдельным клиентом.

Параметры `nOutBufferSize` и `nInBufferSize` определяют соответственно размеры выходного и входного буферов для обмена данными по именованному каналу. Однако эти значения рассматриваются операционными системами Windows только как пожелания пользователя, а сам выбор размеров буферов остается за операционной системой.

Параметр `nDefaultTimeOut` устанавливает время ожидания клиентом связи с сервером. Это время используется при вызове клиентом функции `WaitNamedPipe`, в которой параметр `nTimeOut` имеет значение `NMPWAIT_USE_DEFAULT_WAIT`.

Для связи сервера с несколькими клиентами по одному именованному каналу сервер должен создать несколько экземпляров этого канала. Каждый экземпляр именованного канала создается вызовом функции `CreateNamedPipe`, которая возвращает дескриптор экземпляра именованного канала. Отметим, что в этом случае поток, создающий экземпляр именованного канала, должен иметь право доступа `FILE_CREATE_PIPE_INSTANCE` к именованному каналу. Этим правом по умолчанию обладает владелец именованного канала, т. е. тот процесс, который создал этот именованный канал.

16.3. Соединение сервера с клиентом

После того как сервер создал именованный канал, он должен дождаться соединения клиента с этим каналом. Для этого сервер вызывает функцию `ConnectNamedPipe`, которая имеет следующий прототип;

```
BOOL  ConnectNamedPipe(
  HANDLE  hNamedPipe,            // дескриптор канала
  LPOVERLAPPED  lpOverlapped     // асинхронная связь
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Сервер использует эту функцию для связи с клиентом по каждому свободному экземпляру именованного канала.

После окончания обмена данными с клиентом сервер может вызвать функцию `DisconnectNamedPipe`, которая имеет следующий прототип:

```
BOOL  DisconnectNamedPipe(
  HANDLE  hNamedPipe     // дескриптор канала
);
```

И возвращает ненулевое значение — в случае успеха — или значение FALSE — в случае неудачи. Эта функция разрывает связь сервера с клиентом. После этого клиент не может обмениваться данными с сервером по данному именованному каналу и поэтому любая операция доступа к именованному каналу со стороны клиента вызовет ошибку. После разрыва связи с одним клиентом сервер снова может вызвать функцию ConnectNamedPipe, чтобы установить связь по этому же именованному каналу с другим клиентом.

16.4. Соединение клиентов с именованным каналом

Прежде чем соединяться с именованным каналом, клиент должен определить, доступен ли какой-либо экземпляр этого канала для соединения. С этой целью клиент должен вызвать функцию WaitNamedPipe, которая имеет следующий прототип:

```
BOOL  WaitNamedPipe(
  LPCTSTR  lpNamedPipeName,  // указатель на имя канала
  DWORD    nTimeOut          // интервал ожидания
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — FALSE. Опишем параметры этой функции.

Параметр lpNamedPipeName указывает на строку, которая должна иметь вид:

```
\\server_name\pipe\pipe_name
```

Здесь server_name обозначает имя компьютера, на котором выполняется сервер именованного канала, слово pipe фиксировано, а pipe_name задает имя именованного канала.

Параметр nTimeOut задает временной интервал, в течение которого клиент ждет связь с сервером. Этот временной интервал определяется в миллисекундах или может быть равен одному из следующих значений:

- NMPWAIT_USE_DEFAULT_WAIT — интервал времени ожидания определяется значением параметра nDefaultTimeOut, который задается в функции CreateNamedPipe;
- NMPWAIT_WAIT_FOREVER — бесконечное время ожидания связи с именованным каналом.

Сделаем два важных замечания относительно работы функции WaitNamedPipe.

Замечание

Если не существует экземпляров именованного канала с именем lpNamedPipe, то функция WaitNamedPipe немедленно завершается неудачей независимо от времени ожидания, заданного параметром nTimeOut.

Замечание

Если клиент соединяется с каналом до вызова сервером функции ConnectNamedPipe, то функция WaitNamedPipe возвращает значение FALSE и функция GetLastError вернет код ERROR_PIPE_CONNECTED. Поэтому функцию WaitNamedPipe нужно вызывать только после соединения сервера с каналом посредством функции ConnectNamedPipe.

После того как обнаружен свободный экземпляр канала, чтобы установить связь с этим каналом клиент должен вызвать функцию CreateFile, которая имеет следующий прототип:

```
HANDLE  CreateFile(
  LPCTSTR  lpFileName,        // указатель на имя канала
  DWORD    dwDesiredAccess,   // чтение или запись в канал
  DWORD    dwShareMode,       // режим совместного использования
  LPSECURITY_ATTRIBUTES  lpSecurityAttributes, // атрибуты безопасности
  DWORD    dwCreationDisposition,  // флаг открытия канала
  DWORD    dwFlagsAndAttributes,   // флаги и атрибуты
  HANDLE   hTemplateFile           // дополнительные атрибуты
);
```

В случае успешного завершения эта функция возвращает дескриптор именованного канала, а в случае неудачи — значение INVALID_HANDLE_VALUE.

Если функция используется для открытия именованного канала, то ее параметры могут принимать следующие значения.

Параметр lpFileName должен указывать на имя канала, которое должно быть задано в том же формате, что и в функции WaitNamedPipe. Отметим, что если клиент работает на той же машине, что и сервер, и использует для открытия именованного канала в функции CreateFile имя канала как \\.\pipe\pipe_name, то файловая система именованных каналов (Named Pipe File System, NPFS) открывает этот именованный канал в режиме передачи данных потоком.

Чтобы открыть именованный канал в режиме передачи данных сообщениями нужно задавать имя канала в виде:

```
\\server_name\pipe\pipe_name
```

Параметр dwDesiredAccess может принимать одно из следующих значений:

- 0 — разрешает получить атрибуты канала;
- GENERIC_READ — разрешает чтение из канала;
- GENERIC_WRITE — разрешает запись в канал.

Следует отметить, что функция CreateFile завершается неудачей, если доступ к именованному каналу, заданный этими значениями, не соответствует

значениям параметра dwOpenMode в функции CreateNamedPipe. Кроме того, в этом параметре программист может определить стандартные права доступа к именованному каналу. За более подробной информацией по этому вопросу нужно обратиться к *разд. 36.7*, *41.3*, в которых описываются стандартные режимы доступа.

Параметр dwShareMode определяет режим совместного использования именованного канала и может принимать значение 0, которое запрещает совместное использование именованного канала или любую комбинацию следующих значений:

- FILE_SHARE_READ — разрешает совместное чтение из канала;
- FILE_SHARE_WRITE — разрешает совместную запись в канал.

Параметр lpSecurityAttributes задает атрибуты безопасности именованного канала. Пока значение этого параметра будем устанавливать в NULL.

Для именованного канала параметр dwCreationDisposition должен быть равен значению OPEN_EXISTING, т. к. клиент всегда открывает существующий именованный канал.

Для именованного канала параметр dwFlagsAndAttributes можно задать равным 0, что определяет флаги и атрибуты по умолчанию. Подробную информацию о значениях этого параметра смотрите в *гл. 24*, посвященной работе с файлами в Windows.

Значение параметра hTemplateFile задается равным NULL.

Сделаем следующие замечания относительно работы с функцией CreateFile в случае, когда она используется для открытия доступа к именованному каналу.

Замечание

Несмотря на то, что функция WaitNamedPipe может успешно завершиться, последующий вызов функции CreateFile может завершиться неудачей по следующим причинам: между вызовами этих функций сервер закрыл канал или между вызовами функций другой клиент связался с экземпляром этого канала. Для предотвращения последней ситуации сервер должен создавать новый экземпляр именованного канала после каждого успешного завершения функции ConnectNamedPipe или создать сразу несколько экземпляров именованного канала.

Замечание

Если заранее известно, что сервер вызвал функцию ConnectNamedPipe, то функция CreateFile может вызываться без предварительного вызова функции WaitNamedPipe.

Отметим также один момент, который касается связи сервера с клиентом именованного канала. Может возникнуть такая ситуация, что сервер вызвал функцию `ConnectNamedPipe`, а клиента, который хочет связаться с именованным каналом, не существует. В этом случае серверное приложение будет заблокировано. Чтобы иметь возможность обработать такую ситуацию, функцию `ConnectNamedPipe` следует вызывать в отдельном потоке серверного приложения. Тогда для разблокировки серверного приложения можно вызвать функцию для связи клиента с именованным каналом из другого потока этого приложения.

16.5. Обмен данными по именованному каналу

Как и в случае с анонимным каналом, для обмена данными по именованному каналу используются функции `ReadFile` и `WriteFile`, но с одним отличием, которое заключается в следующем. Так как в случае именованного канала разрешен асинхронный обмен данными, то в функциях `ReadFile` и `WriteFile` может использоваться параметр `lpOverlapped` при том условии, что в вызове функции `CreateNamedPipe` в параметре `dwOpenMode` был установлен флаг `FILE_FLAG_OVERLAPPED`. Максимально в именованный канал может быть записано до 65 535 байт одной операцией `WriteFile`. Подробно работа с функциями `ReadFile` и `WriteFile` рассмотрена в *гл. 24*, которая посвящена работе с файлами в Windows. Кроме того, отметим, что для асинхронного обмена данными по именованному каналу могут использоваться также функции `ReadFileEx` и `WriteFileEx`, которые также рассмотрены в *гл. 24*.

После завершения обмена данными по именованному каналу потоки должны закрыть дескрипторы экземпляров именованного канала, используя функцию `CloseHandle`.

Теперь перейдем к примерам программ, в которых показано, как обмениваться данными по именованному каналу. Сначала рассмотрим пример, в котором процесс-сервер создает именованный канал, а затем ждет, пока клиент не соединится с именованным каналом. После этого сервер читает из именованного канала десять чисел и выводит их на консоль. Сначала приведем программу процесса-сервера именованного канала.

Листинг 16.1. Процесс-сервер именованного канала

```
#include <windows.h>
#include <iostream.h>


int main()
```

```
{
  HANDLE hNamedPipe;

  // создаем именованный канал для чтения
  hNamedPipe = CreateNamedPipe(
    "\\\\.\\pipe\\demo_pipe",      // имя канала
    PIPE_ACCESS_INBOUND,           // читаем из канала
    PIPE_TYPE_MESSAGE | PIPE_WAIT, // синхронная передача сообщений
    1,         // максимальное количество экземпляров канала
    0,         // размер выходного буфера по умолчанию
    0,         // размер входного буфера по умолчанию
    INFINITE,  // клиент ждет связь бесконечно долго
    NULL       // защита по умолчанию
    );
  // проверяем на успешное создание
  if (hNamedPipe == INVALID_HANDLE_VALUE)
  {
    cerr << "Create named pipe failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to exit.";
    cin.get();

    return 0;
  }

  // ждем, пока клиент свяжется с каналом
  cout << "The server is waiting for connection with a client." << endl;
  if(!ConnectNamedPipe(
    hNamedPipe,    // дескриптор канала
    NULL      // связь синхронная
    ))
  {
    cerr << "The connection failed." << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hNamedPipe);
    cout << "Press any key to exit.";
    cin.get();

    return 0;
```

```
  }

  // читаем данные из канала
  for (int i = 0; i < 10; i++)
  {
    int nData;
    DWORD dwBytesRead;
    if (!ReadFile(
      hNamedPipe,    // дескриптор канала
      &nData,        // адрес буфера для ввода данных
      sizeof(nData), // число читаемых байтов
      &dwBytesRead,  // число прочитанных байтов
      NULL           // передача данных синхронная
      ))
    {
      cerr << "Read file failed." << endl
        << "The last error code: " << GetLastError() << endl;
      CloseHandle(hNamedPipe);
      cout << "Press any key to exit.";
      cin.get();

      return 0;
    }
    // выводим прочитанные данные на консоль
    cout << "The number " << nData << " was read by the server" << endl;
  }
  // закрываем дескриптор канала
  CloseHandle(hNamedPipe);
  // завершаем процесс
  cout << "The data are read by the server."<<endl;
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

Теперь, в листинге 16.2, приведем программу процесса-клиента именованного канала, который сначала связывается с именованным каналом, а затем записывает в него десять чисел.

Листинг 16.2. Процесс-клиент именованного канала

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE hNamedPipe;
  char pipeName[] = "\\\\.\\pipe\\demo_pipe";

  // связываемся с именованным каналом
  hNamedPipe = CreateFile(
    pipeName,         // имя канала
    GENERIC_WRITE,    // записываем в канал
    FILE_SHARE_READ, // разрешаем одновременное чтение из канала
    NULL,             // защита по умолчанию
    OPEN_EXISTING,    // открываем существующий канал
    0,      // атрибуты по умолчанию
    NULL    // дополнительных атрибутов нет
    );

  // проверяем связь с каналом
  if (hNamedPipe == INVALID_HANDLE_VALUE)
  {
    cerr << "Connection with the named pipe failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to exit.";

    return 0;
  }

  // пишем в именованный канал
  for (int i = 0; i < 10; i++)
  {
    DWORD dwBytesWritten;
    if (!WriteFile(
      hNamedPipe,  // дескриптор канала
      &i,          // данные
      sizeof(i),   // размер данных
```

```
      &dwBytesWritten,   // количество записанных байтов
      NULL         // синхронная запись
      ))
    {
      // ошибка записи
      cerr << "Writing to the named pipe failed: " << endl
        << "The last error code: " << GetLastError() << endl;
      CloseHandle(hNamedPipe);
      cout << "Press any key to exit.";
      cin.get();

      return 0;
    }
    // выводим число на консоль
    cout << "The number " << i << " is written to the named pipe."
         << endl;
    Sleep(1000);
  }
  // закрываем дескриптор канала
  CloseHandle(hNamedPipe);
  // завершаем процесс
  cout << "The data are written by the client." << endl
    << "Press any key to exit.";
  cin.get();

  return 0;
}
```

В листинге 16.3 рассмотрим программу процесса-сервера именованного канала, который сначала создает именованный канал, затем ждет подключения к нему клиента. После этого сервер принимает от клиента одно сообщение, выводит это сообщение на консоль и посылает клиенту ответное сообщение.

Листинг 16.3. Сервер именованного канала, отвечающий на сообщение от клиента

```
#include <windows.h>
#include <iostream.h>
#include <string.h>
```

```
int main()
{
  HANDLE   hNamedPipe;
  DWORD   dwBytesRead;     // для количества прочитанных байтов
  DWORD   dwBytesWrite;    // для количества записанных байтов
  char     pchMessage[80]; // для сообщения
  int    nMessageLength;   // длина сообщения

  // создаем именованный канал для чтения и записи
  hNamedPipe = CreateNamedPipe(
    "\\\\.\\pipe\\demo_pipe",  // имя канала
    PIPE_ACCESS_DUPLEX,        // читаем из канала и пишем в канал
    PIPE_TYPE_MESSAGE | PIPE_WAIT,   // синхронная передача сообщений
    1,         // максимальное количество экземпляров канала
    0,         // размер выходного буфера по умолчанию
    0,         // размер входного буфера по умолчанию
    INFINITE,  // клиент ждет связь бесконечно долго
    NULL       // безопасность по умолчанию
    );
  // проверяем на успешное создание
  if (hNamedPipe == INVALID_HANDLE_VALUE)
  {
    cerr << "Create named pipe failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to exit.";
    cin.get();

    return 0;
  }

  // ждем, пока клиент свяжется с каналом
  cout << "The server is waiting for connection with a client." << endl;
  if(!ConnectNamedPipe(
    hNamedPipe,    // дескриптор канала
    NULL      // связь синхронная
    ))
  {
    cerr << "Connect named pipe failed." << endl
      << "The last error code: "<<GetLastError() << endl;
```

```
  CloseHandle(hNamedPipe);
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
// читаем сообщение от клиента
if (!ReadFile(
  hNamedPipe,      // дескриптор канала
  pchMessage,      // адрес буфера для ввода данных
  sizeof(pchMessage),  // количество читаемых байтов
  &dwBytesRead,    // количество прочитанных байтов
  NULL))           // передача данных синхронная
{
  cerr << "Data reading from the named pipe failed." << endl
    << "The last error code: "<< GetLastError() << endl;
  CloseHandle(hNamedPipe);
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}

// выводим полученное от клиента сообщение на консоль
cout << "The server received the message from a client: "
  << endl << '\t' << pchMessage << endl;

// вводим строку
cout << "Input a string: ";
cin.getline(pchMessage, 80);
// определяем длину строки
nMessageLength = strlen(pchMessage) + 1;

// отвечаем клиенту
if (!WriteFile(
  hNamedPipe,      // дескриптор канала
  pchMessage,      // адрес буфера для вывода данных
  nMessageLength,  // количество записываемых байтов
```

```
    &dwBytesWrite,   // количество записанных байтов
    NULL             // передача данных синхронная
    ))
  {
    cerr << "Write file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hNamedPipe);
    cout << "Press any key to exit.";
    cin.get();

    return 0;
  }

  // выводим посланное клиенту сообщение на консоль
  cout << "The server sent the message to a client: "
    << endl << '\t' << pchMessage << endl;

  // закрываем дескриптор канала
  CloseHandle(hNamedPipe);
  // завершаем процесс
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

В этой программе обратим внимание на следующий момент: если клиент и сервер работают на разных компьютерах в локальной сети, то вход, как на компьютер сервера, так и на компьютер клиента, должен осуществляться с одинаковыми именами (логинами) и паролями. Так как по умолчанию атрибуты безопасности именованного канала устанавливаются таким образом, что он принадлежит только пользователю, создавшему этот именованный канал. В следующем примере процесса-сервера именованного канала атрибуты безопасности будут установлены таким образом, чтобы они разрешали доступ к именованному каналу любому пользователю. Теперь же, в листинге 16.4, приведем программу процесса-клиента именованного канала, который сначала вводит с консоли имя компьютера в локальной сети, на котором запущен сервер именованного канала. Затем связывается с этим именованным каналом. После этого клиент передает серверу одно сообщение и получает от него ответное сообщение и выводит на консоль.

Листинг 16.4. Клиент именованного канала, посылающий сообщение серверу

```
#include <windows.h>
#include <iostream.h>
#include <string.h>

int main()
{
  char   machineName[80];
  char   pipeName[80];
  HANDLE   hNamedPipe;
  DWORD   dwBytesWritten;    // для количества записанных байтов
  DWORD   dwBytesRead;       // для количества прочитанных байтов
  char     pchMessage[80];   // для сообщения
  int    nMessageLength;     // длина сообщения

  // вводим имя машины в сети, на которой работает сервер
  cout << "Enter a name of the server machine: ";
  cin >> machineName;
  // подставляем имя машины в имя канала
  wsprintf(pipeName, "\\\\%s\\pipe\\demo_pipe", machineName);

  // связываемся с именованным каналом
  hNamedPipe = CreateFile(
    pipeName,          // имя канала
    GENERIC_READ | GENERIC_WRITE,       // читаем и записываем в канал
    FILE_SHARE_READ | FILE_SHARE_WRITE, // разрешаем чтение и запись
    NULL,                  // безопасность по умолчанию
    OPEN_EXISTING,         // открываем существующий канал
    FILE_ATTRIBUTE_NORMAL, // атрибуты по умолчанию
    NULL);                 // дополнительных атрибутов нет

  // проверяем связь с каналом
  if (hNamedPipe==INVALID_HANDLE_VALUE)
  {
    cerr << "Connection with the named pipe failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to exit.";
```

```
  cin.get();

  return 0;
}

// вводим строку
cin.get();
cout << "Input a string: ";
cin.getline(pchMessage, 80);
// определяем длину строки
nMessageLength = strlen(pchMessage) + 1;

// пишем в именованный канал
if (!WriteFile(
  hNamedPipe,      // дескриптор канала
  pchMessage,      // данные
  nMessageLength,  // размер данных
  &dwBytesWritten, // количество записанных байтов
  NULL))           // синхронная запись
{
  // ошибка записи
  cerr << "Write file failed: " << endl
    << "The last error code: " << GetLastError() << endl;
  CloseHandle(hNamedPipe);
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
// выводим посланное сообщение на консоль
cout << "The client sent the message to a server: "
  << endl << '\t' << pchMessage << endl;
// читаем из именованного канала
if (!ReadFile(
  hNamedPipe,      // дескриптор канала
  pchMessage,      // данные
  sizeof(pchMessage),  // размер данных
  &dwBytesRead,    // количество записанных байтов
  NULL))           // синхронное чтение
```

```
  {
    // ошибка чтения
    cerr << "Read file failed: " << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hNamedPipe);
    cout << "Press any key to exit.";
    cin.get();

    return 0;
  }
  // выводим полученное сообщение на консоль
  cout << "The client received the message from a server: "
    << endl << '\t' << pchMessage << endl;
  // закрываем дескриптор канала
  CloseHandle(hNamedPipe);
  // завершаем процесс
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

Теперь, в листинге 16.5, приведем программу процесса-сервера именованного канала, который создает именованный канал, доступный всем пользователям. В этом случае связаться с именованным каналом можно любому пользователю.

Листинг 16.5. Сервер, создающий общедоступный именованный канал

```
#include <windows.h>
#include <iostream.h>
#include <string.h>

int main()
{
  HANDLE   hNamedPipe;
  SECURITY_ATTRIBUTES sa;     // атрибуты безопасности
  SECURITY_DESCRIPTOR sd;     // дескриптор безопасности
  DWORD   dwBytesRead;        // для количества прочитанных байтов
```

```
DWORD  dwBytesWrite;       // для количества записанных байтов
char  pchMessage[80];      // для сообщения
int  nMessageLength;       // длина сообщения

// инициализация атрибутов безопасности
sa.nLength = sizeof(sa);
sa.bInheritHandle = FALSE;   // дескриптор канала ненаследуемый
// инициализируем дескриптор безопасности
InitializeSecurityDescriptor(&sd, SECURITY_DESCRIPTOR_REVISION);
// разрешаем доступ к именованному каналу всем пользователям
SetSecurityDescriptorDacl(&sd, TRUE, NULL, FALSE);
sa.lpSecurityDescriptor = &sd;

// создаем именованный канал для чтения и записи
hNamedPipe = CreateNamedPipe(
  "\\\\.\\pipe\\demo_pipe",      // имя канала
  PIPE_ACCESS_DUPLEX,    // читаем из канала и пишем в канал
  PIPE_TYPE_MESSAGE | PIPE_WAIT, // синхронная передача сообщений
  1,         // максимальное количество экземпляров канала
  0,         // размер выходного буфера по умолчанию
  0,         // размер входного буфера по умолчанию
  INFINITE,  // клиент ждет связь бесконечно долго
  &sa        // доступ для всех пользователей
  );
// проверяем на успешное создание
if (hNamedPipe == INVALID_HANDLE_VALUE)
{
  cerr << "Create named pipe failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}

// ждем, пока клиент свяжется с каналом
cout << "The server is waiting for connection with a client." << endl;
if(!ConnectNamedPipe(
  hNamedPipe,  // дескриптор канала
```

```
  NULL          // связь синхронная
  ))
{
  cerr << "Connect named pipe failed." << endl
    << "The last error code: "<<GetLastError() << endl;
  CloseHandle(hNamedPipe);
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
// читаем сообщение от клиента
if (!ReadFile(
  hNamedPipe,     // дескриптор канала
  pchMessage,     // адрес буфера для ввода данных
  sizeof(pchMessage),  // количество читаемых байтов
  &dwBytesRead,  // количество прочитанных байтов
  NULL))          // передача данных синхронная
{
  cerr << "Data reading from the named pipe failed." << endl
    << "The last error code: "<< GetLastError() << endl;
  CloseHandle(hNamedPipe);
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}

// выводим полученное от клиента сообщение на консоль
cout << "The server received the message from a client: "
  << endl << '\t' << pchMessage << endl;

// вводим строку
cout << "Input a string: ";
cin.getline(pchMessage, 80);
// определяем длину строки
nMessageLength = strlen(pchMessage) + 1;

// отвечаем клиенту
if (!WriteFile(
```

```
    hNamedPipe,    // дескриптор канала
    pchMessage,    // адрес буфера для вывода данных
    nMessageLength,    // количество записываемых байтов
    &dwBytesWrite,     // количество записанных байтов
    NULL           // передача данных синхронная
    ))
  {
    cerr << "Write file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hNamedPipe);
    cout << "Press any key to exit.";
    cin.get();

    return 0;
  }

  // выводим посланное клиенту сообщение на консоль
  cout << "The server sent the message to a client: "
    << endl << '\t' << pchMessage << endl;

  // закрываем дескриптор канала
  CloseHandle(hNamedPipe);
  // завершаем процесс
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

В заключение этого раздела отметим, что для создания и связи с именованными каналами пользователь должен иметь соответствующие полномочия.

16.6. Копирование данных из именованного канала

Для копирования данных из именованного канала используется функция `PeekNamedPipe`, которая копирует данные в буфер, не удаляя их из канала. Эта функция имеет следующий прототип:

```
BOOL  PeekNamedPipe(
  HANDLE   hNamedPipe,          // дескриптор канала
```

```
    LPVOID   lpBuffer,           // буфер данных
    DWORD    nBufferSize,        // размер буфера данных
    LPDWORD  lpBytesRead,        // количество прочитанных байтов
    LPDWORD  lpTotalBytesAvail,  // количество доступных байтов
    LPDWORD  lpBytesLeftThisMessage   // количество непрочитанных байтов
  );
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — FALSE. Следует отметить, что если данных в канале нет, то функция немедленно возвращает управление и устанавливает в ноль значения переменных, на которые указывают параметры lpBytesRead, lpTotalBytesAvail и lpBytesLeftThisMessage. Параметры функции PeekNamedPipe имеют следующее назначение.

В параметре hNamedPipe должен быть установлен дескриптор именованного канала. Причем канал должен быть открыт в режиме чтения.

Параметр lpBuffer должен указывать на область памяти, в которую функция читает данные из именованного канала. Если в этом параметре установлено значение NULL, то данные из именованного канала читаться не будут.

В параметре nBufferSize должен быть установлен размер области памяти, на которую указывает параметр lpBuffer.

Параметр lpBytesRead должен указывать на переменную типа DWORD, в которую функция записывает количество прочитанных байтов. Если данные из именованного канала не читаются, то значение этого параметра может быть установлено в NULL.

Параметр lpTotalBytesAvail должен указывать на переменную типа DWORD, в которую функция записывает количество доступных для чтения байтов, находящихся в именованном канале. Если данные из именованного канала не читаются, то значение этого параметра может быть установлено в NULL.

Параметр lpBytesLeftThisMessage должен указывать на переменную, в которую функция помещает количество непрочитанных байтов из сообщения. Если данные из именованного канала не читаются, то значение этого параметра может быть установлено в NULL.

Отметим, что если данные передаются потоком, а не сообщениями, то функция PeekNamedPipe читает ровно столько байтов данных, какова длина буфера данных. Если же данные передаются сообщениями, то функция читает полностью сообщение, которое входит в буфер данных. В противном случае читается часть сообщения, а количество непрочитанных байтов возвращается через переменную, на которую указывает параметр lpBytesLeftThisMessage.

В листинге 16.6 приведена программа процесса-клиента именованного канала, который использует функцию PeekNamedPipe для копирования данных из именованного канала.

Листинг 16.6. Пример копирования сообщения клиентом именованного канала

```
#include <windows.h>
#include <iostream.h>
#include <string.h>

int main()
{
  char    machineName[80];
  char    pipeName[80];
  HANDLE   hNamedPipe;
  DWORD   dwBytesRead;          // для количества прочитанных байтов
  DWORD  dwTotalBytesAvail;     // количество байтов в канале
  DWORD  dwBytesLeftThisMessage;   // количество непрочитанных байтов
  char     pchMessage[80];      // для сообщения

  // вводим имя машины в сети, на которой работает сервер
  cout << "Enter a name of the server machine: ";
  cin >> machineName;
  cin.get();
  // подставляем имя машины в имя канала
  wsprintf(pipeName, "\\\\%s\\pipe\\demo_pipe", machineName);

  // связываемся с именованным каналом
  hNamedPipe = CreateFile(
    pipeName,           // имя канала
    GENERIC_READ | GENERIC_WRITE,    // читаем и записываем в канал
    FILE_SHARE_READ | FILE_SHARE_WRITE,  // разрешаем чтение и запись
    NULL,             // безопасность по умолчанию
    OPEN_EXISTING,    // открываем существующий канал
    FILE_ATTRIBUTE_NORMAL,    // атрибуты по умолчанию
    NULL);            // дополнительных атрибутов нет

  // проверяем связь с каналом
  if (hNamedPipe==INVALID_HANDLE_VALUE)
  {
    cerr << "Connection with the named pipe failed." << endl
      << "The last error code: " << GetLastError() << endl;
```

```
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}

// ждем команду на копирование сообщения из канала
cout << "Press any key to peek a message." << endl;
cin.get();

// копируем информацию из именованного канала
if (!PeekNamedPipe(
  hNamedPipe,            // дескриптор канала
  pchMessage,            // данные
  sizeof(pchMessage),    // размер данных
  &dwBytesRead,          // количество записанных байтов
  &dwTotalBytesAvail,    // количество байтов в канале
  &dwBytesLeftThisMessage  // количество непрочитанных байтов
  ))
{
  // ошибка чтения сообщения
  cerr << "Peek named pipe failed: " << endl
    << "The last error code: " << GetLastError() << endl;
  CloseHandle(hNamedPipe);
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
// выводим полученное сообщение на консоль
if (dwTotalBytesAvail)
  cout << "The peeked message: "
  << endl << '\t' << pchMessage << endl;
else
  cout << "There is no mesage." << endl;

// теперь читаем сообщение из именованного канала
if (!ReadFile(
  hNamedPipe,      // дескриптор канала
```

```
    pchMessage,      // данные
    sizeof(pchMessage),  // размер данных
    &dwBytesRead,    // количество записанных байтов
    NULL))           // синхронное чтение
  {
    // ошибка чтения
    cerr << "Read file failed: " << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hNamedPipe);
    cout << "Press any key to exit.";
    cin.get();

    return 0;
  }
  // выводим полученное сообщение на консоль
  cout << "The client received the message from a server: "
    << endl << '\t' << pchMessage << endl;
  // закрываем дескриптор канала
  CloseHandle(hNamedPipe);
  // завершаем процесс
  cout << "Press any key to exit." << endl;
  cin.get();

  return 0;
}
```

16.7. Передача транзакций по именованному каналу

Для обмена сообщениями по сети может также использоваться функция `TransactNamedPipe`, которая объединяет операции записи и чтения в одну операцию, которая называется транзакцией. Отметим, что функция `TransactNamedPipe` может использоваться только в том случае, если сервер при создании именованного канала установил флаг `PIPE_TYPE_MESSAGE`. Функция `TransactNamedPipe` имеет следующий прототип:

```
BOOL  TransactNamedPipe(
  HANDLE   hNamedPipe,      // дескриптор именованного канала
  LPVOID   lpInBuffer,      // буфер для записи в канал
  DWORD    dwInBufferSize,  // длина буфера для записи
```

```
    LPVOID   lpOutBuffer,     // буфер для чтения из канала
    DWORD    dwOutBufferSize, // длина буфера для чтения
    LPDWORD  lpBytesRead,     // количество прочитанных байтов
    LPOVERLAPPED  lpOverlapped   // асинхронный доступ к каналу
  );
```

В случае успешного завершения функция `TransactNamedPipe` возвращает ненулевое значение, а в случае неудачи — `FALSE`. Параметры функции имеют следующее назначение.

В параметре `hNamedPipe` должен быть установлен дескриптор именованного канала.

Параметр `lpInBuffer` должен указывать на буфер, из которого записываются данные в именованный канал.

Параметр `dwInBufferSize` должен содержать длину передаваемого сообщения в байтах.

Параметр `lpOutBuffer` должен указывать на буфер, в который читаются данные из именованного канала.

Параметр `dwOutBufferSize` должен содержать длину буфера, в который читается сообщение.

Параметр `lpBytesRead` должен указывать на переменную типа `DWORD`, в которую функция поместит количество прочитанных байтов. Если осуществляется асинхронный доступ к именованному каналу, то значение этого параметра можно установить в `NULL`.

Параметр `lpOverlapped` используется в том случае, если осуществляется асинхронный доступ к именованному каналу. В этом случае параметр должен указывать на структуру типа `OVERLAPPED`. Пока будем устанавливать этот параметр в `NULL`, что задает синхронную передачу данных. Подробно асинхронная передача данных описана в *гл. 27*.

В листинге 16.7 приведена программа процесса-клиента именованного канала, которая передает транзакцию, используя функцию `TransactNamedPipe`. Отметим, что если клиент и сервер именованного канала работают на одном компьютере, то для связи клиента с именованным каналом нужно вводить полное имя компьютера. Если же вместо имени компьютера будет введена точка, то именованный канал откроется в режиме передачи данных потоком, а не сообщениями. Это, в свою очередь, вызовет ошибку при передаче данных.

Листинг 16.7. Пример передачи транзакции по именованному каналу

```
#include <windows.h>
#include <iostream.h>
#include <string.h>
```

```
int main()
{
  char    machineName[80];
  char    pipeName[80];
  HANDLE  hNamedPipe;
  DWORD   dwBytesRead;       // для количества прочитанных байтов
  char    pchInBuffer[80];   // для записи сообщения
  char    pchOutBuffer[80];  // для чтения сообщения
  int     nMessageLength;    // длина сообщения

  // вводим имя машины в сети, на которой работает сервер
  cout << "Enter a name of the server machine: ";
  cin >> machineName;
  // подставляем имя машины в имя канала
  wsprintf(pipeName, "\\\\%s\\pipe\\demo_pipe", machineName);

  // связываемся с именованным каналом
  hNamedPipe = CreateFile(
    pipeName,          // имя канала
    GENERIC_READ | GENERIC_WRITE,    // читаем и записываем в канал
    FILE_SHARE_READ | FILE_SHARE_WRITE,  // разрешаем чтение и запись
    NULL,            // безопасность по умолчанию
    OPEN_EXISTING,   // открываем существующий канал
    FILE_ATTRIBUTE_NORMAL,   // атрибуты по умолчанию
    NULL);           // дополнительных атрибутов нет

  // проверяем связь с каналом
  if (hNamedPipe==INVALID_HANDLE_VALUE)
  {
    cerr << "Connection with the named pipe failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to exit.";
    cin.get();

    return 0;
  }

  // вводим строку
  cin.get();
```

```
  cout << "Input a string: ";
  cin.getline(pchInBuffer, 80);
  // определяем длину строки
  nMessageLength = strlen(pchInBuffer) + 1;

  // пишем и читаем из именованного канала одной транзакцией
  if (!TransactNamedPipe(
    hNamedPipe,       // дескриптор канала
    &pchInBuffer,     // адрес входного буфера канала
    nMessageLength,   // длина входного сообщения
    &pchOutBuffer,    // адрес выходного буфера канала
    sizeof(pchOutBuffer),     // длина выходного буфера канала
    &dwBytesRead,     // количество прочитанных байтов
    NULL))            // передача транзакции синхронная
  {
    // ошибка транзакции
    cerr << "Transact named pipe failed: " << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hNamedPipe);
    cout << "Press any key to exit.";
    cin.get();

    return 0;
  }
  // выводим посланное сообщение на консоль
  cout << "The sent message: "
    << endl << '\t' << pchInBuffer << endl;
  // выводим полученное сообщение на консоль
  cout << "The received message: "
    << endl << '\t' << pchOutBuffer << endl;
  // закрываем дескриптор канала
  CloseHandle(hNamedPipe);
  // завершаем процесс
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

Для передачи единственной транзакции по именованному каналу используется функция `CallNamedPipe`, которая работает следующим образом. Сначала она связывается с именованным каналом, используя его имя. При этом заметим, что именованный канал должен быть открыт в режиме передачи данных сообщениями. Потом функция передает по именованному каналу единственное сообщение и получает сообщение в ответ, а после этого разрывает связь с именованным каналом. Функция `CallNamedPipe` имеет следующий прототип:

```
BOOL  CallNamedPipe(
  LPCTSTR  lpNamedPipeName,  // имя именованного канала
  LPVOID   lpInBuffer,       // буфер для записи данных в канал
  DWORD    dwInBufferSize,   // размер буфера для записи данных
  LPVOID   lpOutBuffer,      // буфер для чтения данных из канала
  DWORD    dwOutBufferSize,  // размер буфера для чтения данных
  LPDWORD  lpBytesRead,      // количество прочитанных байтов
  DWORD    dwTimeOut         // интервал ожидания
);
```

В случае успешного завершения функция `CallNamedPipe` возвращает ненулевое значение, а в случае неудачи — `FALSE`. Опишем параметры этой функции.

Параметр `lpNamedPipeName` должен указывать на строку, содержащую имя именованного канала.

Параметр `lpInBuffer` должен указывать на буфер, из которого записываются данные в именованный канал.

Параметр `dwInBufferSize` должен содержать длину передаваемого сообщения в байтах.

Параметр `lpOutBuffer` должен указывать на буфер, в который читаются данные из именованного канала.

Параметр `dwOutBufferSize` должен содержать длину буфера, в который читается сообщение.

Параметр `lpBytesRead` должен указывать на переменную типа `DWORD`, в которую функция поместит количество прочитанных байтов.

В параметре `dwTimeOut` должен быть установлен интервал ожидания в миллисекундах, в течение которого функция ждет связи с именованным каналом. Кроме того, можно установить следующие значения этого параметра:

- `NMPWAIT_NOWAIT` — если нет свободного экземпляра именованного канала, то функция немедленно возвращает управление;
- `NMPWAIT_WAIT_FOREVER` — функция ждет бесконечно долго связи с экземпляром именованного канала;

- NMPWAIT_USE_DEFAULT_WAIT — интервал ожидания определяется значением, заданным при создании именованного канала функцией CreateNamedPipe.

В листинге 16.8 приведена программа процесса-клиента именованного канала, который передает транзакцию, используя функцию CallNamedPipe.

Листинг 16.8. Пример передачи транзакции по именованному каналу

```
#include <windows.h>
#include <iostream.h>
#include <string.h>

int main()
{
  char    machineName[80];
  char    pipeName[80];
  DWORD   dwBytesRead;       // для количества прочитанных байтов
  char    pchInBuffer[80];   // для записи сообщения
  char    pchOutBuffer[80];  // для чтения сообщения
  int     nMessageLength;    // длина сообщения

  // вводим имя машины в сети, на которой работает сервер
  cout << "Enter a name of the server machine: ";
  cin >> machineName;
  // подставляем имя машины в имя канала
  wsprintf(pipeName, "\\\\%s\\pipe\\demo_pipe", machineName);

  // вводим строку
  cin.get();
  cout << "Input a string: ";
  cin.getline(pchInBuffer, 80);
  // определяем длину строки
  nMessageLength = strlen(pchInBuffer) + 1;

  // связываемся, пишем и читаем из именованного канала одной транзакцией
  if (!CallNamedPipe(
    pipeName,        // имя именованного канала
    &pchInBuffer,    // адрес входного буфера канала
    nMessageLength,  // длина входного сообщения
    &pchOutBuffer,   // адрес выходного буфера канала
```

```
    sizeof(pchOutBuffer),    // длина выходного буфера канала
    &dwBytesRead,    // количество прочитанных байтов
    NMPWAIT_WAIT_FOREVER))   // ждем бесконечно долго
  {
    // ошибка транзакции
    cerr << "Call named pipe failed: " << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to exit.";
    cin.get();

    return 0;
  }
  // выводим посланное сообщение на консоль
  cout << "The sent message: "
    << endl << '\t' << pchInBuffer << endl;
  // выводим полученное сообщение на консоль
  cout << "The received message: "
    << endl << '\t' << pchOutBuffer << endl;
  // завершаем процесс
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

16.8. Определение и изменение состояния именованного канала

Для получения информации о состоянии именованного канала используется функция `GetNamedPipeHandleState`, которая имеет следующий прототип:

```
BOOL  GetNamedPipeHandleState(
  HANDLE   hNamedPipe,           // дескриптор именованного канала
  LPDWORD  lpState,              // состояние канала
  LPDWORD  lpCurrentInstances,   // количество экземпляров канала
  LPDWORD  lpMaxCollectionCount, // максимальное количество байтов
  LPDWORD  lpCollectionDataTimeout,  // интервал ожидания
  LPTSTR   lpUserName,           // имя клиента именованного канала
  DWORD    dwMaxUserNameSize     // длина буфера для имени клиента
);
```

В случае успешного завершения функция GetNamedPipeHandleState возвращает ненулевое значение, а в случае неудачи — FALSE. Параметры функции имеют следующее назначение.

В параметре hNamedPipe должен быть установлен дескриптор именованного канала. Причем канал должен быть открыт в режиме чтения.

Параметр lpState должен указывать на переменную типа DWORD, в которую функция записывает любую комбинацию следующих значений:

- PIPE_NOWAIT — канал не блокирован;
- PIPE_READMODE_MESSAGE — канал открыт в режиме передачи данных сообщениями.

Если значение PIPE_NOWAIT не установлено, то канал блокирован. Если не установлено значение PIPE_READMODE_MESSAGE, то канал открыт в режиме передачи данных потоком. Если определять состояние именованного канала не нужно, то в параметре lpState может быть установлено значение NULL.

Параметр lpCurrentInstances должен указывать на переменную типа DWORD, в которую функция записывает количество созданных экземпляров именованного канала. Если эта информация не нужна, то значение этого параметра может быть установлено в NULL.

Параметр lpMaxCollectionCount должен указывать на переменную типа DWORD, в которую функция запишет максимальное количество байтов, которые клиент именованного канала должен записать в этот канал, прежде чем данные будут переданы серверу канала. Этот параметр должен быть установлен в NULL, если функция вызывается сервером именованного канала или если клиент и сервер работают на одном компьютере и для связи с сервером клиент использует символ "." вместо имени сервера. Если информация о максимальном количестве байтов не нужна, то в этом параметре может быть установлено значение NULL.

Параметр lpCollectionDataTimeout должен указывать на переменную типа DWORD, в которую функция поместит количество миллисекунд, которые могут пройти, прежде чем данные могут быть переданы по сети. Как и в предыдущем случае, этот параметр должен быть установлен в NULL, в том случае, если функция вызывается сервером именованного канала или если клиент и сервер работают на одном компьютере, и для связи с сервером клиент использует символ "." вместо имени сервера. Если информация об интервале задержки перед передачей данных не нужна, то в этом параметре может быть установлено значение NULL.

Параметр lpUserName должен указывать на символьный массив, в который функция поместит строку с именем владельца именованного канала. Имя владельца можно получить только в том случае, если доступ к каналу открыт всем пользователям. Если информация об имени владельца канала не нужна, то этот параметр может быть установлен в NULL.

Параметр `dwMaxUserNameSize` должен содержать размер области памяти, на которую указывает параметр `lpUserName`.

В листинге 16.9 приведена программа процесса-клиента именованного канала, которая получает информацию о состоянии именованного канала посредством вызова функции `GetNamedPipeHandleState`.

Листинг 16.9. Пример определения состояния именованного канала

```
#include <windows.h>
#include <iostream.h>
#include <string.h>

int main()
{
  char  machineName[80];
  char  pipeName[80];
  HANDLE   hNamedPipe;

  DWORD  dwState;              // состояние канала
  DWORD  dwCurInstances;       // количество экземпляров канала
  DWORD  dwMaxCollectionCount; // размер буфера клиента канала
  DWORD  dwCollectDataTimeout; // задержка перед передачей данных
  TCHAR  chUserName[255];      // имя владельца именованного канала

  // вводим имя машины в сети, на которой работает сервер
  cout << "Enter a name of the server machine: ";
  cin >> machineName;
  cin.get();
  // подставляем имя машины в имя канала
  wsprintf(pipeName, "\\\\%s\\pipe\\demo_pipe", machineName);

  // связываемся с именованным каналом
  hNamedPipe = CreateFile(
    pipeName,                // имя канала
    GENERIC_READ | GENERIC_WRITE,        // читаем и записываем в канал
    FILE_SHARE_READ | FILE_SHARE_WRITE,  // разрешаем чтение и запись
                                         // в канал
    NULL,                    // безопасность по умолчанию
    OPEN_EXISTING,           // открываем существующий канал
```

```
  FILE_ATTRIBUTE_NORMAL,   // атрибуты по умолчанию
  NULL);                   // дополнительных атрибутов нет

// проверяем связь с каналом
if (hNamedPipe==INVALID_HANDLE_VALUE)
{
  cerr << "Connection with the named pipe failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}

// определяем состояние канала
if (!GetNamedPipeHandleState(
  hNamedPipe,            // дескриптор именованного канала
  &dwState,              // состояние именованного канала
  &dwCurInstances,       // количество экземпляров канала
  &dwMaxCollectionCount, // размер буфера клиента канала
  &dwCollectDataTimeout, // макс. задержка перед передачей данных
  chUserName,            // имя пользователя канала
  255))                  // максимальная длина имени
{
  cerr << "Get named pipe handle state failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}

// выводим состояние канала на консоль
cout << "State: ";
switch (dwState)
{
case (PIPE_NOWAIT):
  cout << "PIPE_NOWAIT" << endl;
  break;
```

```
  case (PIPE_READMODE_MESSAGE):
    cout << "PIPE_READMODE_MESSAGE" << endl;
    break;
  case (PIPE_NOWAIT | PIPE_READMODE_MESSAGE):
    cout << "PIPE_NOWAIT and PIPE_READMODE_MESSAGE" << endl;
    break;
  default:
    cout << "Unknown state." << endl;
    break;
  }

  cout << "Current instances: " << dwCurInstances << endl
    << "Max collection count: " << dwMaxCollectionCount << endl
    << "Collection data timeout: " << dwCollectDataTimeout << endl
    << "User name: " << chUserName << endl;

  // закрываем дескриптор канала
  CloseHandle(hNamedPipe);
  // завершаем процесс
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

Изменить некоторые характеристики именованного канала можно посредством функции `SetNamedPipeHandleState`, которая имеет следующий прототип:

```
BOOL  SetNamedPipeHandleState(
  HANDLE   hNamedPipe,              // дескриптор именованного канала
  LPDWORD  lpMode,                  // новый режим передачи данных
  LPDWORD  lpMaxCollectionCount,    // максимальное количество байтов
  LPDWORD  lpCollectionDataTimeout  // интервал ожидания
);
```

В случае успешного завершения функция `SetNamedPipeHandleState` возвращает ненулевое значение, а в случае неудачи — `FALSE`. Параметры функции имеют следующее назначение.

В параметре `hNamedPipe` должен быть установлен дескриптор именованного канала. Причем канал должен быть открыт в режиме записи.

Параметр lpMode должен указывать на переменную типа DWORD, которая содержит новые режимы работы именованного канала. Возможно изменить следующие режимы работы именованного канала: режим передачи данных и режим ожидания при выполнении записи или чтении данных в именованный канал, а также ожидания сервером соединения клиента с именованным каналом. Режим передачи данных может принимать следующие значения:

- PIPE_READMODE_BYTE — передача данных потоком;
- PIPE_READMODE_MESSAGE — передача данных сообщениями.

Режим ожидания может принимать следующие значения:

- PIPE_WAIT — блокирование приложения до завершения выполнения функций ConnectNamedPipe, WriteFile и ReadFile;
- PIPE_NOWAIT — выполнение функций ConnectNamedPipe, WriteFile и ReadFile не блокирует работу приложения.

Отметим, что эти режимы ожидания не влияют на работу асинхронных операций доступа к именованному каналу, а предназначены только для работы менеджера локальной сети. В параметре lpMode может быть установлена любая комбинация флагов режимов передачи данных и ожидания или только один из этих флагов. Если режимы работы именованного канала не изменяются, то параметр lpMode должен содержать значение NULL.

Параметр lpMaxCollectionCount должен указывать на переменную типа DWORD, которая содержит максимальное количество байтов, которые клиент именованного канала может записать в этот канал, прежде чем эти данные будут переданы серверу канала. Однако если клиент открыл именованный канал в режиме FILE_FLAG_WRITE_THROUGH, то параметр lpMaxCollectionCount игнорируется. Этот параметр должен быть установлен в NULL, если функция не изменяет размер максимального количества записываемых перед передачей байтов.

Параметр lpCollectionDataTimeout должен указывать на переменную типа DWORD, которая содержит новое количество миллисекунд, которые могут пройти, прежде чем данные могут быть переданы по сети. Как и в предыдущем случае, если клиент открыл именованный канал в режиме FILE_FLAG_WRITE_THROUGH, то параметр lpCollectionDataTimeout игнорируется. Этот параметр должен быть установлен в NULL, если функция не изменяет интервал ожидания до передачи данных по именованному каналу.

В листинге 16.10 приведена программа процесса-клиента именованного канала, которая изменяет состояние именованного канала посредством вызова функции SetNamedPipeHandleState.

Листинг 16.10. Пример изменения состояния именованного канала

```
#include <windows.h>
#include <iostream.h>
#include <string.h>

int main()
{
  char  machineName[80];
  char  pipeName[80];
  HANDLE   hNamedPipe;

  DWORD  dwState;                // состояние канала
  DWORD  dwCurInstances;         // количество экземпляров канала
  DWORD  dwMaxCollectionCount;   // размер буфера клиента канала
  DWORD  dwCollectDataTimeout;   // временная задержка перед передачей
данных
  TCHAR  chUserName[255];        // имя владельца именованного канала

  // вводим имя машины в сети, на которой работает сервер
  cout << "Enter a name of the server machine: ";
  cin >> machineName;
  cin.get();
  // подставляем имя машины в имя канала
  wsprintf(pipeName, "\\\\%s\\pipe\\demo_pipe", machineName);

  // связываемся с именованным каналом
  hNamedPipe = CreateFile(
    pipeName,              // имя канала
    GENERIC_READ | GENERIC_WRITE,    // читаем и записываем в канал
    FILE_SHARE_READ | FILE_SHARE_WRITE,  // разрешаем чтение и запись
    NULL,                  // безопасность по умолчанию
    OPEN_EXISTING,         // открываем существующий канал
    FILE_ATTRIBUTE_NORMAL, // атрибуты по умолчанию
    NULL);                 // дополнительных атрибутов нет

  // проверяем связь с каналом
  if (hNamedPipe==INVALID_HANDLE_VALUE)
  {
```

```
  cerr << "Connection with the named pipe failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}

// определяем состояние канала
if (!GetNamedPipeHandleState(
  hNamedPipe,              // дескриптор именованного канала
  &dwState,                // состояние именованного канала
  &dwCurInstances,         // количество экземпляров канала
  &dwMaxCollectionCount,   // размер буфера клиента канала
  &dwCollectDataTimeout,   // макс. задержка перед передачей данных
  chUserName,              // имя пользователя канала
  255))                    // максимальная длина имени
{
  cerr << "Get named pipe handle state failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}

// выводим интервал ожидания на консоль
cout << "Collection data timeout: " << dwCollectDataTimeout << endl;

// изменяем состояние именованного канала
dwCollectDataTimeout = 100;
if (!SetNamedPipeHandleState(
  hNamedPipe,  // дескриптор именованного канала
  NULL,        // режим передачи данных не изменяем
  NULL,        // размер буфера не изменяем
  &dwCollectDataTimeout))  // макс. задержка равна 100 миллисекунд
{
  cerr << "Set named pipe handle state failed." << endl
    << "The last error code: " << GetLastError() << endl;
```

```
    cout << "Press any key to exit.";
    cin.get();

    return 0;
  }

  // определяем состояние канала
  GetNamedPipeHandleState(
    hNamedPipe,         // дескриптор именованного канала
    &dwState,           // состояние именованного канала
    &dwCurInstances,    // количество экземпляров канала
    &dwMaxCollectionCount,    // размер буфера клиента канала
    &dwCollectDataTimeout,    // макс. задержка перед передачей данных
    chUserName,         // имя пользователя канала
    255);               // максимальная длина имени

  // выводим интервал ожидания на консоль
  cout << "Collection data timeout: " << dwCollectDataTimeout << endl;

  // закрываем дескриптор канала
  CloseHandle(hNamedPipe);
  // завершаем процесс
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

16.9. Получение информации об именованном канале

Для получения информации об атрибутах именованного канала, которые не могут быть изменены, используется функция `GetNamedPipeInfo`. Эта функция имеет следующий прототип:

```
BOOL  GetNamedPipeInfo (
  HANDLE   hNamedPipe,       // дескриптор именованного канала
  LPDWORD  lpFlags,          // тип канала
  LPDWORD  lpOutBufferSize,  // размер выходного буфера
```

```
    LPDWORD  lpInBufferSize,   // размер входного буфера
    LPDWORD  lpMaxInstances    // макс. количество экземпляров канала
  );
```

В случае успешного завершения функция `GetNamedPipeInfo` возвращает ненулевое значение, а в случае неудачи — `FALSE`. Опишем параметры этой функции.

В параметре `hNamedPipe` должен быть установлен дескриптор именованного канала. Причем канал должен быть открыт в режиме чтения.

Параметр `lpFlags` должен указывать на переменную типа `DWORD`, в которой установлен тип именованного канала, о котором нужно получить информацию. Тип именованного канала должен содержать информацию о способе передачи данных по каналу и указывать, кому принадлежит дескриптор: клиенту или серверу именованного канала. Для этого должны использоваться следующие константы:

- `PIPE_CLIENT_END` — дескриптор клиента именованного канала;
- `PIPE_SERVER_END` — дескриптор сервера именованного канала;
- `PIPE_TYPE_BYTE` — передача данных потоком;
- `PIPE_TYPE_MESSAGE` — передача данных сообщениями.

Если эта информация не установлена, т. е. параметр `lpFlags` имеет значение `NULL`, то функция рассматривает тот конец именованного канала, который соединен с клиентом, и по которому данные передаются потоком.

Параметр `lpOutBufferSize` должен указывать на переменную типа `DWORD`, в которую функция `GetNamedPipeInfo` поместит размер выходного буфера именованного канала. Если информация о размере выходного буфера не нужна, то значение этого параметра может быть установлено в `NULL`.

Параметр `lpInBufferSize` должен указывать на переменную типа `DWORD`, в которую функция `GetNamedPipeInfo` поместит размер входного буфера именованного канала. Если информация о размере входного буфера не нужна, то значение этого параметра может быть установлено в `NULL`.

Параметр `lpMaxInstances` должен указывать на переменную типа `DWORD`, в которую функция `GetNamedPipeInfo` поместит число, обозначающее максимальное допустимое количество экземпляров именованного канала. Если это число равно значению `PIPE_UNLIMITED_INSTANCES`, то количество экземпляров именованного канала ограничено только наличием системных ресурсов. Если информация о максимальном количестве экземпляров именованного канала не нужна, то значение этого параметра может быть установлено в `NULL`.

В листинге 16.11 приведена программа процесса-клиента именованного канала, которая получает информацию об неизменяемых атрибутах именованного канала посредством вызова функции `GetNamedPipeInfo`.

Листинг 16.11. Пример получения информации об именованном канале

```
#include <windows.h>
#include <iostream.h>
#include <string.h>

int main()
{
  char  machineName[80];
  char  pipeName[80];
  HANDLE  hNamedPipe;

  DWORD  dwFlags = PIPE_CLIENT_END |
        PIPE_TYPE_MESSAGE;    // клиент канала и передача данных сообщениями
  DWORD  dwOutBufferSize;     // состояние канала
  DWORD  dwInBufferSize;      // количество экземпляров канала
  DWORD  dwMaxInstances;      // размер буфера клиента канала

  // вводим имя машины в сети, на которой работает сервер
  cout << "Enter a name of the server machine: ";
  cin >> machineName;
  cin.get();
  // подставляем имя машины в имя канала
  wsprintf(pipeName, "\\\\%s\\pipe\\demo_pipe", machineName);

  // связываемся с именованным каналом
  hNamedPipe = CreateFile(
    pipeName,           // имя канала
    GENERIC_READ | GENERIC_WRITE,    // читаем и записываем в канал
    FILE_SHARE_READ | FILE_SHARE_WRITE,  // разрешаем чтение и запись
    NULL,                   // безопасность по умолчанию
    OPEN_EXISTING,          // открываем существующий канал
    FILE_ATTRIBUTE_NORMAL, // атрибуты по умолчанию
    NULL);                  // дополнительных атрибутов нет

  // проверяем связь с каналом
  if (hNamedPipe==INVALID_HANDLE_VALUE)
  {
```

```
    cerr << "Connection with the named pipe failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to exit.";
    cin.get();

    return 0;
  }

  // получаем информацию о канале
  if (!GetNamedPipeInfo(
    hNamedPipe,        // дескриптор именованного канала
    &dwFlags,          // тип канала
    &dwOutBufferSize,  // размер выходного буфера
    &dwInBufferSize,   // размер входного буфера
    &dwMaxInstances))  // максимальное количество экземпляров канала
  {
    cerr << "Get named pipe info failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to exit.";
    cin.get();

    return 0;
  }

  // выводим информацию на консоль
  cout << "Out buffer size: " << dwOutBufferSize << endl
    << "In buffer size: " << dwInBufferSize << endl
    << "Max instances: " << dwMaxInstances << endl;

  // закрываем дескриптор канала
  CloseHandle(hNamedPipe);
  // завершаем процесс
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

Глава 17

Работа с почтовыми ящиками в Windows

17.1. Концепция почтовых ящиков

Почтовым ящиком называется объект ядра операционной системы, который обеспечивает передачу сообщений от процессов-клиентов к процессам-серверам, выполняющимся на компьютерах в пределах локальной сети. Процесс, который создает почтовый ящик, называется *сервером почтового ящика*. Процессы, которые связываются с именованным почтовым ящиком, называются *клиентами почтового ящика*. Перечислим характеристики почтовых ящиков:

- имеют имя, которое используется клиентами для связи с почтовыми ящиками;
- направление передачи данных от клиента к серверу;
- передача данных осуществляется сообщениями;
- обмен данными может быть как синхронным, так и асинхронным.

Хотя передача данных осуществляется только от клиента к серверу, один почтовый ящик может иметь несколько серверов. Это происходит в том случае, если несколько серверов создают почтовые ящики с одинаковыми именами. Тогда все сообщения, которые посылает клиент в такой почтовый ящик, будут получать все серверы этого почтового ящика. Однако это выполняется только при условии, что длина сообщения меньше 425 байт, так как в этом случае сообщения передаются дейтаграммами. Таким образом, можно сказать, что почтовые ящики обеспечивают однонаправленную связь типа "многие-ко-многим". При этом доставка сообщения от клиента к серверам почтового ящика не подтверждается системой. Заметим также, что операционные системы семейства Windows NT не поддерживают передачу сообщений длиной 425 и 426 байт.

Немного коснемся вопроса передачи сообщений посредством почтовых ящиков. Если длина сообщения меньше чем 425 байт, то такое сообщение передается как дейтаграмма. Дейтаграмма представляет собой небольшой

пакет с передаваемым по сети сообщением, который содержит также информацию об отправителе и получателе сообщения. Дейтаграмма рассылается всем серверам данного почтового ящика. Так как размер пакета небольшой, то дейтаграммы рассылаются быстро, но нет гарантии доставки сообщения, так как в дейтаграмме не хранится информация, поддерживающая контроль доставки. Если же длина сообщения больше 426 байт, то такие сообщения могут передаваться только от одного клиента к одному серверу, используя при этом SMB (Server Message Block) протокол передачи данных по сети. При этом отметим, что длина сообщения, передаваемого в почтовый ящик, не может превышать 64 Кбайт. Кроме того, при использовании протокола SMB сообщения передаются только от одного клиента к одному серверу почтового ящика.

Почтовые ящики можно рассматривать как псевдофайлы, расположенные в оперативной памяти компьютера. Поэтому для доступа к почтовым ящикам используются те же функции, что и для доступа к обычным файлам. Подробно эти функции рассмотрены в *гл. 24*, посвященной работе с файлами.

Теперь приведем порядок работы с почтовыми ящиками, который и будет использоваться в дальнейшем.

1. Создание почтового ящика сервером.
2. Соединение клиента с почтовым ящиком.
3. Обмен данными через почтовый ящик.
4. Закрытие почтового ящика клиентом и сервером.

Подробно эти пункты работы с почтовыми ящиками рассмотрены в следующих разделах.

17.2. Создание почтовых ящиков

Создаются почтовые ящики процессом-сервером при помощи функции `CreateMailslot`, которая имеет следующий прототип:

```
HANDLE  CreateMailslot(
  LPCTSTR  lpName,           // имя почтового ящика
  DWORD    dwMaxMessageSize  // максимальная длина сообщения
  DWORD    dwReadTimeout,    // интервал ожидания
  LPSECURITY_ATTRIBUTES  lpSecurityAttributes  // атрибуты безопасности
);
```

В случае успешного завершения эта функция вернет дескриптор почтового ящика, а в случае неудачи — значение `INVALID_HANDLE_VALUE`. Опишем параметры функции.

Параметр `lpName` указывает на строку, которая должна иметь вид:

```
\\.\mailslot\mailslot_name
```

Здесь символ "." обозначает локальную машину, так как новый почтовый ящик всегда создается на локальной машине, слово `mailslot` — фиксировано, а `mailslot_name` обозначает имя почтового ящика, которое задается пользователем и нечувствительно к верхнему и нижнему регистрам.

Параметр `dwMaxMessageSize` задает максимальную длину сообщения в байтах, которое может быть записано в почтовый ящик.

Параметр `dwReadTimeout` задает в миллисекундах временной интервал, в течение которого функция `ReadFile` ждет поступления сообщения в почтовый ящик. Если в этом параметре установлено значение 0, то в случае отсутствия в почтовом ящике сообщения функция немедленно возвращает управление. Для задания бесконечного времени ожидания в этом параметре нужно установить значение `MAILSLOT_WAIT_FOREVER`.

Заметим, что несколько процессов могут создать почтовые ящики с одним и тем же именем. В этом случае сообщение, посланное клиентом, может доставляться не только в почтовый ящик одного процесса, а также в почтовые ящики всех таких процессов при условии, что они работают на компьютерах внутри одного домена. Режим доставки сообщений зависит от режима открытия почтового ящика клиентом.

Пример создания почтового ящика сервером будет показан в *разд. 17.4*, посвященном обмену данными через почтовый ящик.

17.3. Соединение клиентов с почтовым ящиком

Как уже говорилось, для установления связи с почтовым ящиком клиент использует функцию `CreateFile`, которая имеет следующий прототип:

```
HANDLE  CreateFile(
  LPCTSTR  lpFileName,       // указатель на имя почтового ящика
  DWORD    dwDesiredAccess,  // чтение или запись в канал
  DWORD    dwShareMode,      // режим совместного использования
  LPSECURITY_ATTRIBUTES  lpSecurityAttributes, // атрибуты безопасности
  DWORD    dwCreationDisposition,  // флаги открытия почтового ящика
  DWORD    dwFlagsAndAttributes,   // флаги и атрибуты
  HANDLE   hTemplateFile           // дополнительные атрибуты
);
```

В случае успешного завершения эта функция возвращает дескриптор почтового ящика, а в случае неудачи — значение `INVALID_HANDLE_VALUE`.

Параметры функции `CreateFile` могут принимать следующие значения (если эта функция используется для открытия почтового ящика).

Параметр `lpFileName` должен указывать на имя почтового ящика, которое может быть задано в одном из следующих форматов:

- почтовый ящик на локальном компьютере:

  ```
  \\.\mailslot\имя_почтового_ящика
  ```

- почтовый ящик на компьютере с указанным именем:

  ```
  \\имя_компьтера\mailslot\имя_почтового_ящика
  ```

- почтовый ящик в домене с указанным именем:

  ```
  \\имя_домена\mailslot\имя_почтового_ящика
  ```

- почтовый ящик в первичном домене системы:

  ```
  \\*\mailslot\имя_почтового_ящика
  ```

В первом случае сообщения будут доставляться только в почтовые ящики с заданным именем, которые расположены на локальной машине. Во втором случае сообщения будут доставляться в почтовые ящики с заданным именем, расположенные на компьютере с указанным именем. В третьем случае сообщения будут доставляться в почтовые ящики с заданным именем, которые созданы внутри домена с указанным именем. В четвертом случае сообщения будут доставляться в почтовые ящики с заданным именем, которые созданы внутри первичного домена системы.

Параметр `dwDesiredAccess` должен иметь значение `GENERIC_WRITE`, которое разрешает запись в почтовый ящик.

Параметр `dwShareMode` определяет режим совместного использования почтового ящика и может принимать любую комбинацию из следующих значений:

- `FILE_SHARE_READ` — разрешает совместное чтение из почтового ящика;
- `FILE_SHARE_WRITE` — разрешает совместную запись в почтовый ящик.

Параметр `lpSecurityAttributes` задает атрибуты безопасности почтового ящика. Пока этот параметр будем устанавливать в `NULL`.

Для почтового ящика параметр `dwCreationDisposition` должен быть равен значению `OPEN_EXISTING`, т. к. клиент всегда открывает существующий почтовый ящик.

Для почтового ящика параметр `dwFlagsAndAttributes` можно задать равным 0, что определяет флаги и атрибуты по умолчанию, или установить в этом параметре значение `FILE_ATTRIBUTE_NORMAL`.

Параметр `hTemplateFile` при работе с почтовыми ящиками не используется, поэтому в нем устанавливается значение `NULL`.

Пример соединения клиента с почтовым ящиком приводится в следующем разделе, посвященном обмену данными через почтовый ящик.

17.4. Обмен данными через почтовый ящик

Для обмена данными через почтовый ящик используются обычные функции доступа к файлу `WriteFile` и `ReadFile`. Процесс-клиент записывает данные в почтовый ящик при помощи функций `WriteFile`, а процесс-сервер читает данные из почтового ящика, используя функции `ReadFile`. Подробно эти функции рассмотрены в *гл. 24*, посвященной работе с файлами в Windows. В этом разделе приведем только программы, которые обмениваются данными через почтовый ящик. Конечно, процесс-сервер также может записывать сообщения в почтовый ящик, но это имеет смысл лишь в том случае, если в домене расположено несколько почтовых ящиков с одинаковыми именами, которые были созданы разными процессами. Тогда одинаковые сообщения получают все почтовые ящики с одинаковыми именами.

Сначала, в листинге 17.1, приведем программу процесса-сервера, который создает почтовый ящик. В этой программе почтовый ящик создается на локальной машине.

Листинг 17.1. Процесс-сервер почтового ящика

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hMailslot;       // дескриптор почтового ящика

  // создаем почтовый ящик
  hMailslot = CreateMailslot(
    "\\\\.\\mailslot\\demo_mailslot",  // имя почтового ящика
    0,       // длина сообщения произвольна
    MAILSLOT_WAIT_FOREVER,   // ждем сообщения произвольно долго
    NULL     // безопасность по умолчанию
    );
  // проверяем на успешное создание
  if (hMailslot == INVALID_HANDLE_VALUE)
  {
    cerr << "Create mailslot failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish server.";
```

```
    cin.get();

    return 0;
  }

  cout << "The mailslot is created." << endl;

  cout << "The mailslot is waiting a message." << endl;
  // читаем одно целое число из почтового ящика
  int nData;
  DWORD dwBytesRead;
  if (!ReadFile(
    hMailslot,       // дескриптор почтового ящика
    &nData,          // адрес буфера для ввода данных
    sizeof(nData),   // количество читаемых байтов
    &dwBytesRead,    // количество прочитанных байтов
    (LPOVERLAPPED)NULL   // передача данных синхронная
    ))
  {
    cerr << "Read file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hMailslot);
    cout << "Press any key to finish server.";
    cin.get();

    return 0;
  }

  // выводим число на консоль
  cout << "The number " << nData << " was read by the server" << endl;

  // закрываем дескриптор почтового ящика
  CloseHandle(hMailslot);

  // завершаем процесс
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

Теперь, в листинге 17.2, приведем программу процесса-клиента почтового ящика. В этой программе предполагается, что процесс-клиент запускается на той же локальной машине, что и процесс-сервер. Иначе в имени почтового ящика нужно указать не точку, а имя компьютера, на котором создан почтовый ящик, или имя домена, в котором созданы почтовые ящики с одинаковыми именами.

Листинг 17.2. Процесс-клиент почтового ящика

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hMailslot;
  char mailslotName[] = "\\\\.\\mailslot\\demo_mailslot";

  // связываемся с почтовым ящиком
  hMailslot = CreateFile(
    mailslotName,      // имя почтового ящика
    GENERIC_WRITE,     // записываем в ящик
    FILE_SHARE_READ,   // разрешаем одновременное чтение из ящика
    NULL,              // защита по умолчанию
    OPEN_EXISTING,     // открываем существующий канал
    0,          // атрибуты по умолчанию
    NULL        // дополнительных атрибутов нет
  );

  // проверяем связь с почтовым ящиком
  if (hMailslot == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish the client.";
    cin.get();

    return 0;
  }

  // вводим целое число
```

```
  int n;
  cout << "Input an integer: ";
  cin >> n;

  // пишем число в почтовый ящик
  DWORD dwBytesWritten;
  if (!WriteFile(
      hMailslot,   // дескриптор почтового ящика
      &n,          // данные
      sizeof(n),   // размер данных
      &dwBytesWritten,   // количество записанных байтов
      NULL         // синхронная запись
    ))
  {
    // ошибка записи
    cerr << "Write file failed: " << endl
    << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish the client.";
    cin.get();

    CloseHandle(hMailslot);
    return 0;
  }

  // закрываем дескриптор канала
  CloseHandle(hMailslot);
  // завершаем процесс
  cout << "The number is written by the client." << endl
    << "Press any key to exit." << endl;
  cin.get();

  return 0;
}
```

В заключение этого раздела отметим, что, т. к. доступ к почтовым ящикам не отличается от доступа к файлам, для обмена данными через почтовые ящики можно также использовать функции `WriteFileEx` и `ReadFileEx`.

17.5. Получение информации о почтовом ящике

Для получения информации о характеристиках почтового ящика используется функция GetMailslotInfo, которая имеет следующий прототип:

```
BOOL  GetMailslotInfo(
  HANDLE   hMailslot,         // дескриптор почтового ящика
  LPDWORD  lpMaxMessageSize   // максимальная длина сообщения
  LPDWORD  lpNextSize,        // длина следующего сообщения
  LPDWORD  lpMessageCount     // количество сообщений
  LPDWORD  lpReadTimeout      // интервал ожидания сообщения
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — NULL. Параметры функции GetMailslotInfo имеют следующее назначение.

В параметре hMailslot должен быть установлен дескриптор почтового ящика, который был возвращен функцией CreateMailslot.

Параметр lpMaxMessageSize должен указывать на переменную типа DWORD, в которую функция GetMailslotInfo поместит максимальную длину сообщения, которое может быть записано в почтовый ящик. Если это значение не нужно, то этот параметр может быть установлен в NULL.

Параметр lpNextSize должен указывать на переменную типа DWORD, в которую функция GetMailslotInfo поместит длину следующего сообщения в почтовом ящике. Если в почтовом ящике нет сообщений, то в этот параметр функция запишет значение MAILSLOT_NO_MESSAGE. Если значение длины последнего сообщения не нужно, то параметр lpNextSize может быть установлен в NULL.

Параметр lpMessageCount должен указывать на переменную типа DWORD, в которую функция GetMailslotInfo поместит количество сообщений, находящихся в почтовом ящике. Если это значение не нужно, то этот параметр может быть установлен в NULL.

Параметр lpReadTimeout должен указывать на переменную типа DWORD, в которую функция GetMailslotInfo поместит целое число без знака, обозначающее временной интервал в миллисекундах. Если почтовый ящик пуст, то в течение этого интервала функция ReadFile будет ждать, пока процесс-клиент не запишет сообщение в почтовый ящик. Если это значение не нужно, то в этом параметре может быть установлено значение NULL.

В листингах 17.3, 17.4 приведены программы, в которых функция GetMailslotInfo используется для получения информации о сообщениях,

хранящихся в почтовом ящике. Сначала приведем программу сервера почтового ящика, который читает сообщения по мере их поступления от клиента.

Листинг 17.3. Процесс-сервер почтового ящика

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hMailslot;          // дескриптор почтового ящика
  DWORD  dwNextMessageSize;  // размер следующего сообщения
  DWORD  dwMessageCount;      // количество сообщений

  // создаем почтовый ящик
  hMailslot = CreateMailslot(
    "\\\\.\\mailslot\\demo_mailslot",    // имя почтового ящика
    0,        // длина сообщения произвольна
    MAILSLOT_WAIT_FOREVER,   // ждем сообщения произвольно долго
    NULL      // безопасность по умолчанию
    );
  // проверяем на успешное создание
  if (hMailslot == INVALID_HANDLE_VALUE)
  {
    cerr << "Create mailslot failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish server.";
    cin.get();

    return 0;
  }

  cout << "The mailslot is created." << endl;

  // ждем сообщений
  cout << "Press any key to read messages." << endl;
  cin.get();

  // получаем информацию о почтовом ящике
```

```
if (!GetMailslotInfo(
  hMailslot,   // дескриптор почтового ящика
  NULL,       // максимальный размер сообщения не нужен
  &dwNextMessageSize,  // размер следующего сообщения
  &dwMessageCount,     // количество сообщений
  NULL))       // интервал ожидания не нужен
{
  cerr << "Get mailslot info failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish server.";
  cin.get();

  return 0;
}

// читаем сообщения
while (dwMessageCount != 0)
{
  DWORD dwBytesRead;
  char*  pchMessage;

  // захватываем память для сообщения
  pchMessage = (char*) new char[dwNextMessageSize];

  // читаем одно сообщение
  if (!ReadFile(
    hMailslot,       // дескриптор канала
    pchMessage,      // адрес буфера для ввода данных
    dwNextMessageSize,   // количество читаемых байтов
    &dwBytesRead,        // количество прочитанных байтов
    NULL             // передача данных синхронная
    ))
  {
    cerr << "Read file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hMailslot);
    cout << "Press any key to finish server.";
    cin.get();
```

```
      return 0;
    }

    // выводим сообщение на консоль
    cout << "The message << " << pchMessage << " >> was read" << endl;

    // получаем информацию о следующем сообщении
    if (!GetMailslotInfo(
      hMailslot,   // дескриптор почтового ящика
      NULL,        // максимальный размер сообщения не нужен
      &dwNextMessageSize,  // размер следующего сообщения
      &dwMessageCount,     // количество сообщений
      NULL))       // интервал ожидания не нужен
    {
      cerr << "Get mailslot info failed." << endl
        << "The last error code: " << GetLastError() << endl;
      cout << "Press any key to finish server.";
      cin.get();

      return 0;
    }

    // освобождаем память для сообщения
    delete[] pchMessage;
  }

  // закрываем дескриптор почтового ящика
  CloseHandle(hMailslot);

  // завершаем процесс
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

Теперь, в листинге 17.4, приведем программу клиента почтового ящика, которая посылает сообщения процессу-серверу. Отметим, что как процесс-сервер, так и процесс-клиент находятся на локальном компьютере.

Листинг 17.4. Процесс-клиент почтового ящика

```
#include <windows.h>
#include <iostream.h>
#include <string.h>

int main()
{
  HANDLE  hMailslot;
  char  mailslotName[] = "\\\\.\\mailslot\\demo_mailslot";

  // связываемся с почтовым ящиком
  hMailslot = CreateFile(
    mailslotName,      // имя почтового ящика
    GENERIC_WRITE,     // записываем в ящик
    FILE_SHARE_READ,   // разрешаем одновременное чтение из ящика
    NULL,              // защита по умолчанию
    OPEN_EXISTING,     // открываем существующий канал
    0,                 // атрибуты по умолчанию
    NULL               // дополнительных атрибутов нет
    );

  // проверяем связь с почтовым ящиком
  if (hMailslot == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish the client.";
    cin.get();

    return 0;
  }

  // вводим количество передаваемых сообщений
  int  n;
  cout << "Input a number of messages: ";
  cin >> n;
  cin.get();
```

```
  // пишем сообщения в почтовый ящик
for (int i = 0; i < n; ++i)
{
  DWORD  dwBytesWritten;
  char  pchMessage[256];
  int    nMessageSize;

  cout << "Input message: ";
  // читаем сообщение
  cin.getline(pchMessage, 256);
  // определем длину сообщения
  nMessageSize = strlen(pchMessage) + 1;

  // пишем сообщение
  if (!WriteFile(
    hMailslot,       // дескриптор почтового ящика
    pchMessage,      // данные
    nMessageSize,    // размер данных
    &dwBytesWritten, // количество записанных байтов
    NULL             // синхронная запись
    ))
  {
    // ошибка записи
    cerr << "Write file failed: " << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish the client.";
    cin.get();

    CloseHandle(hMailslot);

    return 0;
  }
}

// закрываем дескриптор канала
CloseHandle(hMailslot);
// завершаем процесс
```

```
  cout << "The messages are written by the client." << endl
    << "Press any key to exit." << endl;
  cin.get();

  return 0;
}
```

17.6. Изменение времени ожидания сообщения

Для изменения времени ожидания сервером сообщения от клиента используется функция `SetMailslotInfo`, которая имеет следующий прототип:

```
BOOL  SetMailslotInfo(
  HANDLE  hMailslot,       // дескриптор почтового ящика
  DWORD   dwReadTimeout    // интервал ожидания сообщения
);
```

В случае успешного завершения эта функция возвратит ненулевое значение, а в случае неудачи — `NULL`.

В параметре `hMailslot` должен быть установлен дескриптор почтового ящика, который был получен вызовом функции `CreateMailslot`.

Параметр `dwReadTimeout` задает в миллисекундах новый временной интервал, в течение которого функция `ReadFile` ждет поступления сообщения в почтовый ящик. Если в этом параметре устанавливается значение 0, то в случае отсутствия в почтовом ящике сообщения функция немедленно возвращает управление. Для задания бесконечного времени ожидания в этом параметре нужно установить значение `MAILSLOT_WAIT_FOREVER`.

В листинге 17.5 приведена программа, в которой функция `SetMailslotInfo` используется для установки нового временного интервала для ожидания поступления сообщения в почтовый ящик.

Листинг 17.5. Изменение времени ожидания сообщения

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hMailslot;     // дескриптор почтового ящика
```

```
DWORD  dwReadTimeout;  // интервал для ожидания сообщения

// создаем почтовый ящик
hMailslot = CreateMailslot(
  "\\\\.\\mailslot\\demo_mailslot",  // имя почтового ящика
  0,     // длина сообщения произвольна
  0,     // интервал ожидания равен нулю
  NULL   // защита по умолчанию
  );
// проверяем на успешное создание
if (hMailslot == INVALID_HANDLE_VALUE)
{
  cerr << "Create mailslot failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish server.";
  cin.get();

  return 0;
}

cout << "The mailslot is created." << endl;

// получаем информацию о почтовом ящике
if (!GetMailslotInfo(
  hMailslot,  // дескриптор почтового ящика
  NULL,       // максимальный размер сообщения не нужен
  NULL,       // размер следующего сообщения не нужен
  NULL,       // количество сообщений не нужно
  &dwReadTimeout))   // интервал ожидания сообщения
{
  cerr << "Get mailslot info failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish server.";
  cin.get();

  return 0;
}

cout << "Read timeout: " << dwReadTimeout << endl;
```

```
  if (!SetMailslotInfo(
    hMailslot,   // дескриптор почтового ящика
    3000))       // изменяем интервал ожидания
  {
    cerr << "Set mailslot info failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish server.";
    cin.get();

    return 0;
  }

  // получаем информацию о почтовом ящике
  if (!GetMailslotInfo(
    hMailslot,   // дескриптор почтового ящика
    NULL,       // максимальный размер сообщения не нужен
    NULL,       // размер следующего сообщения не нужен
    NULL,       // количество сообщений не нужно
    &dwReadTimeout))   // интервал ожидания сообщения
  {
    cerr << "Get mailslot info failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish server.";
    cin.get();

    return 0;
  }

  cout << "Read timeout: " << dwReadTimeout << endl;

  // закрываем дескриптор почтового ящика
  CloseHandle(hMailslot);

  // завершаем процесс
  cout << "Press any key to exit.";
  cin.get();

  return 0;
}
```

Часть V

Структурная обработка исключений

Глава 18

Фреймовая обработка исключений

18.1. Исключения и их обработчики

Исключением называется событие, которое произошло во время выполнения программы, в результате совершения которого дальнейшее нормальное выполнение программы становится невозможным. Как правило, такие события являются ошибками в программе. Поэтому для дальнейшей работы приложения требуется или восстановление программы в рабочее состояние, или ее аварийное завершение с освобождением всех захваченных программой ресурсов.

В операционных системах Windows для этой цели предназначен механизм *структурной обработки исключений* (structured exception handling, SEH). Смысл механизма структурной обработки исключений заключается в следующем. В программе выделяется блок программного кода, в котором может произойти исключение. Такой блок кода называется *фреймом*, а сам код называется *охраняемым кодом*. Затем, после фрейма вставляется программный блок, который обрабатывает происшедшее исключение. Этот блок называется *обработчиком исключения*. После обработки исключения управление передается первой инструкции, следующей за обработчиком исключения.

Очевидно, что для того чтобы использовать этот механизм в программе, в язык программирования C++ нужно ввести новые ключевые слова. Такими ключевыми словами являются `__try` и `__except`, которые расширяют список стандартных ключевых слов языка программирования C++ и различаются только компилятором фирмы Microsoft. Ключевое слово `__try` отмечает фрейм, а ключевое слово `__except` отмечает обработчик исключения. В результате фрагмент программы, который использует механизм структурной обработки исключений, выглядит следующим образом:

```
__try
{
  // охраняемый код
```

```
}
__except (выражение-фильтр)
{
  // код обработки исключения
}
```

Здесь `выражение-фильтр` является выражением языка программирования C++ и указывает на то, как должна выполняться программа после обработки исключения. Вычисление этого выражения выполняется сразу после возникновения исключения и должно давать в результате одно из следующих значений:

- `EXCEPTION_EXECUTE_HANDLER` — управление передается обработчику исключений;
- `EXCEPTION_CONTINUE_SEARCH` — система продолжает поиск обработчика исключения;
- `EXCEPTION_CONTINUE_EXECUTION` — система передает управление в точку прерывания программы.

Сделаем два важных замечания относительно механизма структурной обработки исключений. Во-первых, не допускается использование оператора `goto` для передачи управления внутрь фрейма или обработчика исключения. Во-вторых, в выражении фильтра допускается использование функций `GetExceptionCode` и `GetExceptionInformation`, которые предоставляют информацию о происшедшем исключении. Эти функции будут рассмотрены в *разд. 18.3*.

Отметим также, что переменные, объявленные внутри фрейма или блока обработки исключения, являются локальными и видны только внутри соответствующего блока, как это и принято в языке программирования C++.

Рассмотрим, в листинге 18.1, программу, в которой происходит исключение при попытке обращения к памяти по адресу, который не инициализирован. Так как значение выражения-фильтра равно `EXCEPTION_EXECUTE_HANDLER`, то управление передается обработчику исключений, который выводит на консоль сообщение об исключении и инициализирует этот указатель. После этого программа выводит на консоль нужное значение числа и заканчивает свое выполнение.

Листинг 18.1. Обработка исключения

```
#include <windows.h>
#include <iostream.h>

int main()
```

```
{
  int  a = 10;
  int  *p = NULL;    // пустой указатель на целое число

  __try
  {
    cout << "a = " << *p << endl;  // ошибка, так как p = NULL
  }
  __except(EXCEPTION_EXECUTE_HANDLER)
  {
    cout << "There was some exception." << endl;
    p = &a;
  }

  cout << "a = " << *p << endl;    // нормально

  return 0;
}
```

Остальные варианты обработки исключений в зависимости от значения выражения-фильтра будут в *разд. 18.2*, *18.3*. А в заключение этого параграфа сделаем несколько замечаний относительно механизма структурной обработки исключений и механизма обработки исключений в языке программирования C++. Концептуально механизм структурной обработки исключений в Windows немного отличается от механизма обработки исключений, принятого в языке программирования C++. Дело в том, что механизм структурной обработки исключений был разработан раньше, чем принят стандарт языка C++. Кроме того, в отличие от языка программирования C++ механизм структурной обработки исключений ориентирован не только на обработку программных исключений, но и на обработку аппаратных исключений. В SEH исключение рассматривается как ошибка, происшедшая при выполнении программы. В языке программирования C++ используется более абстрактный подход и исключение рассматривается как объект произвольного типа, который может выбросить программа, используя оператор `throw`. В свою очередь обработчик исключения `catch` может рассматриваться как функция с одним параметром, которая выполняется только в том случае, если тип ее параметра соответствует типу выброшенного исключения. Другие отличия касаются раскрутки стека и будут рассмотрены в *разд. 18.9*.

18.2. Получение кода исключения

Получить код происшедшего исключения можно при помощи функции `GetExceptionCode`, которая имеет следующий прототип:

```
DWORD  GetExceptionCode(VOID);
```

Функция `GetExceptionCode` возвращает одно из следующих значений:

- `EXCEPTION_ACCESS_VIOLATION` — попытка чтения или записи в виртуальную память без соответствующего права доступа;
- `EXCEPTION_BREAKPOINT` — встретилась точка останова;
- `EXCEPTION_DATATYPE_MISALIGNMENT` — доступ к данным, адрес которых не выровнен по границе слова или двойного слова;
- `EXCEPTION_SINGLE_STEP` — механизм трассировки программы сообщает, что выполнена одна инструкция;
- `EXCEPTION_ARRAY_BIUNDS_EXCEEDED` — выход за пределы массива, если аппаратное обеспечение поддерживает такую проверку;
- `EXCEPTION_FLT_DENORMAL_OPERAND` — один из операндов с плавающей точкой является ненормализованным;
- `EXCEPTION_FLT_DIVIDE_BY_ZERO` — попытка деления на ноль в операции с плавающей точкой;
- `EXCEPTION_FLT_INEXACT_RESULT` — результат операции с плавающей точкой не может быть точно представлен десятичной дробью;
- `EXCEPTION_FLT_INVALID_OPERATION` — ошибка в операции с плавающей точкой, для которой не предусмотрены другие коды исключения;
- `EXCEPTION_FLT_OVERFLOW` — при выполнении операции с плавающей точкой произошло переполнение;
- `EXCEPTION_FLT_STACK_CHECK` — переполнение или выход за нижнюю границу стека при выполнении операции с плавающей точкой;
- `EXCEPTION_FLT_UNDERFLOW` — результат операции с плавающей точкой является числом, которое меньше минимально возможного числа с плавающей точкой;
- `EXCEPTION_INT_DIVIDE_BY_ZERO` — попытка деления на ноль в операции с целыми числами;
- `EXCEPTION_INT_OVERFLOW` — при выполнении операции с целыми числами произошло переполнение;
- `EXCEPTION_PRIV_INSTRUCTION` — попытка выполнения привилегированной инструкции процессора, которая недопустима в текущем режиме процессора;

- `EXCEPTION_NONCONTINUABLE_EXCEPTION` — попытка возобновления исполнения программы после исключения, которое запрещает выполнять такое действие.

Отметим, что функция `GetExceptionCode` может вызываться только в выражении-фильтре или в блоке обработки исключения. Следовательно, эта функция вызывается всегда только в том случае, если исключение произошло. Отсюда можно определить назначение функции `GetExceptionCode`. Если эта функция вызывается в выражении фильтра, то она используется для того, чтобы определить выполняет ли текущий обработчик исключения обработку исключений с данным кодом или нужно продолжить поиск подходящего обработчика исключения. Если же функция `GetExceptionCode` вызывается в блоке обработки исключения, то она также предназначена для проверки кодов исключений, которые обрабатывает текущий обработчик исключения, но в этом случае поиск другого обработчика исключений не выполняется.

В листинге 18.2 приведена программа, в которой функция `GetExceptionCode` вызывается в выражении фильтра и используется для проверки того, что нужно ли передавать управление текущему блоку обработки исключений или продолжить поиск другого обработчика.

Листинг 18.2. Проверка кода исключения в выражении-фильтре

```
#include <windows.h>
#include <iostream.h>

int main()
{
  int  a = 10;
  int  *p = NULL;

  __try
  {
    cout << "a = " << *p << endl;  // ошибка, так как p = NULL
  }
  __except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
  {
    cout << "Exception access violation." << endl;
    p = &a;
  }

  cout << "a = " << *p << endl;     // нормально

  return 0;
}
```

Теперь приведем в листинге 18.3 программу, в которой проверка кода исключения используется в обработчике исключений.

Листинг 18.3. Проверка кода исключения в обработчике исключений

```
#include <windows.h>
#include <iostream.h>

int main()
{
  int  a = 10;
  int  *p = NULL;    // пустой указатель на целое число

  __try
  {
    cout << "a = " << *p << endl;  // ошибка, так как p = NULL
  }
  __except(EXCEPTION_EXECUTE_HANDLER)
  {
    DWORD  ec = GetExceptionCode();    // получаем код исключения

    if (ec == EXCEPTION_ACCESS_VIOLATION)
      cout << "Exception access violation." << endl;
    else
      cout << "Some other exception." << endl;

    p = &a;
  }

  cout << "a = " << *p << endl;    // нормально

  return 0;
}
```

18.3. Функции фильтра

Если для принятия решения об обработке исключения требуется более детально обработать информацию об исключении, то в выражении-фильтре используют функцию, которая в этом случае называется функцией фильтра.

В функции фильтра не разрешается вызывать функции `GetExceptionCode` и `GetExceptionInformation`, однако эти функции могут вызываться для инициализации параметров функции фильтра.

В листинге 18.4 приведена программа, в которой используется функция фильтра для принятия решения о дальнейшей обработке исключения. В этой программе функция фильтра возвращает одно из двух значений `EXCEPTION_CONTINUE_EXECUTION` или `EXCEPTION_EXECUTE_HANDLER`. Первое значение возвращается в случае исключения, которое генерируется системой при целочисленном делении на ноль, а второе — в остальных случаях.

Листинг 18.4. Функция фильтра и возобновление исполнения программы

```
#include <windows.h>
#include <iostream.h>


DWORD filter_function(DWORD  ec, int &a)
{
    // проверяем код исключения
  if (ec == EXCEPTION_INT_DIVIDE_BY_ZERO)
  {
    cout << "Integer divide by zero exception." << endl;
    cout << "a = " << a << endl;

      // восстанавливаем ошибку
    a = 10;

      // возобновляем выполнение программы
    cout << "Continue execution." << endl;
    cout << "a = " << a << endl;

    return EXCEPTION_CONTINUE_EXECUTION;
  }

  else
      // прекращаем выполнение программы
    return EXCEPTION_EXECUTE_HANDLER;
}

int main()
```

```
{
  int a = 0;
  int b = 1000;

  __try
  {
    b /= a;
    cout << "b = " << b << endl;
  }
  __except(filter_function(GetExceptionCode(), a))
  {
    cout << "There was some exception." << endl;
  }

  return 0;
}
```

Отметим, что в общем случае возобновление программы выполнить довольно сложно, т .к. процессор повторяет попытку выполнения программы с машинной команды, на которой произошло исключение. Естественно, что эта машинная команда не обязательно соответствует инструкции на языке высокого уровня, которая вызвала исключение.

18.4. Получение информации об исключении

Более подробную информацию об исключении можно получить при помощи вызова функции `GetExceptionInformation`, которая имеет следующий прототип:

```
LPEXCEPTION_POINTERS  GetExceptionInformation(VOID);
```

Эта функция возвращает указатель на структуру типа:

```
typedef  struct  EXCEPTION_POINTERS {
  PEXEPTION_RECORD   ExceptionRecord;
  PCONTEXT    Context;
}  EXCEPTION_POINTERS,  *PEXCEPTION_POINTERS;
```

которая, в свою очередь, содержит два указателя: `ExceptionRecord` и `Context` на структуры типа `EXCEPTION_RECORD` и `CONTEXT` соответственно.

В структуру типа `CONTEXT` система записывает содержимое всех регистров процессора на момент исключения. Эта структура имеет довольно громоздкое описание, которое можно найти в заголовочном файле WinNt.h.

Структура типа EXCEPTION_RECORD имеет следующий формат:

```
typedef  struct  EXCEPTION_RECORD {
  DWORD   ExceptionCode;
  DWORD   ExceptionFlags,
  struct  _EXCEPTION_RECORD  *ExceptionRecord;
  PVOID   ExceptionAddress,
  DWORD   NumberParameters,
  ULONG_PTR  ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
}  EXCEPTION_RECORD,  *PEXCEPTION_RECORD;
```

В нее система записывает информацию об исключении. Поля этой структуры имеют следующее назначение.

Поле ExceptionCode содержит код исключения, который может принимать такие же значения, как и код исключения, возвращаемый функцией GetExceptionCode.

Поле ExceptionFlags может принимать одно из двух значений:

- 0 — которое обозначает, что после обработки исключения возможно возобновление выполнения программы;
- EXCEPTION_NONCONTINUABLE — которое обозначает, что после обработки исключения возобновление выполнения программы невозможно.

Если установлено значение EXCEPTION_NONCONTINUABLE и выполнена попытка возобновления выполнения программы, то система выбросит исключение EXCEPTION_NONCONTINUABLE_EXCEPTION.

Поле ExceptionRecord содержит указатель на следующую структуру типа EXCEPTION_RECORD, которая может быть создана в случае вложенных исключений.

Поле ExceptionAddress содержит адрес инструкции в программе, на которой произошло исключение.

Поле NumberParameters содержит количество параметров, заданных в поле ExceptionInformation, которое является последним в этой структуре.

Поле ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS] определяет массив 32-битных аргументов, которые описывают исключение. Элементы этого массива могут использоваться функцией генерации программных исключений RaiseException. Отметим, что в операционных системах Windows NT/2000 для исключения с кодом EXCEPTION_ACCESS_VIOLATION определены значения первых двух элементов этого массива. В этом случае первый элемент массива содержит одно из значений:

- 0 — попытка чтения виртуальной памяти;
- 1 — попытка записи в виртуальную память.

А второй элемент массива содержит адрес виртуальной памяти, по которому программа пыталась прочитать или записать данные.

Сделаем важное замечание о том, что функция GetExceptionInformation может вызываться только в выражении фильтра. Поэтому эта функция вызывается всегда только в том случае, если исключение произошло. Кроме того, структуры типа EXCEPTION_POINTERS, EXCEPTION_RECORD и CONTEXT действительны только на время вычисления выражения-фильтра. Чтобы использовать содержимое структур типа EXCEPTION_RECORD и CONTEXT в блоке обработки исключения, его нужно сохранить в объявленных в программе переменных такого же типа. Как видно из описания структуры EXCEPTION_RECORD, функцию GetExceptionInformation можно использовать для двух целей: первая цель заключается в получении более подробной информации об исключении, учитывая содержимое структуры типа CONTEXT; вторая цель состоит в обработке вложенных исключений.

Теперь приведем, в листинге 18.5, программу, которая выводит на консоль информацию об исключении, используя для получения этой информации функцию GetExceptionInformation.

Листинг 18.5. Получение информации об исключении

```
#include <windows.h>
#include <iostream.h>

EXCEPTION_RECORD er;  // информация об исключении

DWORD filter_function(EXCEPTION_POINTERS *p)
{
  // сохраняем содержимое структуры EXCEPTION_RECORD
  er = *(p->ExceptionRecord);

  return EXCEPTION_EXECUTE_HANDLER;
}

int main()
{
  int  *p = NULL;  // пустой указатель на целое число

  __try
  {
    *p = 10;       // ошибка, так как пустой указатель
```

```
  }
  __except(filter_function(GetExceptionInformation()))
  {
    // распечатываем информацию об исключении
    cout << "ExceptionCode = " << er.ExceptionCode << endl;
    cout << "ExceptionFlags = " << er.ExceptionFlags << endl;
    cout << "ExceptionRecord = " << er.ExceptionRecord << endl;
    cout << "ExceptionAddress = " << er.ExceptionAddress << endl;
    cout << "NumberParameters = " << er.NumberParameters << endl;
    // распечатываем параметры
    if (er.ExceptionCode == EXCEPTION_ACCESS_VIOLATION)
    {
      cout << "Type of access = " << er.ExceptionInformation[0]
           << endl;
      cout << "Address of access = " << er.ExceptionInformation[1]
           << endl;
    }
    cout << endl;
  }

  return 0;
}
```

18.5. Генерация программных исключений

Механизм структурной обработки исключений в Windows позволяет генерировать программные исключения при помощи функции `RaiseException`, которая имеет следующий прототип:

```
VOID  RaiseException(
  DWORD  dwExceptionCode,        // код исключения
  DWORD  dwExceptionFlags,       // флаг возобновляемого исключения
  DWORD  nNumberOfArguments,     // количество аргументов
  CONST  ULONG_PTR  *lpArgumens  // массив аргументов
);
```

Эта функция выбрасывает исключение с кодом, заданным параметром `dwExceptionCode`. Отметим, что код исключения следует формировать в соответствии с правилами, принятыми в операционных системах Windows. Формат кода ошибки в Windows определен в файле winerror.h и имеет структуру, которая описана в табл. 18.1.

Таблица 18.1. *Формат кода ошибки*

Биты	Назначение	Описание
0 — 15	Код ошибки	Устанавливает пользователь, могут принимать любые значения
16 — 27	Код подсистемы, в которой произошла ошибка	Устанавливает пользователь, могут принимать любые значения
28	Зарезервирован системой	Всегда сбрасывается системой в ноль
29	Флаг системной или пользовательской ошибки	Может принимать одно из двух значений: 0 — системная ошибка; 1 — пользовательская ошибка
30 — 31	Код серьезности	Могут принимать следующие значения: 0 — успех; 1 — информация; 2 — предупреждение; 3 — ошибка

Остальные параметры функции `RaiseException` имеют следующее назначение.

Параметр `dwExceptionFlag` может принимать одно из двух значений: `0` или `EXCEPTION_NONCONTINUABLE`. Значение 0 означает, что после исключения возможно восстановление программы, вызвавшей это исключение. Если же параметр `dwExceptionFlag` имеет значение `EXCEPTION_NONCONTINUABLE`, то выполнение программы не может быть возобновлено после исключения. В случае попытки возобновления программы система сгенерирует исключение с кодом `EXCEPTION_NONCONTINUABLE_EXCEPTION`.

Параметр `nNumberOfArguments` указывает количество параметров, передаваемых функции фильтра из блока обработки исключения. Это значение не может превышать величину `EXCEPTION_MAXIMUM_PARAMETERS`.

Параметр `lpArguments` задает параметры, которые передаются в функцию фильтра из блока обработки исключения. Это значение может быть установлено в `NULL`. В этом случае параметр `nNumberOfArguments` игнорируется.

В листинге 18.6 приведена программа, которая генерирует программное исключение и передает функции фильтра два параметра.

Листинг 18.6. Генерация программного исключения

```
#include <windows.h>
#include <iostream.h>

EXCEPTION_RECORD  er;
```

```
DWORD filter_function(EXCEPTION_POINTERS *p)
{
  // сохраняем содержимое структуры EXCEPTION_RECORD
  er = *(p->ExceptionRecord);
  // передаем управление обработчику исключений
  return EXCEPTION_EXECUTE_HANDLER;
}

int main()
{
  int a = 10;
  int b = 20;
  DWORD Arguments[2];

  __try
  {
    if (a < b)
    {
      // устанавливаем аргументы исключения
      Arguments[0] = a;
      Arguments[1] = b;
        // генерируем исключение
      RaiseException(0xFF, 0, 2, Arguments);
    }
    else
    {
      cout << "There is no any exception." << endl;
      cout << "a - b = " << (a - b) << endl;
    }
  }
  __except(filter_function(GetExceptionInformation()))
  {
    cout << "There is an exception." << endl;

    if (er.ExceptionCode == 0xFF)
    {
      cout << "Exception code = " << hex << er.ExceptionCode << endl;
      cout << "Number parameters = " << dec << er.NumberParameters
           << endl;
```

```
      cout << "Parameter[0] = " << er.ExceptionInformation[0] << endl;
      cout << "Parameter[1] = " << er.ExceptionInformation[1] << endl;
    }
  }

  return 0;
}
```

18.6. Необработанные исключения

Если в программе произошло исключение, для которого не существует обработчика исключений, то в этом случае вызывается функция-фильтр системного обработчика исключений, которая выводит на экран окно сообщений с предложением пользователю закончить программу аварийно или выполнить отладку приложения. Системная функция-фильтр `UnhandledExceptionFilter` имеет следующий прототип:

```
LONG  UnhandledExceptionFilter(PEXCEPTION_POINTERS  pExceptionInfo);
```

Эта функция имеет один параметр, который указывает на структуру типа `EXCEPTION_INFO` и возвращает одно из следующих значений:

- `EXCEPTION_CONTINUE_SEARCH` — передать управление отладчику приложения;
- `EXCEPTION_EXECUTE_HANDLER` — передать управление обработчику исключений.

Приложение может заменить системную функцию-фильтр с помощью функции `SetUnhandledExceptionFilter`, которая имеет следующий прототип:

```
LPTOP_LEVEL_EXCEPTION_FILTER  SetUnhandledExceptionFilter(
  LPTOP_LEVEL_EXCEPTION_FILTER  lpTopLevelExceptionFilter
);
```

Эта функция возвращает адрес старой функции фильтра или `NULL`, если установлен системный обработчик исключений. Единственным параметром этой функции является указатель на новую функцию-фильтр, которая будет установлена вместо системной. Эта функция-фильтр должна иметь прототип, соответствующий системной функции фильтра `UnhandledExceptionFilter`, и возвращать одно из следующих значений:

- `EXCEPTION_EXECUTE_HANDLER` — выполнение программы прекращается;
- `EXCEPTION_CONTINUE_EXECUTION` — возобновить исполнение программы с точки исключения;
- `EXCEPTION_CONTINUE_SEARCH` — выполняется системная функция `UnhandledExceptionFilter`.

Для того чтобы восстановить системную функцию-фильтр `UnhandledExceptionFilter`, нужно вызвать функцию `UnhandledExceptionFilter` с параметром `NULL`.

В листинге 18.7 приведена программа, которая устанавливает новую функцию-фильтр для необработанных исключений, а затем восстанавливает системную функцию `UnhandledExceptionFilter`.

Листинг 18.7. Установка новой функции-фильтра для необработанных исключений

```
#include <windows.h>
#include <iostream.h>

LONG new_filter(PEXCEPTION_POINTERS  pExceptionInfo)
{
  cout << "New filter-function is called." << endl;

  cout << "Exception code = " << hex
       << pExceptionInfo->ExceptionRecord->ExceptionCode << endl;

  return EXCEPTION_EXECUTE_HANDLER;
}

int main()
{
  int *p = NULL;
  LPTOP_LEVEL_EXCEPTION_FILTER  old_filter;

  // устанавливаем новую функцию-фильтр необработанных исключений
  old_filter =
   SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)new_filter);
  // выводим адрес старой функции-фильтра
  cout << "Old filter-function address = " << hex << old_filter << endl;
  // создаем необработанное исключение
  *p = 10;

  return 0;
}
```

18.7. Обработка исключений с плавающей точкой

По умолчанию система отключает все исключения с плавающей точкой. Поэтому если при выполнении операции с плавающей точкой было получено число, которое не входит в диапазон представления чисел с плавающей точкой, то в результате система вернет NAN или INFINITY в случае слишком малого или слишком большого числа соответственно. Чтобы включить режим генерации исключений с плавающей точкой нужно изменить состояние слова, управляющего обработкой операций с плавающей точкой. Это можно сделать при помощи функции _controlfp, которая имеет следующий прототип:

```
unsigned int  _controlfp(
  unsigned int  new,
  unsigned int  mask
);
```

Прототип определен в заголовочном файле float.h. Эта функция возвращает старое слово, управляющее обработкой исключений. Здесь параметр new задает новое управляющее слово, а параметр mask определяет, какая битовая маска устанавливается в этом управляющем слове. Для разрешения генерации исключений с плавающей точкой параметр mask должен принимать значение _MCW_EM. Если значение этого параметра равно 0, то функция _controlfp просто возвращает старое управляющее слово.

В параметре new для управления исключениями можно сбрасывать или устанавливать следующие значения:

- _EM_INVALID — исключение EXCEPTION_FLT_INVALID_OPERATION;
- _EM_DENORMAL — исключение EXCEPTION_FLT_DENORMAL_OPERAND;
- _EM_ZERODIVIDE — исключение EXCEPTION_FLT_DIVIDE_BY_ZERO;
- _EM_OVERFLOW — исключение EXCEPTION_FLT_OVERFLOW;
- _EM_UNDERFLOW — исключение EXCEPTION_FLT_UNDERFLOW;
- _EM_INEXACT — исключение EXCEPTION_FLT_INEXACT_RESULT.

Если бит, соответствующий одному из этих значений, сброшен, то система генерирует соответствующее исключение с плавающей точкой, в противном случае исключение не генерируется.

В листинге 18.8 приведена программа, которая обрабатывает исключение с плавающей точкой при делении на ноль.

Листинг 18.8. Обработка исключения с плавающей точкой

```
#include <windows.h>
#include <iostream.h>
#include <float.h>

int main()
{
  double  a = 0;

  // получить управляющее слово, заданное по умолчанию
  int cw = _controlfp( 0, 0 );
  // разрешить обработку исключений с плавающей точкой
  cw &=~(EM_OVERFLOW | EM_UNDERFLOW
        | EM_INEXACT | EM_ZERODIVIDE | EM_DENORMAL);
  // установить новое управляющее слово
  _controlfp( cw, _MCW_EM );
  // теперь можно обрабатывать исключения
  __try
  {
    double  b;

    b = 1/a;  // ошибка, деление на 0
  }
  __except(EXCEPTION_EXECUTE_HANDLER)
  {
    DWORD  ec = GetExceptionCode();  // получаем код исключения

    if (ec == EXCEPTION_FLT_DIVIDE_BY_ZERO)
      cout << "Exception float divide by zero." << endl;
    else
      cout << "Some other exception." << endl;

  }

  return 0;
}
```

18.8. Обработка вложенных исключений

При использовании структурной обработки исключений возможно вкладывать блоки __try и __except в другой блок __try. В этом случае если функция-фильтр внутреннего блока __except возвращает значение EXCEPTION_CONTINUE_SEARCH, то система удаляет все локальные объекты, принадлежащие текущим блокам __try и __except, и продолжает поиск обработчика исключений во внешних блоках __try и __except. Так как локальные объекты, определенные внутри любого блока, хранятся в стеке процесса, то фактически система очищает стек процесса. Область стека, которую занимают локальные объекты одного блока, называется *фреймом стека*. Поэтому можно сказать, что при обработке вложенных исключений выполняется очистка стека процесса от локальных объектов, определенных внутри вложенных блоков __try и __except. Такая очистка стека от локальных объектов называется *глобальной раскруткой стека* или просто *раскруткой стека*.

В листинге 18.9 приведена программа, в которой используется обработка вложенных исключений.

Листинг 18.9. Обработка вложенных исключений

```
#include <windows.h>
#include <iostream.h>

void f(int *p)
{
  int  a = 10;

  *p = a;    // ошибка, так как пустой указатель
}

void g(int *p)
{
  int a = 0;

  *p /= a;     // ошибка, деление на ноль
}

int main()
{
  int a = 10;
```

```
  int  *p = NULL;    // пустой указатель на целое число

  __try
  {
    __try
    {
      f(p);
    }
    __except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
      EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
      cout << "Exception access violation." << endl;
    }

    p = &a;    // указатель не пустой

    __try
    {
      g(p);
    }
    __except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
      EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
      cout << "Exception access violation." << endl;
    }
  }
  __except(GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO ?
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
  {
    cout << "Exception integer divide by zero." << endl;
  }

  return 0;
}
```

В работе программы из листинга 18.9 отметим, что, в первом случае вызывается вложенный обработчик исключения, а во втором — вызывается внешний обработчик исключения.

18.9. Передача управления и выход из фрейма

Для передачи управления из фрейма можно использовать инструкцию `goto` языка программирования C++. В этом случае система считает, что блок с охраняемым кодом завершился аварийно и поэтому выполняет глобальную раскрутку стека. Следовательно, использование инструкции `goto` вызывает исполнение дополнительного программного кода, что замедляет выполнение программы. В листинге 18.10 приведена программа, которая использует инструкцию `goto` для выхода из блока `__try`.

Листинг 18.10. Выход из блока `__try` с использованием инструкции `goto`

```
#include <windows.h>
#include <iostream.h>

int main()
{
  int  *p = NULL;    // пустой указатель на целое число

  __try
  {
    if (!p)
    {
      cout << "Exit with goto from the try block." << endl;

      goto exit_try;   // выходим их блока
    }
    else
      *p = 10;
  }
  __except(EXCEPTION_EXECUTE_HANDLER)
  {
    cout << "There was some exception." << endl;
  }

  cout << "*p = " << *p << endl;

exit_try: return 0;

}
```

Отметим, что в программе из листинга 18.10 выход из фрейма при помощи инструкции `goto` используется для обхода инструкции, следующей после блока `__except`.

Если необходимо просто завершить выполнение блока `__try` без аварийного выхода, т. е. не начиная раскрутку стека, то в этом случае нужно использовать инструкцию `__leave`, которая введена в язык программирования C++ фирмой Microsoft. В листинге 18.11 приведена программа, в которой используется инструкция `__leave` для выхода из блока `__try`.

Листинг 18.11. Выход из блока `__try` с использованием инструкции `__leave`

```
#include <windows.h>
#include <iostream.h>

int main()
{
  int  *p = NULL;    // пустой указатель на целое число

  __try
  {
    if (!p)
    {
      cout << "Exit with __leave from the try block." << endl;

      __leave;     // выходим их блока
    }
    else
    {
      *p = 10;

      cout << "*p = " << *p << endl;
    }
  }
  __except(EXCEPTION_EXECUTE_HANDLER)
  {
    cout << "There was some exception." << endl;
  }

  return 0;

}
```

В заключение этого раздела можно сказать, что если в программе используется инструкция `goto` для выхода из блока `__try`, то, как правило, эта программа плохо структурирована.

18.10. Встраивание SEH в механизм исключений C++

В реализации языка программирования C++ фирмой Microsoft, т. е. в Visual C++, предусмотрен механизм, который позволяет использовать механизм структурной обработки исключений в механизме обработки исключений, используемом в C++. Для этой цели была разработана функция `_set_se_translator`. Эта функция устанавливает в системе функцию, которая называется *функцией-транслятором*, назначение которой состоит в том, чтобы преобразовывать структурные исключения в исключения языка программирования C++. Если функция-транслятор установлена, то она вызывается всегда при выбросе структурного исключения. В функции-трансляторе можно использовать инструкцию `throw` языка программирования C++, которая будет выбрасывать исключение C++ нужного типа.

Функция-транслятор должна иметь следующий прототип:

```
typedef void (*_se_translator_function) (unsigned int,
                                struct _EXCEPTION_POINTERS*);
```

который описан в заголовочном файле eh.h. Как видно из этого описания — функция-транслятор не возвращает значения и получает два параметра: код исключения и указатель на структуру типа `_EXCEPTION_POINTERS`.

Функция `_set_se_translator`, которая используется для установки функции-транслятора, также описана в заголовочном файле eh.h и имеет следующий прототип:

```
_se_translator_function
    _set_se_translator(_se_translator_function  se_trans_func);
```

Единственным параметром этой функции является указатель на новую функцию-транслятор, а возвращает функция `_set_se_translator` адрес старой функции-транслятора, которая в дальнейшем может быть восстановлена при помощи вызова `_set_se_translator`. Если функция-транслятор устанавливается в первый раз, то возвращаемое значение может быть равно `NULL`.

В листинге 18.12 приведена программа, в которой устанавливается функция-транслятор. Эта функция просто выбрасывает исключение языка программирования C++, которое является кодом структурного исключения.

Листинг 18.12. Выброс кода структурного исключения как исключения языка программирования C++

```
#include <windows.h>
#include <iostream.h>
#include <eh.h>

void se_trans_func(unsigned code, EXCEPTION_POINTERS *)
{
  throw code;
}

int main()
{
  int a = 10, b = 0;

  // устанавливаем функцию-транслятор
  _set_se_translator(se_trans_func);
  // перехватываем структурное исключение средствами C++
  try
  {
    a /= b;    // ошибка, деление на ноль
  }
  catch(unsigned code)
  {
    cout << "Exception code = " << hex << code << endl;
  }

  return 0;
}
```

Теперь приведем в листинге 18.13 программу, которая выбрасывает структуру типа `EXCEPTION_POINTERS` как исключение языка программирования C++.

Листинг 18.13. Выброс структуры типа `EXCEPTION_POINTERS` как исключения языка программирования C++

```
#include <windows.h>
#include <iostream.h>
#include <eh.h>
```

```
void se_trans_func(unsigned code, EXCEPTION_POINTERS *info)
{
  EXCEPTION_RECORD er;
  CONTEXT c;

  EXCEPTION_POINTERS ep = {&er, &c};

  er = *(info->ExceptionRecord);
  c = *(info->ContextRecord);

  throw ep;
}

int main()
{
  int a = 10, b = 0;

  // устанавливаем функцию-транслятор
  _set_se_translator(se_trans_func);
  // перехватываем структурное исключение средствами C++
  try
  {
    a /= b;    // ошибка, деление на ноль
  }
  catch(EXCEPTION_POINTERS ep)
  {
    cout << "Exception code = " << hex
         << ep.ExceptionRecord->ExceptionCode << endl;
  }

  return 0;
}
```

Глава 19

Финальная обработка исключений

19.1. Финальные блоки фрейма

В операционных системах Windows существует еще один способ обработки исключений, суть которого заключается в следующем. Код, при исполнении которого возможен выброс исключения, как и в случае с фреймовой обработкой исключений, заключается в блок `__try`. Но только теперь за блоком `__try` следует код, который заключается в блок `__finally`. Система гарантирует, что при любой передаче управления из блока `__try`, независимо от того, произошло или нет исключение внутри этого блока, предварительно управление будет передано блоку `__finally`. Такой способ обработки исключений называется *финальная обработка исключений*. Структурно финальная обработка исключений выглядит следующим образом:

```
__try
{
  // охраняемый код
}
__finally
{
  // финальный код
}
```

Очевидно, что, как и в случае с фреймовой обработкой исключений, финальная обработка исключений требует поддержки как компилятора, так и операционной системы. Кроме того, в язык программирования C++ добавляется новое ключевое слово `__finally`.

Финальная обработка исключений используется для того, чтобы при любом исходе исполнения блока `__try` освободить ресурсы, которые были захвачены внутри этого блока. Такими ресурсами могут быть память, файлы, критические секции и т. д.

В листинге 19.1 приведена программа, в которой используется финальная обработка исключений.

Листинг 19.1. Финальная обработка исключений

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char c;
  int  *p = NULL;  // пустой указатель на целое число

  // введите 'y', чтобы распределить память под целое число
  cout << "Input 'y' to allocate memory: ";
  // вводим символ
  cin >> c;
  // если ввели 'y', то память распределяется, иначе - не распределяется
  if (c == 'y')
    p = new int;

  __try
  {
    *p = 10;  // может быть исключение, если память не распределена
  }
  __finally
  {
    // в любом случае пробуем освободить память
    delete p;
    // выводим сообщение
    cout << "The finally block finished." << endl;
  }

  return 0;
}
```

Отметим, что блок `__finally` выполняется при любом исходе выполнения блока `__try`. Недостатком такой работы блока `__finally` является то, что инструкция `delete` будет выполняться в любом случае, независимо от того, произошло исключение в блоке `__try` или нет. Чтобы избежать такой си-

туации, нужно проверить, как завершился блок __try — нормально или нет. О том, как это делается, будет рассказано в следующем разделе.

19.2. Проверка завершения фрейма

Управление из блока __try может быть передано одним из следующих способов:

- нормальное завершение блока;
- выход из блока при помощи управляющей инструкции __leave;
- выход из блока при помощи одной из управляющих инструкций return, break, continue или goto языка программирования C++;
- передача управления обработчику исключения.

В первых двух случаях считается, что блок __try завершился нормально, а в последних двух случаях — ненормально.

Для того чтобы определить, как завершился блок __try, используется функция AbnormalTermination, которая имеет следующий прототип:

```
BOOL  AbnormalTermination(VOID);
```

В случае если блок __try завершился ненормально, эта функция возвращает ненулевое значение, а в противном случае — значение FALSE.

Используя функцию AbnormalTermination, ресурсы, захваченные в блоке __try, можно освобождать только в том случае, если блок __try завершился ненормально.

В листинге 19.2 приведена программа, которая использует функцию AbnormalTermination для проверки нормального завершения блока __try.

Листинг 19.2. Проверка нормального завершения блока __try

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char c;
  int  *p = NULL;  // пустой указатель на целое число

  // введите 'y', чтобы распределить память под целое число
  cout << "Input 'y' to allocate memory: ";
  // вводим символ
```

```
  cin >> c;
  // если ввели 'y', то память распределяется, иначе - не распределяется
  if (c == 'y')
    p = new int;

  __try
  {
    *p = 10;  // может быть исключение, если память не распределена
  }
  __finally
  {
    // если блок __try закончился нормально
    if (!AbnormalTermination())
    {
      // то освобождаем память
      delete p;
      cout << "The memory is free." << endl;
    }
    else
      // иначе нечего освобождать
      cout << "The memory was not allocated." << endl;
  }

  return 0;
}
```

19.3. Обработка вложенных финальных блоков

Как и в случае фреймовой обработки исключений, можно вкладывать блоки `__try` и `__finally` в другой блок `__try`. В этом случае, если внутри самого внутреннего блока `__try` произошло исключение, то, как и в случае фреймовой обработки исключений, выполняется раскрутка стека. Если самые внутренние блоки `__try` и `__finally` вложены в другие блоки `__try` с финальной обработкой исключений, то при раскрутке стека управление передается всем вложенным блокам `__finally` в порядке, обратном их вложенности.

В листинге 19.3 приведена программа, в которой используется обработка вложенных исключений.

Листинг 19.3. Обработка вложенных финальных блоков

```
#include <windows.h>
#include <iostream.h>

int main()
{
  __try
  {
    int *a, *b;

    __try
    {
      a = new int(10);

      __try
      {
        b = new int(0);
        // ошибка, деление на ноль
        cout << "a/b = " << (*a)/(*b) << endl;
      }
      __finally
      {
        // освобождаем память для 'b'
        delete b;
        cout << "The memory for 'b' is free." << endl;
      }
    }
    __finally
    {
      // освобождаем память для 'a'
      delete a;
      cout << "The memory for 'a' is free." << endl;
    }
  }
  __except(EXCEPTION_EXECUTE_HANDLER)
  {
    cout << "There was some exception." << endl;
  }

  return 0;
}
```

Часть VI

Работа с виртуальной памятью

Глава 20

Виртуальная память

20.1. Концепция виртуальной памяти

Интегральные схемы, предназначенные для хранения программ и данных, называются *физической памятью*. Обычно под физической памятью мы понимаем память, к которой процессор может обращаться, используя адресную шину и шину данных, а внутренняя память самого процессора представляется регистрами. Каждый байт физической памяти имеет свой номер или индекс, который называется *физическим адресом*. При обращении к физической памяти процессор должен выставить на адресную шину физический адрес памяти, к которой он хочет получить доступ.

Под *логической памятью процесса* понимается массив байтов, к которым может обратиться процесс. Индекс каждого элемента этого массива называется *логическим адресом*. Так как логическая память процесса представляется линейным массивом байтов, то логический адрес процесса обычно называют *линейным адресом*.

Так как в действительности процесс может работать только с данными в физической памяти, то во время работы процесса необходимо отображать логическую память процесса в физическую память компьютера. Обычно, прямое отображение невозможно по той простой причине, что объем логической памяти процесса превышает объем физической памяти компьютера. Для решения этой задачи физическую память компьютера дополняют памятью на дисках. Полученную расширенную память называют *виртуальной памятью*, а адрес элемента этой памяти называют *виртуальным адресом*. Тогда при работе процесса выполняется преобразование адресов, которое схематически показано на рис. 20.1.

Преобразование линейного адреса процесса в виртуальный адрес выполняется операционной системой посредством настройки регистров процессора. Обычно линейный адрес процесса отличается от виртуального адреса только интерпретацией бит в этом адресе. Преобразование виртуального адреса в физический адрес выполняется аппаратным образом, а именно — процессором.

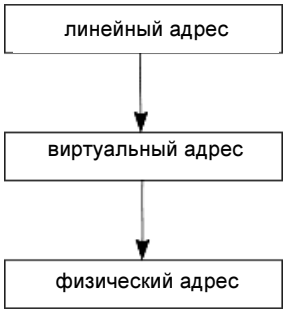


Рис. 20.1. Преобразование адресов при работе процесса

Так как виртуальная память также представляет собой физическую память, то часто для того, чтобы отличать виртуальную память от физической на интегральных схемах, последнюю также называют *реальной памятью.*

20.2. Организация виртуальной памяти

Для реализации преобразования виртуального адреса в физический поступают следующим образом. Виртуальную память разбивают на блоки одинаковой длины, обычно равной 4 Кбайт, которые называют *страницами.* В этом случае файлы, в которых хранятся страницы виртуальной памяти, называются *файлами страниц* или *файлами подкачки.* При обращении процесса по адресу в виртуальной странице, если необходимо, то происходит загрузка этой страницы в реальную память компьютера и настройка адресного пространства процесса на работу с этой страницей. Такая организация виртуальной памяти называется *страничной.*

Учитывая страничную организацию виртуальной памяти — физический и виртуальный адреса имеют форматы, которые показаны на рис. 20.2.

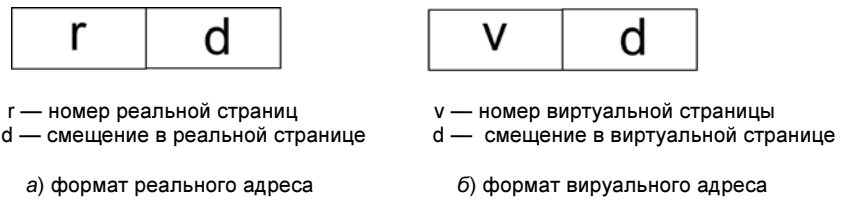
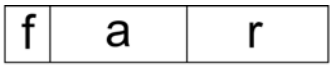


Рис. 20.2. Форматы реального и виртуального адресов

Фактически в этом случае адрес делится на две части: старшую и младшую. Старшая часть адреса рассматривается как номер страницы в реальной или виртуальной памяти, а его младшая часть — как смещение внутри этой страницы. Например, если адрес имеет длину 32 бита, а длина страницы равна 4 Кбайт, то младшие 12 бит рассматриваются как смещение внутри

страницы, старшие 20 бит — как номер страницы. Это разбиение обусловлено тем, что $2^{12} = 4096$ байт. Единственное различие между виртуальным и реальным адресами состоит в том, что виртуальный адрес имеет большую длину поля, отведенного на номер виртуальной страницы.

Для преобразования виртуальных адресов в реальные адреса в системной области физической памяти для каждого процесса хранится таблица страниц, строки которой имеют структуру, которая показана на рис. 20.3.



f — флаг, отмечающий нахождение виртуальной страницы в реальной памяти
a — адрес вирутуальной страницы во внешней памяти
r — адрес реальной страницы в физической памяти

Рис. 20.3. Формат строки таблицы страниц процесса

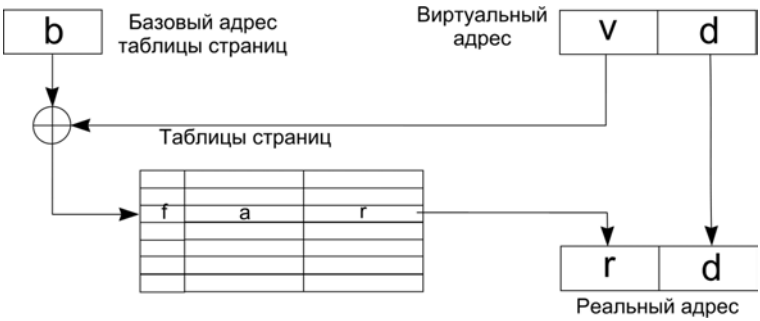


Рис. 20.4. Схема преобразования виртуального адреса в реальный адрес

Схема преобразования виртуального адреса в физический адрес изображена на рис. 20.4, где регистр b содержит базовый адрес таблицы страниц процесса.

Используя эту схему преобразования адресов, алгоритм отображения виртуальной памяти в реальную память может быть описан следующим образом.

1. Находится строка в таблице страниц, соответствующая номеру виртуальной страницы. Индекс этой строки равен b + v.
2. Если f = 1, то виртуальная страница находится в реальной памяти. Если же f = 0, то виртуальная страница находится на диске. В этом случае выполняется загрузка виртуальной страницы в реальную память. После чего значение столбца f устанавливается в 1, а столбец r указывает на адрес виртуальной страницы в реальной памяти.

3. Вычисляется реальный адрес, который формируется из адреса реальной страницы, заданного значением r, и смещения в виртуальной странице, заданного значением d.

Повторим, что все эти действия выполняются аппаратным образом, а именно, процессором.

В заключение этого раздела отметим, что возможен другой подход к организации виртуальной памяти, при котором виртуальная память разбивается на блоки неравной длины, которые называются *сегментами*. В этом случае в реальную память загружаются не страницы, а блоки виртуальной памяти. Существует также комбинированный подход, который заключается в том, что виртуальная память разбивается на страницы, которые потом объединяются в сегменты. Эти подходы к организации виртуальной памяти нами рассматриваться не будут, т. к. в операционных системах Windows используется страничная организация виртуальной памяти. Сегментная организация виртуальной памяти использовалась в старой операционной системе Windows 3.1.

20.3. Алгоритмы замещения страниц

При подкачке виртуальной страницы в физическую память может оказаться, что все страницы физической памяти уже заняты другими виртуальными страницами. В этом случае одна из физических страниц выталкивается из физической памяти на диск, а на ее место с диска загружается требуемая виртуальная страница. Возникает следующая проблема — какую виртуальную страницу вытолкнуть из физической памяти на диск? Для определения такой страницы чаще всего используются следующие алгоритмы:

- алгоритм FIFO (first in first out), который заключается в том, что первой выталкивается на диск первая из загруженных в реальную память виртуальных страниц;
- алгоритм LRU (least recently used), который заключается в том, что первой из памяти выталкивается на диск виртуальная страница, которая дольше всего не использовалась;
- алгоритм LFU (least frequently used), который заключается в том, что на диск выталкивается виртуальная страница, которая наименее часто используется;
- алгоритм NUR (not used recently), который похож на алгоритм LRU, но при выталкивании на диск выбираются те страницы, на которые дольше всего не было записи.

Все эти алгоритмы имеют свои преимущества и недостатки, и для каждого из них можно подобрать такой случай использования виртуальных страниц, при котором система начнет буксовать. То есть при каждом новом обращении

к памяти будет необходима подкачка новой виртуальной страницы в реальную память. Оптимального алгоритма подкачки виртуальных страниц не существует, т. к. поведение приложений непредсказуемо.

Если загрузкой виртуальных страниц управляет сама программа, все шаги выполнения которой заранее известны, то, очевидно, что возможен оптимальный порядок загрузки и выгрузки виртуальных страниц, который минимизирует обращения к диску.

Так как запись виртуальной страницы на диск довольно медленная операция, то обычно элемент таблицы страниц содержит также флаг, отмечающий, была ли произведена запись на виртуальную страницу, находящуюся в реальной памяти. Если записи не было, то эта виртуальная страница просто затирается при подкачке на ее место другой виртуальной страницы с диска.

20.4. Рабочее множество процесса

Эмпирически было определено, что при работе многих программ наблюдается свойство локальности. То есть выполняемый в какой-то интервал времени код программы и используемая программой память расположены локально, а не разбросаны по всей программе. Обычно, программисты пишут свои программы, также следуя этому правилу. Поэтому для эффективной работы программы необходимо, чтобы какое-то множество часто используемых на данном интервале времени виртуальных страниц находилось в реальной памяти. Это множество виртуальных страниц называется *рабочим множеством страниц* процесса. Естественно, рабочее множество страниц процесса изменяется со временем: какие-то страницы начинают использоваться реже, а какие-то — чаще. Но для каждого процесса можно определить размеры рабочего множества страниц таким образом, что на каждом интервале времени все часто используемые страницы будут находиться в реальной памяти.

20.5. Организация виртуальной памяти в Windows

Как уже говорилось, в операционных системах Windows используется страничная организация виртуальной памяти. При этом линейный адрес процесса совпадает с его виртуальным адресом, формат которого показан на рис. 20.5.

Исходя из этого формата, реальный адрес вычисляется по схеме, которая показана на рис. 20.6.

Единственное отличие этой схемы от изложенной ранее схемы преобразования виртуального адреса в реальный заключается в том, что теперь адрес таблицы страниц хранится не в регистре процессора, а в специальной таблице, которая называется *каталогом таблиц страниц*. Каждая строка катало-

га таблиц страниц содержит адрес уникальной таблицы страниц. В свою очередь поле p виртуального адреса имеет уникальное значение для каждого процесса. Поэтому каждый процесс использует единственную таблицу страниц. Отсюда следует, что адресные пространства разных процессов изолированы друг от друга.

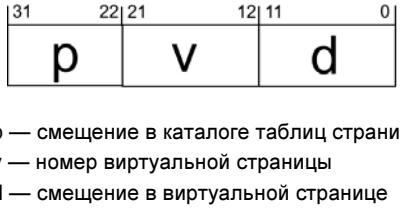


Рис. 20.5. Формат виртуального адреса в Windows

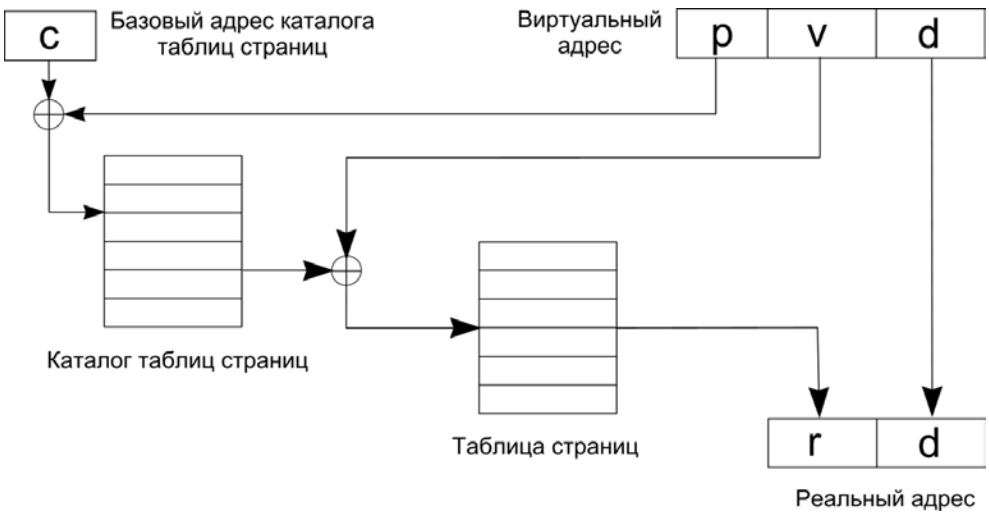
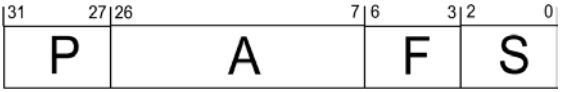


Рис. 20.6. Схема преобразования виртуального адреса в реальный адрес в Windows

Строка таблицы страниц имеет структуру, которая показана на рис. 20.7. Как видно из этого рисунка, строка таблицы страниц имеет довольно сложную структуру, основным назначением которой является сокращение обмена виртуальными страницами между реальной и виртуальной памятью. Опишем назначение полей строки таблицы страниц более подробно. Это прояснит организацию виртуальной памяти в Windows.



S — описывает состояние страницы
F — задает номер файла подкачки
A — содержит физический адрес страницы
P — содержит атрибуты доступа к странице

Рис. 20.7. Структура строки таблицы страниц в Windows

Поле S описывает состояние виртуальной страницы, которое определяется комбинацией трех бит. Различные комбинации этих бит определяют следующие состояния виртуальной страницы:

- страницы нет в реальной памяти (invalid page);
- страница находится в реальной памяти (valid page);
- страница находится в реальной памяти и была модифицирована (valid dirty page);
- страница загружается в реальную память (invalid page in transition);
- страница сохраняется на диск (invalid dirty page in transition).

Поле F задает номер файла подкачки на диске. Учитывая длину этого поля, можно определить до 16 файлов подкачки виртуальных страниц.

Поле A содержит физический адрес страницы в реальной памяти при условии, что страница загружена с диска. Учитывая длину этого поля, можно определить, что виртуальная память процесса может содержать 2^{20} виртуальных страниц. Отсюда следует, что, т. к. длина виртуальной страницы равна 4 Кбайт, то вся виртуальная память процесса составляет $2^{20} \times 4$ Кбайт = 4 Гбайта виртуальной памяти.

Поле P содержит атрибуты доступа к виртуальной странице, которые могут принимать следующие значения:

- `PAGE_NOACCESS` — доступ к странице запрещен;
- `PAGE_READONLY` — доступно только чтение страницы;
- `PAGE_READWRITE` — доступны чтение и запись страницы.

В заключение этого раздела отметим, что в операционных системах семейства Windows NT управлением виртуальной памяти занимается специальный процесс, который называется менеджером виртуальной памяти (Virtual Memory Manager — VMM). Менеджер виртуальной памяти поддерживает свое внутреннее описание состояния каждой страницы в реальной памяти. В соответствии с этим описанием каждая страница реальной памяти может находиться в одном из следующих состояний:

- страница в рабочем состоянии и используется процессом (valid);

- страница записывается на диск (modified);
- страница удаляется из рабочего множества страниц процесса (standby);
- страница освобождена процессом, но не заполнена нулями (free);
- страница заполнена нулями и может использоваться любым процессом (zeroed);
- страница в нерабочем состоянии (bad).

Для каждого процесса операционная система Windows определяет рабочее множество страниц этого процесса. При замещении страниц Wndows использует алгоритм LRU, но только с тем отличием, что он применяется не для всех виртуальных страниц, находящихся в реальной памяти, а отдельно для рабочего множества страниц каждого процесса. Во время работы процесса менеджер виртуальной памяти периодически проверяет частоту использования страниц из рабочего множества процесса. Если некоторая виртуальная страница используется редко, то она удаляется из рабочего множества процесса.

Глава 21

Работа с виртуальной памятью в Windows

21.1. Состояния виртуальной памяти процесса

Линейный адрес процесса в Windows состоит из 32 бит и изменяется в пределах от `0x00000000` до `0xFFFFFFFF`. Это теоретически позволяет процессу обращаться к 4 Гбайт логической памяти. В операционных системах семейства Windows NT процессу доступны два младших гигабайта этой памяти с диапазоном адресов от `0x00000000` до `0x7FFFFFFF`, а ее старшие два гигабайта с диапазоном адресов от `0x80000000` до `0xFFFFFFFF` используются системой. В операционной системе Windows 98 из 2 Гбайт памяти доступной процессу, операционная система использует еще 64 Кбайт памяти с диапазоном адресов от `0x00000000` до `0x0000FFFF` для проверки присваивания значений через указатели, значения которых инициализированы в `NULL`, и для поддержки совместимости со старыми операционными системами MS-DOS и Windows 3.1.

В операционных Windows виртуальный адрес процесса отличается от линейного адреса этого же процесса только интерпретацией бит линейного адреса. Поэтому, можно сказать, что каждому процессу в Windows также доступно два гигабайта виртуальной памяти. Это не значит, что процесс может использовать всю эту память одновременно. Количество виртуальной памяти, доступной процессу, зависит от емкости физической памяти и дисков. Чтобы ограничить процесс в использовании виртуальной памяти, некоторые страницы в таблице страниц могут быть помечены как недоступные.

После этих замечаний перейдем к описанию состояния виртуальной памяти процесса. С точки зрения процесса, страницы его виртуальной памяти могут находиться в одном из трех состояний:

- свободны для использования (free);
- распределены процессу для использования (committed).
- зарезервированы, но не используются процессом (reserved);

Поясним эти состояния более подробно. Первоначально, при запуске процесса, все страницы виртуальной памяти считаются свободными, естественно кроме тех, в которые загружена сама программа. Адреса загрузки всех модулей можно узнать, выбрав, при отладке программы, в среде разработки Visual C++ в пункте меню **Debug | Modules**. Чтобы распределить для использования свободные или зарезервированные страницы виртуальной памяти, процесс должен вызвать функцию `VirtualAlloc`. Только после успешного завершения этой функции процесс может использовать распределенную ему виртуальную память. Третье состояние характеризует виртуальные страницы как зарезервированные. Это значит, что эти виртуальные страницы зарезервированы процессом для дальнейшего использования и не будут выделяться системой для использования процессу без точного указания процессом их адреса. Следует отметить, что при резервировании виртуальных страниц реальная память под эти страницы не выделяется.

21.2. Резервирование, распределение и освобождение виртуальной памяти

Для резервирования или распределения области виртуальной памяти процесс должен вызвать функцию `VirtualAlloc`, которая имеет следующий прототип:

```
LPVOID  VirtualAlloc(
  LPVOID  lpAddress,    // область для распределения или резервирования
  SIZE_T  dwSize,           // размер области
  DWORD   flAllocationType, // тип распределения
  DWORD   flProtect         // тип защиты доступа
);
```

В случае успешного завершения эта функция возвращает адрес виртуальной памяти, распределенной или зарезервированной процессом, а в случае неудачи — `NULL`. При этом отметим такую деталь, если распределение виртуальной памяти функцией `VirtualAlloc` завершается успешно, то выделенная память автоматически инициализируется нулями. Опишем назначение параметров этой функции.

Параметр `lpAddress` устанавливается вызывающей программой и указывает системе начальный адрес виртуальной памяти, которую процесс хочет зарезервировать или распределить. Этот адрес может указывать как на свободную, так и зарезервированную ранее виртуальную память. При установке этого адреса следует различать следующие ситуации:

- в случае резервирования виртуальной памяти этот адрес выравнивается системой до границы в 64 Кбайт, которая предшествует указанному адресу;

- в случае распределения виртуальной памяти из зарезервированной ранее области этот адрес округляется операционной системой до границы виртуальной страницы, содержащей этот адрес;
- в случае, если параметр `lpAddress` равен `NULL`, то операционная система сама выбирает начальный адрес области виртуальной памяти.

Параметр `dwSize` устанавливается вызывающей программой и указывает размер распределяемой или резервируемой области виртуальной памяти в байтах. Если параметр `lpAddress` равен `NULL`, то система округляет эту величину в большую сторону до кратности размеру виртуальной страницы. Если же память распределяется по конкретным адресам, то эта память будет включать все страницы, которые содержат байты из диапазона от `lpAddress` до `lpAddress+dwSize`.

Параметр `flAllocationType` устанавливается программой и указывает на тип операции, которую выполняет функция `VirtualAlloc`. Значением этого параметра может быть любая комбинация следующих флагов:

- `MEM_COMMIT` — распределить память программе;
- `MEM_RESERVE` — зарезервировать область физической памяти.

В операционных системах Windows NT/2000 в этом параметре могут устанавливаться следующие флаги:

- `MEM_RESET` — память временно не используется;
- `MEM_TOP_DOWN` — распределить память, начиная с наибольшего из свободных адресов.

Кроме того, в операционной системе Windows 98 может быть установлен флаг:

- `MEM_WRITE_WATCH` — запоминать адреса виртуальных страниц, в которые была проведена запись.

Впоследствии адреса этих страниц можно узнать, вызвав функцию `GetWriteWatch`. Очистить список таких страниц можно посредством вызова функции `ResetWriteWatch`. Так как использование этих функций возможно только в операционной системе Windows 98, то мы их подробно рассматривать не будем.

Параметр `flProtect` устанавливает атрибуты доступа к области виртуальной памяти, которые разрешают выполнять над страницами виртуальной памяти только определенные операции. Этот параметр может быть комбинацией следующих флагов:

- `PAGE_READONLY` — разрешает только чтение виртуальных страниц;
- `PAGE_READWRITE` — разрешает чтение и запись в виртуальных страницах;
- `PAGE_EXECUTE` — разрешает только исполнение кода в виртуальных страницах;

- PAGE_EXECUTE_READ — разрешает исполнение и чтение кода в виртуальных страницах;
- PAGE_EXECUTE_READWRITE — разрешает выполнение, чтение и запись виртуальных страниц;
- PAGE_NOACCESS — к виртуальным страницам нет доступа;
- PAGE_NOCACHE — виртуальные страницы можно не помещать в кэш.

Отметим, что попытка чтения или записи в страницу, которая предназначена только для исполнения кода (флаг PAGE_EXECUTE), вызовет ошибку доступа (access violation). Эта же ошибка возникнет в случае любого доступа к виртуальной странице, которая отмечена флагом PAGE_NOACCESS.

В операционных системах Windows NT/2000 в параметре flProtect может быть также установлен флаг:

- PAGE_GUARD — охраняемая страница.

Если этот флаг установлен и к странице произведено обращение, все равно, чтение или запись, система сбрасывает этот флаг и возбуждает исключение типа EXCEPTION_GUARD_PAGE, которое имеет код 0x80000001. Этот флаг используется для определения момента, когда процессу необходимо выделить дополнительную виртуальную память. Например, этот флаг можно установить для последней виртуальной страницы области, которая используется процессом при работе с некоторыми данными. Тогда при попытке доступа к этой странице генерируется исключение, и процесс будет знать, что виртуальная память для данных заканчивается. Отметим также, что флаг PAGE_GUARD не может быть использован совместно с флагом PAGE_NOACCESS.

После завершения работы с виртуальной памятью ее необходимо освободить, используя функцию VirtualFree, которая имеет следующий прототип:

```
BOOL  VirtualFree(
  LPVOID  lpAddress,     // адрес области виртуальной памяти
  SIZE_T  dwSize,        // размер области
  DWORD   dwFreeType     // тип операции
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — FALSE. Описание параметров этой функции начнем с последнего параметра, т. к. его значение влияет на значения, в которые программа должна установить первых два параметра.

Параметр dwFreeType может принимать любую комбинацию следующих двух флагов:

- MEM_DECOMMIT — отменить распределение виртуальной памяти;
- MEM_RELEASE — освободить виртуальную память.

Если используется флаг `MEM_DECOMMIT`, то память не освобождается, а остается в зарезервированном состоянии. Чтобы освободить зарезервированную виртуальную память, нужно установить флаг `MEM_RELEASE`. В общем случае относительно установки значения этого параметра можно сказать, что оно должно соответствовать состоянию области памяти, с которой работает функция `VirtualFree`.

Параметр `lpAddress` должен указывать на базовый адрес области, для которой нужно отменить распределение или освободить. Если в параметре `dwFreeType` установлен флаг `MEM_RELEASE`, то этот адрес должен совпадать с адресом, который возвратила функция `VirtualAlloc`.

Параметр `dwSize` задает в байтах размер области виртуальной памяти, распределение которой нужно отменить. Если в параметре `dwFreeType` установлен флаг `MEM_RELEASE`, то значение параметра `dwSize` должно быть равно нулю.

Теперь можно перейти к примерам, которые иллюстрируют работу функций `VirtualAlloc` и `VirtualFree`. Сначала рассмотрим программу в листинге 21.1, которая распределяет область виртуальной памяти под массив целых чисел, а затем освобождает ее. Отметим один момент в этой программе. Чтобы сразу распределить область виртуальной памяти, мы должны установить параметр `lpAddress` в значение `NULL`. В этом случае система сама определяет начальный адрес области виртуальной памяти для распределения процессу. Как распределяется память по конкретному виртуальному адресу, будет показано в листинге 21.2.

Листинг 21.1. Распределение виртуальной памяти процессу

```
#include <windows.h>
#include <iostream.h>

int main()
{
  int *a;    // указатель на массив целых чисел
  const int size = 1000;   // размерность массива

  // распределяем виртуальную память
  a = (int*)VirtualAlloc(
      NULL,
      size * sizeof(int),
      MEM_COMMIT,
      PAGE_READWRITE);
```

```
  if(!a)
  {
    cout << "Virtual allocation failed." << endl;
    return GetLastError();
  }

  cout << "Virtual memory address: " << a << endl;

  // освобождаем виртуальную память
  if (!VirtualFree(a, 0, MEM_RELEASE))
  {
    cout << "Memory release failed." << endl;
    return GetLastError();
  }

  return 0;
}
```

В программе из листинга 21.2 показано, как распределить область виртуальной памяти по конкретному адресу. Отметим, что в этом случае значение параметра `flAllocationType` должно быть установлено как комбинация флагов `MEM_RESERVE` и `MEM_COMMIT`. То есть при распределении памяти по конкретному адресу эта память должна быть предварительно зарезервирована, что может быть выполнено одним вызовом функции `VirtualAlloc`.

Листинг 21.2. Распределение виртуальной памяти по конкретному адресу

```
#include <windows.h>
#include <iostream.h>

int main()
{
  LPVOID  lp;
  const int size = 1000;

  // распределяем виртуальную память
  lp = VirtualAlloc(
      (LPVOID) 0x00890002,
      size,
      MEM_RESERVE | MEM_COMMIT,
```

```
      PAGE_READWRITE);
  if(!lp)
  {
    cout << "Virtual allocation failed." << endl;
    return GetLastError();
  }

  cout << "Virtual memory address: " << lp << endl;

  // освобождаем виртуальную память
  if (!VirtualFree(lp, 0, MEM_RELEASE))
  {
    cout << "Memory release failed." << endl;
    return GetLastError();
  }
  return 0;
}
```

Обратим внимание на следующий момент в программе из листинга 21.2. Хотя мы и установили начальный адрес виртуальной памяти на `0x00890002`, система все равно выровняла его до ближайшей нижней границы виртуальной страницы, т. е. до `0x00890000`.

Теперь распределим память в два этапа. Сначала зарезервируем область виртуальной памяти, а затем распределим некоторую часть этой виртуальной памяти. Как это делается — показано в программе из листинга 21.3. Заметим, что в этой программе функция `VirtualFree` используется также для отмены распределения области виртуальной памяти.

Листинг 21.3. Резервирование и распределение виртуальной памяти

```
#include <windows.h>
#include <iostream.h>

int main()
{
  LPVOID  lpr, lpc;
  const int Kb = 1024;
  const int size = 100;

  // резервируем виртуальную память
```

```
lpr = VirtualAlloc(
    (LPVOID) 0x00880000,
    size * Kb,
    MEM_RESERVE,
    PAGE_READWRITE);
if(!lpr)
{
  cout << "Virtual memory reservation failed." << endl;
  return GetLastError();
}

cout << "Virtual memory address: " << lpr << endl;

// распределяем виртуальную память
lpc = VirtualAlloc(
    (LPVOID) 0x00888000,
    Kb,
    MEM_COMMIT,
    PAGE_READWRITE);
if(!lpc)
{
  cout << "Virtual memory allocation failed." << endl;
  return GetLastError();
}

cout << "Virtual memory address: " << lpc << endl;

// отменяем распределение
if (!VirtualFree(lpc, Kb, MEM_DECOMMIT))
{
  cout << "Memory decommit failed." << endl;
  return GetLastError();
}

// освобождаем виртуальную память
if (!VirtualFree(lpr, 0, MEM_RELEASE))
{
  cout << "Memory release failed." << endl;
```

```
    return GetLastError();
  }

  return 0;
}
```

В завершение этого раздела рассмотрим программу из листинга 21.4, которая устанавливает страницу в охраняемое состояние. При первой попытке доступа к этой странице система сбрасывает флаг PAGE_GUARD и генерирует исключение EXCEPTION_GUARD_PAGE. Состояние страницы не изменяется. Повторный доступ к странице происходит обычным образом, т. к. никаких исключений не генерируется. Заметим, что начальное состояние распределенной виртуальной памяти инициализируется нулями. Также отметим, что этот код работает только в операционной системе Windows 2000.

Листинг 21.4. Доступ к охраняемой странице виртуальной памяти

```
#include <windows.h>
#include <iostream.h>

int main()
{
  int *a;    // указатель на массив целых чисел
  const int size = 1024;   // размерность массива

  // распределяем виртуальную память
  a = (int*)VirtualAlloc(
      NULL,
      size * sizeof(int),
      MEM_COMMIT,
      PAGE_READWRITE|PAGE_GUARD);
  if(!a)
  {
    cout << "Virtual allocation failed." << endl;
    return GetLastError();
  }

  __try
  {
    a[10] = 10;
```

```
  }
  __except(EXCEPTION_EXECUTE_HANDLER)
  {
    DWORD ecode = GetExceptionCode();

    if (ecode == EXCEPTION_GUARD_PAGE)
      cout << "Access to a guard virtual page." << endl;
    else
      cout << "Some exception." << endl;
  }

  cout << "a[10] = " << a[10] << endl;
  a[10] = 10;
  cout << "a[10] = " << a[10] << endl;

  // освобождаем виртуальную память
  if (!VirtualFree(a, 0, MEM_RELEASE))
  {
    cout << "Memory release failed." << endl;
    return GetLastError();
  }

  return 0;
}
```

21.3. Блокирование виртуальных страниц в реальной памяти

Если некоторая область виртуальной памяти будет часто использоваться процессом, то можно запретить системе выгружать эти виртуальные страницы из реальной памяти, иначе говоря, запереть или блокировать эти виртуальные страницы в реальной памяти. Для этого нужно использовать функцию `VirtualLock`, которая имеет следующий прототип:

```
BOOL  VirtualLock(
  LPVOID  lpAddress,     // адрес области
  SIZE_T  dwSize         // размер области
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. Параметры `lpAddress` и `dwSize` указывают,

соответственно, базовый адрес и размеры области, которая запирается в реальной памяти. Подчеркнем, что в реальной памяти запираются все виртуальные страницы, которые содержат диапазон адресов от lpAddress до lpAddress+dwSize. При этом нельзя запирать в реальной памяти виртуальные страницы, к которым доступ запрещен, т. е. для которых установлен флаг PAGE_NOACCESS. Кроме того, операционная система не позволяет процессу блокировать в реальной памяти более 30 виртуальных страниц сразу.

Для отмены блокировки виртуальных страниц в реальной памяти используется функция VirtualUnlock, которая имеет следующий прототип:

```
BOOL  VirtualUnlock(
  LPVOID  lpAddress,     // адрес области
  SIZE_T  dwSize         // размер области
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — FALSE. Параметры этой функции имеют то же назначение, что и параметры функции VirtualLock. При этом до вызова функции VirtualUnlock диапазон адресов от lpAddress до lpAddress+dwSize должен быть предварительно блокирован функцией VirtualLock. Если это условие не выполняется, то операционная система освобождает страницы из рабочего множества страниц процесса.

Необходимо отметить, что функции VirtualLock и VirtualUnlock работают только в операционных системах Windows NT/2000.

Теперь, в листинге 21.5, приведем программу, которая блокирует, а затем разблокирует страницы в виртуальной памяти.

Листинг 21.5. Блокирование и разблокирование виртуальных страниц

```
#include <windows.h>
#include <iostream.h>

int main()
{
  LPVOID  vm;  // указатель на виртуальную память
  SIZE_T  size = 4096;   // размер памяти

  // распределяем виртуальную память
  vm = VirtualAlloc(
      NULL,
      size,
      MEM_COMMIT,
```

```
      PAGE_READWRITE);
  if(!vm)
  {
    cout << "Virtual allocation failed." << endl;
    return GetLastError();
  }

  // блокируем виртуальную память
  if (!VirtualLock(vm, size))
  {
    cout << "Virtual lock failed." << endl;
    return GetLastError();
  }
  // разблокируем виртуальную память
  if (!VirtualUnlock(vm, size))
  {
    cout << "Virtual unlock failed." << endl;
    return GetLastError();
  }

  // освобождаем виртуальную память
  if (!VirtualFree(vm, 0, MEM_RELEASE))
  {
    cout << "Memory release failed." << endl;
    return GetLastError();
  }

  return 0;
}
```

21.4. Изменение атрибутов доступа к виртуальной странице

Изменить атрибуты доступа к области виртуальной памяти можно при помощи вызова функции `VirtualProtect`, которая имеет следующий прототип:

```
BOOL  VirtualProtect(
  LPVOID  lpAddress,       // адрес области памяти
  SIZE_T  dwSize,          // размер области памяти
```

```
    DWORD   flNewProtect,    // флаги новых атрибутов доступа
    PDWORD  lpflOldProtect   // указатель на старые атрибуты доступа
  );
```

В случае успешного завершения эта функция возвращает значение, а в случае неудачи — `FALSE`. Параметры этой функции имеют тот же смысл, что и при распределении виртуальной памяти функцией `VirtualAlloc`. Единственное отличие состоит в том, что старые атрибуты доступа возвращаются функцией `VirtualProtect` по адресу `lpflOldProtect`, который должна установить вызывающая программа.

В листинге 21.6 приведена программа, которая изменяет атрибуты доступа к виртуальной странице, разрешая не только чтение этой виртуальной страницы, но и запись.

Листинг 21.6. Чтение и изменение рабочего множества виртуальных страниц процесса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  DWORD  dwOldProtect;     // для старых атрибутов защиты
  int *a;                  // указатель на массив целых чисел
  const int size = 1000;   // размерность массива

  // распределяем виртуальную память
  a = (int*)VirtualAlloc(
      NULL,
      size * sizeof(int),
      MEM_COMMIT,
      PAGE_READONLY);
  if(!a)
  {
    cout << "Virtual allocation failed." << endl;
    return GetLastError();
  }

  // попробуем записать в виртуальную память
  __try
  {
    a[10] = 10;
```

```
  }
  __except(EXCEPTION_EXECUTE_HANDLER)
  {
    DWORD ecode = GetExceptionCode();

    if (ecode == EXCEPTION_ACCESS_VIOLATION)
      cout << "Access to write protected page." << endl;
    else
      cout << "Some exception." << endl;
  }

  // изменим атрибуты доступа
  if (!VirtualProtect(a, size, PAGE_READWRITE, &dwOldProtect))
  {
    cout << "Virtual protect failed." << endl;
    return GetLastError();
  }

  // теперь можно писать в виртуальную память
  a[10] = 10;
  cout << "a[10] = " << a[10] << endl;

  // освобождаем виртуальную память
  if (!VirtualFree(a, 0, MEM_RELEASE))
  {
    cout << "Memory release failed." << endl;
    return GetLastError();
  }

  return 0;
}
```

21.5. Управление рабочим множеством страниц процесса

Узнать о количестве страниц, которые входят в рабочее множество процесса, можно посредством вызова функции `GetProcessWorkingSetSize`, которая имеет следующий прототип:

```
BOOL  GetProcessWorkingSetSize(
  HANDLE   hProcess,              // дескриптор процесса
```

```
    PSIZE_T  lpMinWorkingSetSize,  // мин. размер рабочего множества
    PSIZE_T  lpMaxWorkingSetSize   // макс. размер рабочего множества
  );
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. При вызове этой функции в параметре `hProcess` должен быть установлен дескриптор процесса, для которого мы хотим узнать диапазон рабочего множества страниц. При успешном завершении функция `GetProcessWorkingSetSize` возвращает по адресам, заданным в параметрах `lpMinWorkingSetSize` и `lpMaxWorkingSetSize`, соответственно минимальный и максимальный размеры рабочего множества процесса в байтах. При работе процесса менеджер виртуальной памяти поддерживает количество виртуальных страниц, распределенных процессу, в этом диапазоне.

Минимальный и максимальный размеры рабочего множества страниц процесса можно изменить посредством вызова функции `SetProcessWorkingSetSize`, которая имеет следующий прототип:

```
  BOOL  SetProcessWorkingSetSize(
    HANDLE   hProcess,             // дескриптор процесса
    SIZE_T   dwMinWorkingSetSize,  // мин. размер рабочего множества
    SIZE_T   dwMaxWorkingSetSize   // макс. размер рабочего множества
  );
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. Как и в функции `GetProcessWorkingSetSize`, параметр `hProcess` содержит дескриптор процесса, для которого изменяется диапазон рабочего множества страниц. Параметры `dwMinWorkingSetSize` и `dwMaxWorkingSetSize` устанавливаются в вызывающей программе. Они указывают на новые минимальный и максимальный размеры рабочего множества процесса в байтах. Если значения обоих этих параметров установлены в `-1`, то из рабочего множества страниц процесса удаляются все страницы.

Отметим, что функция `SetProcessWorkingSetSize` изменит диапазон рабочего множества страниц процесса только в том случае, если в дескрипторе процесса `hProcess` установлен режим доступа `PROCESS_SET_QUOTA`. Этот режим будет установлен автоматически, если процесс создается администратором.

В заключение отметим, что функции `GetProcessWorkingSetSize` и `SetProcessWorkingSetSize` работают только в операционных системах Windows NT/2000.

Теперь приведем в листинге 21.7 программу, которая определяет размеры рабочего множества страниц процесса, а затем устанавливает новые размеры.

Листинг 21.7. Чтение и изменение рабочего множества виртуальных страниц процесса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  const int size = 4096;   // размер страницы
  HANDLE  hProcess;        // дескриптор процесса

  SIZE_T  min, max;  // мин. и макс. размеры рабочего множества страниц
  SIZE_T  *pMin = &min;    // указатель на минимальный размер
  SIZE_T  *pMax = &max;    // указатель на максимальный размер

  // получить дескриптор текущего процесса
  hProcess = GetCurrentProcess();
  // прочитать границы рабочего множества
  if(!GetProcessWorkingSetSize(hProcess, pMin, pMax))
  {
    cout << "Get process working set size failed." << endl;
    return GetLastError();
  }
  else
  {
    cout << "Min = " << (min/size) << endl;
    cout << "Max = " << (max/size) << endl;
  }

  // установить новые границы рабочего множества
  if(!SetProcessWorkingSetSize(hProcess, min-10, max-10))
  {
    cout << "Set process working set size failed." << endl;
    return GetLastError();
  }

  // прочитать новые границы рабочего множества
  if(!GetProcessWorkingSetSize(hProcess, pMin, pMax))
  {
```

```
    cout << "Get process working set size failed." << endl;
    return GetLastError();
  }
  else
  {
    cout << "Min = " << (min/size) << endl;
    cout << "Max = " << (max/size) << endl;
  }
  return 0;
}
```

21.6. Инициализация и копирование блоков виртуальной памяти

Чтобы заполнить блок памяти определенным значением, используется функция `FillMemory`, которая имеет следующий прототип:

```
VOID  FillMemory(
  PVOID   Destination,   // адрес блока памяти
  SIZE_T  Length,        // длина блока
  BYTE    Fill           // символ-заполнитель
);
```

Эта функция заполняет блок памяти, длина в байтах и базовый адрес которого задаются соответственно параметрами `Length` и `Destination`, символом, заданным в параметре `Fill`.

Если блок памяти необходимо заполнить нулями, то для этого можно использовать функцию `ZeroMemory`, которая имеет следующий прототип:

```
VOID  ZeroMemory(
  PVOID   Destination,   // адрес блока памяти
  SIZE_T  Length,        // длина блока
);
```

Параметры этой функции имеют то же назначение, что и параметры функции `FillMemory` , исключая символ-заполнитель.

Для копирования блока виртуальной памяти используется функция `CopyMemory`, которая имеет следующий прототип:

```
VOID  CopyMemory(
  PVOID   Destination,     // адрес области назначения
  CONST  VOID  *Source,    // адрес исходной области
  SIZE_T  Length           // длина блока памяти
);
```

Эта функция копирует блок памяти, длина в байтах и базовый адрес которого задаются соответственно параметрами `Length` и `Source` в область памяти по адресу `Destination`. Отметим, что результат выполнения функции `CopyMemory` непредсказуем, если исходный и результирующий блоки памяти перекрываются.

Для копирования перекрывающихся блоков памяти используется функция `MoveMemory`, которая имеет следующий прототип:

```
VOID  MoveMemory(
  PVOID   Destination,     // адрес области назначения
  CONST  VOID  *Source,    // адрес исходной области
  SIZE_T  Length           // длина блока памяти
);
```

Параметры этой функции полностью совпадают с параметрами функции `CopyMemory`.

Теперь в листинге 21.8 приведем программу, которая использует перечисленные выше функции.

Листинг 21.8. Инициализация и копирование блоков виртуальной памяти

```
#include <windows.h>
#include <iostream.h>

int main()
{
  BYTE *a, *b;      // указатели на блоки памяти
  const int size = 1000;   // размерность массива

  // распределяем виртуальную память
  a = (BYTE*)VirtualAlloc(NULL, size, MEM_COMMIT, PAGE_READWRITE);
  if(!a)
  {
    cout << "Virtual allocation failed." << endl;
    return GetLastError();
  }
  b = (BYTE*)VirtualAlloc(NULL, size, MEM_COMMIT, PAGE_READWRITE);
  if(!b)
  {
    cout << "Virtual allocation failed." << endl;
    return GetLastError();
```

```
  }

  // инициализируем символом X
  FillMemory(a, size, 'X');
  // копируем блок A в блок B
  CopyMemory(b, a, size);
  // распечатываем результат
  cout << "b[10] = " << b[10] << endl;

  // освобождаем виртуальную память
  if (!VirtualFree(a, 0, MEM_RELEASE))
  {
    cout << "Memory release failed." << endl;
    return GetLastError();
  }

  return 0;
}
```

21.7. Определение состояния памяти

Определить состояние области виртуальной памяти процесса можно при помощи вызова функции `VirtualQuery`, которая имеет следующий прототип:

```
DWORD  VirtualQuery(
  LPCVOID  lpAddress,      // адрес области
  PMEMORY_BASIC_INFORMATION  lpBuffer,  // буфер для информации
  DWORD    dwLength        // длина буфера
);
```

Параметр `lpAddress` указывает на область виртуальной памяти, информацию о которой нужно получить. При этом размер области определяется количеством последовательных виртуальных страниц, которые имеют одинаковые атрибуты.

Параметр `lpBuffer` указывает на структуру типа `MEMORY_BASIC_INFORMATION`, в которую функция `VirtualQuery` поместит информацию об указанной параметром `lpAddress` области виртуальной памяти.

Параметр `dwLength` должен содержать длину структуры, на которую указывает параметр `lpBuffer`.

После своего завершения функция VirtualQuery возвращает действительное количество байтов, записанных в структуру по адресу lpBuffer.

Прежде чем приводить пример программы, использующей функцию VirtualQuery, рассмотрим более подробно структуру типа MEMORY_BASIC_INFORMATION, которая состоит из следующих полей:

```
typedef  struct  _MEMORY_BASIC_INFORMATION {
  PVOID  BaseAddress;       // базовый адрес области виртуальной памяти
  PVOID  AllocationBase;    // базовый адрес распределенной памяти
  DWORD  AllocationProtect; // атрибуты доступа к распределенной памяти
  SIZE_T  RegionSize;       // размер области с одинаковыми атрибутами
  DWORD  State;      // состояние памяти
  DWORD  Protect;    // атрибуты доступа к области виртуальной памяти
  DWORD  Type;       // тип страниц в области виртуальной памяти
}  MEMORY_BASIC_INFORMATION,  *PMEMORY_BASIC_INFORMATION;
```

Значения всех этих полей, исключая последнее поле Type, совпадают со значениями параметров, которые задаются в функциях для работы с виртуальной памятью. Существует различие только в областях памяти, которые описывают эти поля.

Различие между значениями полей BaseAddress и AllocationBase заключается в том, что первое указывает на область виртуальной памяти, состояние которой мы хотим узнать, а второе — на область виртуальной памяти, которая была первоначально распределена функцией VirtualAlloc и содержит интересующую нас область.

Такое же различие существует и между назначением полей AllocationProtect и Protect. Поле AllocationProtect содержит атрибуты доступа к области памяти, распределенной функцией VirtualAlloc, а второе поле — атрибуты доступа области виртуальной памяти, о которой мы хотим получить информацию. Эти поля принимают те же значения, что и параметр flProtect при вызове функции VirtualAlloc.

Поле State принимает те же значения, что и параметр flAllocationType при вызове функции VirtualAlloc.

Поле Type описывает тип страниц в области виртуальной памяти, информация о которой определяется. Это поле может принимать одно из следующих значений:

- MEM_IMAGE — исполняемый код;
- MEM_MAPPED — файл, проецируемый в память;
- MEM_PRIVATE — память, принадлежащая процессу.

Теперь, в листинге 21.9, приведем программу, которая использует функцию VirtualQuery для определения состояния области виртуальной памяти.

Листинг 21.9. Инициализация и копирование блоков виртуальной памяти

```
#include <windows.h>
#include <iostream.h>

int main()
{
  BYTE  *a, *b;    // базовый адрес области и подобласти
  const int size = 10000;  // размер области
  const int shift = 5000;  // смещения для подобласти

  MEMORY_BASIC_INFORMATION  mbi;   // структура для информации
                                   // о виртуальной памяти
  DWORD  mbi_size = sizeof(MEMORY_BASIC_INFORMATION);

  // распределяем виртуальную память
  a = (BYTE*)VirtualAlloc(NULL, size, MEM_COMMIT, PAGE_READWRITE);
  if(!a)
  {
    cout << "Virtual allocation failed." << endl;
    return GetLastError();
  }

  // устанавливает адрес подобласти
  b = a + shift;

  // определяем информацию о виртуальной памяти
  if(mbi_size != VirtualQuery(b, &mbi, mbi_size))
  {
    cout << "Virual query failed." << endl;
    return GetLastError();
  }
  // распечатываем эту информацию
  cout << "Base address: " << mbi.BaseAddress << endl;
  cout << "Allocation base: " << mbi.AllocationBase << endl;
  cout << "Allocation protect: " << mbi.AllocationProtect << endl;
  cout << "Region size: " << mbi.RegionSize << endl;
  cout << "State: " << mbi.State << endl;
```

```
  cout << "Protect: " << mbi.Protect << endl;
  cout << "Type: " << mbi.Type << endl;

  // освобождаем виртуальную память
  if (!VirtualFree(a, 0, MEM_RELEASE))
  {
    cout << "Memory release failed." << endl;
    return GetLastError();
  }

  return 0;
}
```

21.8. Работа с виртуальной памятью в другом процессе

В операционных системах Windows существует набор следующих функций:

- `VirtualAllocEx` — распределить виртуальную память (Windows NT/2000);
- `VirtualFreeEx` — освободить виртуальную память (Windows NT/2000);
- `VirtualProtectEx` — изменить атрибуты доступа;
- `VirtualQueryEx` — определить состояние области.

Эти функции предназначены для работы с виртуальной памятью в адресном пространстве другого процесса. Причем первые две работают только в операционных системах Windows NT/2000. Имена этих функций отличаются от имен рассмотренных нами функций только суффиксом `Ex`. Список параметров этих функций содержит такие же параметры, как и рассмотренные выше, но дополнительным первым параметром является дескриптор процесса, с адресным пространством которого работает функция.

В листингах 21.10 и 21.11 приведены программы, которые используют некоторые из этих функций. Это два консольных процесса, которые обмениваются сообщениями через виртуальную память одного из них. В этом случае программа, которая осуществляет доступ к виртуальной памяти другого процесса, должна использовать функции `ReadProcessMemory` и `WriteProcessMemory`.

Функция `ReadProcessMemory`, предназначенная для чтения данных из виртуальной памяти другого процесса, имеет следующий прототип:

```
BOOL  ReadProcessMemory(
  HANDLE   hProcess,              // дескриптор процесса
  LPCVOID  lpBaseAddress,         // базовый адрес области
```

```
  LPVOID   lpBuffer,              // буфер данных
  DWORD    nSize,                 // количество считываемых байтов
  LPDWORD  lpNumberOfBytesRead    // количество прочитанных байтов
);
```

Функция `WriteProcessMemory`, предназначенная для записи данных в виртуальную память другого процесса, имеет следующий прототип:

```
BOOL  WriteProcessMemory(
  HANDLE    hProcess,             // дескриптор процесса
  LPCVOID  lpBaseAddress,         // базовый адрес области
  LPVOID   lpBuffer,              // буфер данных
  DWORD    nSize,                 // количество записываемых байтов
  LPDWORD  lpNumberOfBytesRead    // количество прочитанных байтов
);
```

Обе эти функции в случае успешного завершения возвращают ненулевое значение, а в случае неудачи — значение `FALSE`. Опишем назначение параметров этих функций. Назначение и тип первых двух параметров совпадают для обеих функций, а последние три параметра имеют одинаковый тип, но различаются по смыслу.

Параметр `hProcess` должен содержать дескриптор процесса, к виртуальной памяти которого обращается функция.

Параметр `lpBaseAddress` должен указывать на область виртуальной памяти, к которой обращается функция.

Параметр `lpBuffer` указывает на область памяти, в которую функция `ReadProcessMemory` будет читать данные из виртуальной памяти другого процесса, а функция `WriteProcessMemory` записывать данные в виртуальную память другого процесса.

Параметр `nSize` должен содержать количество байтов, которые функция `ReadProcessMemory` читает из виртуальной памяти другого процесса, а функция `WriteProcessMemory` записывает в виртуальную память другого процесса.

Параметр `lpNumberOfBytes` должен указывать на двойное слово, в которое вызываемая функция `ReadProcessMemory` или `WriteProcessMemory` поместит соответственно количество прочитанных или записанных байтов. Этот параметр может быть установлен в значение `NULL` и тогда вызываемая функция игнорирует его.

Теперь, в листингах 21.10 и 21.11, приведем две программы, первая из которых создает консольный процесс, а затем пишет и читает данные из виртуальной памяти этого процесса. Адрес виртуальной памяти, по которому дочерний консольный процесс должен записать ответное сообщение, передается этому процессу через командную строку. Кроме того, отметим, что программы

синхронизируют свой доступ к виртуальной памяти, чтобы корректно передать и получить сообщения.

Листинг 21.10. Доступ к виртуальной памяти другого процесса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char c;    // служебный символ
  char lpszCommandLine[80];            // командная строка
  char send[] = "This is a message."; // строка для пересылки
  char buffer[80];                     // буфер для ответа
  LPVOID  v = (LPVOID)0x00880000;      // указатель на область памяти

  HANDLE  hWrite, hRead;     // события для синхронизации
                             // записи-чтения в виртуальную память
  char  WriteEvent[] = "WriteEvent";
  char  ReadEvent[] = "ReadEvent";

  // создаем события
  hWrite = CreateEvent(NULL, FALSE, FALSE, WriteEvent);
  hRead = CreateEvent(NULL, FALSE, FALSE, ReadEvent);


  STARTUPINFO si;
  PROCESS_INFORMATION pi;

  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb = sizeof(STARTUPINFO);
  // формируем командную строку
  wsprintf(lpszCommandLine, "C:\\ConsoleProcess.exe %d", (int)v);
  // создаем новый консольный процесс
  if (!CreateProcess(NULL, lpszCommandLine, NULL, NULL, FALSE,
      CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi))
  {
    cout << "Create process failed." << endl;
    return GetLastError();
```

```
  }

  // распределяем виртуальную память в этом процессе
  v = VirtualAllocEx(
      pi.hProcess,
      v,
      sizeof(send),
      MEM_RESERVE | MEM_COMMIT,
      PAGE_READWRITE);
  if(!v)
  {
    cout << "Virtual allocation failed." << endl;
    return GetLastError();
  }

  // записываем в виртуальную память сообщение
  WriteProcessMemory(pi.hProcess, v, (void*)send, sizeof(send), NULL);
  // оповещаем о записи
  SetEvent(hWrite);
  // ждем сигнала на чтение
  WaitForSingleObject(hRead, INFINITE);
  // читаем ответ
  ReadProcessMemory(pi.hProcess, v, (void*)buffer, sizeof(buffer), NULL);
  // выводим ответ
  cout << buffer << endl;

  // освобождаем виртуальную память
  if (!VirtualFreeEx(pi.hProcess, v, 0, MEM_RELEASE))
  {
    cout << "Memory release failed." << endl;
    return GetLastError();
  }

  cout << "Input any char to exit: ";
  cin >> c;

  return 0;
}
```

Листинг 21.11. Работа с виртуальной памятью, захваченной другим процессом

```
#include <windows.h>
#include <iostream.h>

int main(int argc, char *argv[])
{
  char c;
  char  answer[] = "This is an answer.";
  HANDLE  hWrite, hRead;   // события для синхронизации
  char  WriteEvent[] = "WriteEvent";
  char  ReadEvent[] = "ReadEvent";
  char  *v;      // для адреса виртуальной памяти

  // открываем события
  hWrite = OpenEvent(EVENT_MODIFY_STATE, FALSE, WriteEvent);
  hRead = OpenEvent(EVENT_MODIFY_STATE, FALSE, ReadEvent);

  // преобразуем параметр в адрес
  v = (char*)atoi(argv[1]);
  // выводим сообщение
  cout <<  v << endl;

  // ждем разрешения на запись
  WaitForSingleObject(hWrite, INFINITE);
  // записываем ответ
  strcpy(v, "This is an answer.");
  // разрешаем чтение
  SetEvent(hRead);

  // закрываем дескрипторы
  CloseHandle(hWrite);
  CloseHandle(hRead);

  // ждем команды на завершение
  cout << "Input any char to exit: ";
  cin >> c;

  return 0;
}
```

Глава 22

Работа с кучей в Windows

22.1. Создание и удаление кучи

Кучей или *пулом памяти* называется распределенная процессом область виртуальной памяти, используемая им для захвата и освобождения блоков памяти, размер которых меньше размера виртуальной страницы. Куча называется *сериализуемой*, если система синхронизирует доступ параллельно работающих потоков к этой куче. В Windows каждая куча имеет свой дескриптор и, следовательно, является объектом ядра.

Для каждого процесса Windows по умолчанию резервирует одну кучу размером в 1 Мбайт и сразу распределяет из нее 4 Кбайт виртуальной памяти для использования процессом. Функции `malloc` и `free` из стандартной библиотеки языка программирования C, а также операторы `new` и `delete` языка программирования C++ распределяют память из кучи, зарезервированной для процесса по умолчанию.

Дескриптор кучи, созданной для процесса по умолчанию, можно получить при помощи функции `GetProcessHeap`, которая имеет следующий прототип:

```
HANDLE  GetProcessHeap(VOID);
```

В случае успешного завершения эта функция возвращает дескриптор кучи, а в случае неудачи — значение `NULL`.

Кроме того, процесс может создавать кучи динамически во время своей работы. Динамическое создание куч, как правило, направлено на ускорение работы приложения с динамически распределяемой памятью. Эта цель достигается двумя приемами работы с динамически созданной кучей. Во-первых, динамически созданная куча используется для хранения только однотипных объектов. Так как в этом случае исключается фрагментация кучи, то динамическое распределение памяти выполняется быстрее, чем в обычной куче. Да и память в этом случае используется более экономно. Во-вторых, динамически созданная куча может быть несериализуемой, что также ускоряет работу с этой кучей, т. к. не нужно синхронизировать доступ

потоков к такой куче. Но в этом случае кучу может использовать только один поток. Для доступа нескольких потоков к такой куче эти потоки должны выполнять синхронизацию самостоятельно.

Для динамического создания кучи используется функция `HeapCreate`, которая имеет следующий прототип:

```
HANDLE  HeapCreate(
  DWORD   flOptions,         // атрибуты распределения кучи
  SIZE_T  dwInitialState,    // начальное состояние
  SIZE_T  dwMaximumSize      // максимальный размер кучи
);
```

В случае успешного завершения эта функция возвращает дескриптор кучи, а в случае неудачи — значение `NULL`.

Параметр `flOptions` задает дополнительные атрибуты для создаваемой кучи. Эти атрибуты определяются следующими флагами, которые могут быть установлены в любой комбинации:

- `HEAP_GENERATE_EXCEPTIONS` — в случае ошибки функции, которая работает с кучей, система будет генерировать исключение, а не возвращать `NULL`, как она делает это по умолчанию;
- `HEAP_NO_SERIALIZE` — определяет, что куча является сериализуемой (по умолчанию куча не является сериализуемой).

Параметр `dwInitialSize` задает начальный размер физической памяти, которая распределяется куче. Этот размер задается в байтах и округляется системой в большую сторону до кратности размеру виртуальной страницы.

Параметр `dwMaximumSize` задает максимальный размер кучи в байтах, который округляется системой в большую сторону до кратности размеру виртуальной страницы и не может превышать величины в `0x7FFF8` байт. Если параметр `dwMaximumSize` определяет конкретный размер кучи, то размер кучи не может превысить этот размер. Если же параметр `dwMaximumSize` установлен в `0`, то размер кучи ограничен только доступной виртуальной памятью.

Для уничтожения кучи нужно использовать функцию `HeapDestroy`, которая имеет следующий прототип:

```
BOOL  HeapDestroy(
  HANDLE  hHeap    // дескриптор кучи
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Здесь параметр `hHeap` задает дескриптор уничтожаемой кучи. Функцию `HeapDestroy` не нужно применять к куче, созданной для процесса по умолчанию, т. к. в этом случае функция возвратит значение `TRUE`, но сама куча уничтожена не будет.

22.2. Распределение и освобождение памяти из кучи

Для распределения памяти из кучи используется функция HeapAlloc, которая имеет следующий прототип:

```
LPVOID  HeapAlloc(
  HANDLE  hHeap,     // дескриптор кучи
  DWORD   dwFlags,   // управляющие флаги
  SIZE_T  dwBytes    // размер распределяемой памяти
);
```

В случае успешного завершения эта функция возвращает адрес распределенной памяти, а в случае неудачи возможны два варианта работы функции. Если флаг HEAP_GENERATE_EXCEPTIONS не установлен, то функция HeapAlloc в случае неудачи возвращает значение NULL. Если же этот флаг установлен, то в случае неудачи эта функция генерирует одно из следующих исключений:

- STATUS_NO_MEMORY — не хватает памяти или повреждение кучи;
- STATUS_ACCESS_VIOLATION — повреждение кучи или неправильные параметры функции.

Теперь кратко опишем назначение параметров этой функции. В параметре hHeap должен быть установлен дескриптор кучи, из которой распределяется память.

Параметр dwFlags управляет режимом работы функции и может быть установлен в любую комбинацию из следующих управляющих флагов:

- HEAP_GENERATE_EXCEPTIONS — в случае неудачи функция сгенерирует исключение;
- HEAP_NO_SERIALIZE — нет взаимного исключения при доступе к куче;
- HEAP_ZERO_MEMORY — распределенная память инициализируется нулями.

Параметр dwBytes задает в байтах размер памяти, которая будет распределена из кучи.

Если память, распределенная из кучи, больше не используется программой, то ее нужно вернуть обратно в кучу, т. е. — освободить. Для освобождения памяти, распределенной из кучи, используется функция HeapFree, которая имеет следующий прототип:

```
BOOL  HeapFree(
  HANDLE  hHeap,     // дескриптор кучи
  DWORD   dwFlags,   // управляющие флаги
  LPVOID  lpMem      // адрес памяти
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — значение FALSE.

Параметр hHeap этой функции задает дескриптор кучи, из которой освобождается память, а параметр lpMemory — адрес этой памяти.

Параметр dwFlags задает флаги, которые управляют режимом работы функции. В настоящее время в этом параметре может быть установлен только один флаг:

- HEAP_NO_SERIALIZE — нет взаимного исключения при доступе к куче.

Теперь приведем программы, которые иллюстрируют работу с кучей. Сначала рассмотрим программу из листинга 22.1, которая распределяет, а затем освобождает память из кучи процесса.

Листинг 22.1. Распределение и освобождение памяти из кучи процесса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hHeap;
  int *a = NULL;     // указатель на массив
  int size = 1000;   // размерность массива

  // получаем дескриптор кучи процесса, созданной по умолчанию
  hHeap = GetProcessHeap();
  if (!hHeap)
  {
    cout << «Heap create failed.» << endl;
    return GetLastError();
  }
  // распределяем память под массив
  a = (int*)HeapAlloc(hHeap, HEAP_ZERO_MEMORY, size * sizeof(int));
  // распечатываем один элемент массива
  cout << «a[10] = « << a[10] << endl;
  // освобождаем память из кучи
  if (!HeapFree(hHeap, NULL, a))
  {
    cout << «Heap free failed.» << endl;
```

```
    return GetLastError();
  }

  return 0;
}
```

Теперь приведем в листинге 22.2 программу, которая выполняет такие же операции, но с кучей, созданной динамически.

Листинг 22.2. Распределение и освобождение памяти из кучи, созданной динамически

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hHeap;
  int  *a = NULL;         // указатель на массив
  int  h_size = 4096;     // размер кучи
  int  a_size = 2048;     // размер массива

  // создаем кучу динамически
  hHeap = HeapCreate(HEAP_NO_SERIALIZE, h_size, h_size);
  if (!hHeap)
  {
    cout << "Heap create failed." << endl;
    return GetLastError();
  }
  // распределям память под массив
  a = (int*)HeapAlloc(hHeap, NULL, a_size);
  // обрабатываем ошибку в случае неудачи
  if (!a)
  {
    cout << "Heap allocation failed." << endl;
    return GetLastError();
  }
  // распечатываем распределенную память
  cout << "a[10] = " << a[10] << endl;
```

```
  // освобождаем память из кучи
  if (!HeapFree(hHeap, NULL, a))
  {
    cout << "Heap free failed." << endl;
    return GetLastError();
  }
  // разрушаем кучу
  if (!HeapDestroy(hHeap))
  {
    cout << "Heap destroy failed." << endl;
    return GetLastError();
  }

  return 0;
}
```

Различие между этими двумя программами состоит в следующем. Во-первых, кучу процесса не нужно создавать и, следовательно, затем уничтожать. Но эти операции необходимо выполнить с кучей, создаваемой динамически. Во-вторых, при распределении памяти из кучи процесса был установлен флаг `HEAP_ZERO_MEMORY`, поэтому распределенная память была инициализирована нулями. При работе же с кучей, созданной динамически, этот флаг не был установлен. Поэтому элементы массива содержат непредсказуемые значения.

Теперь рассмотрим программы, которые показывают, как обрабатывать ошибки при работе с кучей в зависимости от того, установлен ли флаг `HEAP_GENERATE_EXCEPTIONS`. Приведенная в листинге 22.3 программа не устанавливает этот флаг и поэтому обрабатывает ошибку нехватки памяти, проверяя код возврата из функции `HeapAlloc`.

Листинг 22.3. Обработка ошибки при распределении памяти проверкой кода возврата

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hHeap;
  int  *a = NULL;      // указатель на массив
```

```
  int  size = 4096;    // размер массива, а также кучи

  // создаем кучу динамически
  hHeap = HeapCreate(HEAP_NO_SERIALIZE, size, size);
  if (!hHeap)
  {
    cout << "Heap create failed." << endl;
    return GetLastError();
  }
  // пытаемся распределить память под массив
  a = (int*)HeapAlloc(hHeap, NULL, size * sizeof(int));
  // обрабатываем ошибку в случае неудачи
  if (!a)
  {
    cout << "Heap allocation failed." << endl;
    return GetLastError();
  }
  // разрушаем кучу
  if (!HeapDestroy(hHeap))
  {
    cout << "Heap destroy failed." << endl;
    return GetLastError();
  }

  return 0;
}
```

Теперь в листинге 22.4 приведем программу, которая устанавливает флаг HEAP_GENERATE_EXCEPTIONS и поэтому ошибка нехватки памяти обрабатывается, используя структурную обработку исключений.

Листинг 22.4. Обработка ошибки распределения памяти при помощи исключений

```
#include <windows.h>
#include <iostream.h>

int main()
```

```
{
  HANDLE  hHeap;
  int  *a = NULL;      // указатель на массив
  int  size = 4096;    // размер массива, а также кучи

  // создаем кучу динамически
  hHeap = HeapCreate(HEAP_NO_SERIALIZE | HEAP_GENERATE_EXCEPTIONS,
    size, size);
  if (!hHeap)
  {
    cout << "Heap create failed." << endl;
    return GetLastError();
  }
  // пытаемся распределить память из кучи
  __try
  {
    a = (int*)HeapAlloc(hHeap, NULL, size * sizeof(int));
  }
  __except(EXCEPTION_EXECUTE_HANDLER)
  {
    DWORD ecode = GetExceptionCode();

    if (ecode == STATUS_NO_MEMORY)
      cout << "STATUS_NO_MEMORY exception." << endl;
    else
      cout << "Some exception." << endl;
  }
  // разрушаем кучу
  if (!HeapDestroy(hHeap))
  {
    cout << "Heap destroy failed." << endl;
    return GetLastError();
  }

  return 0;
}
```

22.3. Перераспределение памяти из кучи

Для перераспределения памяти в куче используется функция HeapReAlloc, которая имеет следующий прототип:

```
LPVOID  HeapReAlloc(
  HANDLE  hHeap,     // дескриптор кучи
  DWORD   dwFlags,   // управляющие флаги
  LPVOID  lpMem,     // адрес перераспределяемой памяти
  SIZE_T  dwBytes    // новый размер блока памяти в байтах
);
```

Она, как и функция HeapAlloc, в случае успешного завершения возвращает адрес перераспределенной памяти, а в случае неудачи ведет себя в зависимости от того, установлен ли флаг HEAP_GENERATE_EXCEPTIONS. Если этот флаг не установлен, то в случае неудачи эта функция возвращает значение NULL. В противном случае эта функция генерирует одно из таких же исключений, как и функция HeapAlloc:

- STATUS_NO_MEMORY — не хватает памяти или повреждение кучи;
- STATUS_ACCESS_VIOLATION — повреждение кучи или неправильные параметры функции.

Параметр hHeap функции HeapReAlloc должен указывать на дескриптор кучи, из которой распределяется память.

Параметр dwFlags управляет режимом работы функции и может быть установлен в любую комбинацию из следующих флагов:

- HEAP_GENERATE_EXCEPTIONS — в случае неудачи функция сгенерирует исключение;
- HEAP_NO_SERIALIZE — нет взаимного исключения при доступе к куче;
- HEAP_REALLOC_IN_PLACE_ONLY — перераспределяемый блок памяти передвигать на новое место нельзя, по умолчанию блок может быть переписан на новое место;
- HEAP_ZERO_MEMORY — дополнительная распределенная память инициализируется нулями.

Параметр lpMem задает адрес памяти, которая перераспределяется вызовом функции HeapReAlloc. Этот адрес должен быть предварительно возвращен одной из функций HeapAlloc или HeapReAlloc.

Параметр dwBytes задает размер памяти в байтах, которая будет перераспределена из кучи.

Теперь приведем в листинге 22.5 программу, в которой перераспределяется память из кучи. Отметим, что в этой программе при первом распределении

память не инициализируется, а при перераспределении добавленная память инициализируется нулями.

Листинг 22.5. Перераспределение памяти из кучи

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hHeap;
  int  *a = NULL;         // указатель на массив
  int  h_size = 4096;     // размер кучи
  int  a_size = 5;        // размер массива

  // создаем кучу динамически
  hHeap = HeapCreate(HEAP_NO_SERIALIZE | HEAP_GENERATE_EXCEPTIONS,
    h_size, 0);
  if (!hHeap)
  {
    cout << "Heap create failed." << endl;
    return GetLastError();
  }
  // распределяем память из кучи
  a = (int*)HeapAlloc(hHeap, NULL, a_size * sizeof(int));
  // инициализируем и распечатываем массив
  for (int i = 0; i < a_size; i++)
  {
    a[i] = i;
    cout << "a[" << i << "] = " << a[i] << endl;
  }
  // распределяем дополнительный блок
  a = (int*)HeapReAlloc(hHeap, HEAP_ZERO_MEMORY, a,
                2 * a_size * sizeof(int));
  // распечатываем элементы массива
  for (i = 0; i < 2 * a_size; i++)
    cout << "\ta[" << i << "] = " << a[i] << endl;
  // разрушаем кучу
  if (!HeapDestroy(hHeap))
```

```
  {
    cout << "Heap destroy failed." << endl;
    return GetLastError();
  }

  return 0;
}
```

22.4. Блокирование и разблокирование кучи

Если куча не является сериализуемой, то параллельный доступ нескольких потоков к этой куче может нарушить ее непротиворечивое состояние и вызвать ошибку в работе приложения. Чтобы избежать такой ситуации, в операционных системах Windows предусмотрены функции `HeapLock` и `HeapUnlock`, которые блокируют параллельный доступ нескольких потоков к куче.

Для того чтобы получить монопольный доступ к куче, поток должен вызвать функцию `HeapLock`, которая имеет следующий прототип:

```
BOOL  HeapLock(HANDLE  hHeap);
```

В случае успешного завершения эта функция блокирует доступ остальных потоков к куче и возвращает ненулевое значение, а в случае неудачи возвращает значение `FALSE`. Единственным параметром этой функции является дескриптор кучи, к которой поток хочет получить монопольный доступ.

Если куча заблокирована потоком при помощи функции `HeapLock` и другой поток вызывает какую-нибудь функцию для доступа к этой куче, то система переведет его в состояние ожидания до тех пор, пока поток, вызвавший функцию `HeapLock`, не вызовет функцию `HeapUnlock`, которая имеет следующий прототип:

```
BOOL  HeapUnlock(HANDLE  hHeap);
```

В случае успешного завершения эта функция разблокирует кучу и возвращает ненулевое значение, а в случае неудачи куча остается заблокированной и функция возвращает значение `FALSE`.

В заключение заметим, что эти функции поддерживаются только операционными системами Windows NT/2000.

Теперь приведем, в листинге 22.6, программу, которая вызывает функции `HeapLock` и `HeapUnlock` для блокирования и разблокирования созданной динамически кучи. В связи с этой программой отметим, что блокирование и разблокирование кучи можно выполнить только в том случае, если куча является сериализуемой.

Листинг 22.6. Блокирование и разблокирование кучи

```
#include <windows.h>
#include <iostream.h>

HANDLE  hHeap;

DWORD WINAPI thread(LPVOID)
{
  int  *a;
  // бесконечный цикл распределения и освобождения памяти из кучи
  for ( ; ; )
  {
    // распределяем память
    a = (int*)HeapAlloc(hHeap, NULL, sizeof(int));
    cout << "\tHeap allocated." << endl;
    // освобождаем память
    HeapFree(hHeap, NULL, a);
    cout << "\tHeap freed." << endl;
    // немного подождем
    Sleep(2000);
  }
}

int main()
{
  HANDLE   hThread;
  DWORD  IDThread;
  int  size = 4096;  // размер кучи
  char c;            // служебный символ

  // создаем кучу динамически
  hHeap = HeapCreate(NULL, size, 0);
  if (!hHeap)
  {
    cout << "Heap create failed." << endl;
    return GetLastError();
  }
```

```
// напечатаем как управлять программой
cout << "Input " << endl;
cout << "\tl - to lock the heap." << endl;
cout << "\tu - to unlock the heap." << endl;
cout << "\te - to exit from the process." << endl << endl;
// ждем ввод символа для продолжения выполнения программы
cout << "Now input any char to continue: ";
cin >> c;
// запускаем поток, работающий с кучей
hThread = CreateThread(NULL, 0, thread, NULL, NULL, &IDThread);
if (!hThread)
{
  cout << "Create thread failed." << endl;
  return GetLastError();
}
// блокируем/разблокируем кучу
for ( ; ; )
{
  cin >> c;
  switch (c)
  {
  case 'l':
    if (!HeapLock(hHeap))
    {
      cout << "Heap lock failed." << endl;
      return GetLastError();
    }
    cout << "Heap locked." << endl;
    break;

  case 'u':
    if (!HeapUnlock(hHeap))
    {
      cout << "Heap unlock failed." << endl;
      return GetLastError();
    }
    cout << "Heap unlocked." << endl;
    break;
```

```
    case 'e':
      // разрушаем кучу
      if (!HeapDestroy(hHeap))
      {
        cout << "Heap destroy failed." << endl;
        return GetLastError();
      }
      // прерываем выполнение потока thread
      TerminateThread(hThread, 0);
      // закрываем дескриптор потока
      CloseHandle(hThread);
      // выводим сообщение о завершении работы
      cout << "Exit from the process." << endl;

      return 0;
    }
  }
}
```

22.5. Проверка состояния кучи

Чтобы получить информацию о состоянии кучи или отдельного блока виртуальной памяти, распределенного из кучи, можно использовать функцию `HeapValidate`, которая имеет следующий прототип:

```
BOOL  HeapValidate(
  HANDLE   hHeap,    // дескриптор кучи
  DWORD    dwFlags,  // управляющие флаги
  LPCVOID  lpMem     // адрес блока памяти
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Успешное завершение этой функции означает, что куча или проверяемый блок памяти из кучи находятся в непротиворечивом состоянии. То есть адрес и длина отдельного блока или всех блоков в структурах, управляющих распределением памяти из кучи, соответствуют действительным значениям этих величин. Также отметим, что в случае неудачи функция `HeapValidate` не устанавливает дополнительную информацию об ошибке, которую можно получить, используя функцию `GetLastError`.

Параметр `hHeap` этой функции указывает на дескриптор кучи, состояние которой должна определить функция `HeapValidate`.

Параметр dwFlags может принимать значение HEAP_NO_SERIALIZE, которое говорит о том, что нет взаимного исключения при доступе к куче. Если это значение установлено, то во время работы функции HeapValidate доступ других потоков к куче блокируется. В противном случае блокировка одновременного доступа потоков к куче не происходит.

Параметр lpMem должен указывать на блок памяти, который был распределен из кучи и состояние которого проверяется. Если значение этого параметра равно NULL, то проверяется вся куча.

Отметим, что функция HeapValidate работает только в операционных системах Windows NT/2000.

Теперь приведем в листинге 22.7 программу, которая проверяет состояние блока памяти, распределенного из кучи.

Листинг 22.7. Проверка состояния блока кучи

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hHeap;
  int  *a = NULL;    // указатель на массив
  int  size = 4096;  // размер кучи

  // создаем кучу динамически
  hHeap = HeapCreate(HEAP_NO_SERIALIZE, size, 0);
  if (!hHeap)
  {
    cout << "Heap create failed." << endl;
    return GetLastError();
  }
  // распределяем память под массивы
  a = (int*)HeapAlloc(hHeap, NULL, 4 * sizeof(int));
  // проверяем состояние распределенного блока памяти
  if (!HeapValidate(hHeap, HEAP_NO_SERIALIZE, a))
    cout << "The block is bad." << endl;
  else
    cout << "The block is good." << endl;

  // разрушаем кучу
```

```
  HeapDestroy(hHeap);

  return 0;
}
```

Для получения более подробной информации о состоянии кучи нужно использовать функцию `HeapWalk`, которая записывает информацию о блоках виртуальной памяти из кучи в структуру типа:

```
typedef  struct  _PROCESS_HEAP_ENTRY {
  PVOID  lpData,       // адрес элемента данных
  DWORD  cbData,       // длина элемента данных в байтах
  BYTE   cbOverhead,   // длина данных, описывающих элемент данных
  BYTE   iRegionIndex, // индекс области, содержащей элемент данных
  WORD   wFlags,       // управляющие флаги
  union  {
    struct  {
      HANDLE  hMem,    // дескриптор распределенного блока памяти
      DWORD   dwReserved[3];  // не используется
    }  Block;
    struct  {
      DWORD   dwCommittedSize;   // длина распределенной памяти
      DWORD   dwUnCommittedSize; // длина свободной памяти
      LPVOID  lpFirstBlock;      // адрес первого занятого блока
      LPVOID  lpLastBlock;       // адрес первого свободного блока
    }  Region;
  }
}  PROCESS_HEAP_ENTRY,  *LPPROCESS_HEAP_ENTRY;
```

Прежде чем описать назначение полей этой структуры, рассмотрим поле `wFlags`, от возможных значений которого зависит назначение других полей этой структуры.

Поле `wFlags` может принимать следующие значения:

- `PROCESS_HEAP_REGION` — элемент кучи находится в начале непрерывной области виртуальной памяти;
- `PROCESS_HEAP_UNCOMMITTED_RANGE` — элемент кучи находится в области нераспределенной виртуальной памяти, которая зарезервирована для кучи;
- `PROCESS_HEAP_ENTRY_BUSY` — элемент кучи является распределенным блоком памяти.

Кроме того, совместно с флагом PROCESS_HEAP_ENTRY_BUSY могут использоваться следующие флаги:

- PROCESS_HEAP_ENTRY_MOVEABLE — блок памяти является перемещаемым;
- PROCESS_HEAP_ENTRY_DDESHARE — блок памяти разделяется протоколом DDE.

Теперь рассмотрим назначение полей lpData, cbData, cbOverhead структуры типа PROCESS_HEAP_ENTRY.

Если в поле wFlags установлено значение PROCESS_HEAP_REGION, то содержимое этих полей интерпретируется следующим образом:

- поле lpData указывает на первый виртуальный адрес, используемый в области виртуальной памяти;
- поле cbData содержит общий размер в байтах области виртуальной памяти;
- поле cbOverhead содержит размер в байтах управляющей структуры, которая описывает область виртуальной памяти.

Если же в поле wFlags установлено значение PROCESS_HEAP_UNCOMMITTED_RANGE, то содержимое этих полей интерпретируется следующим образом:

- поле lpData указывает на начальный виртуальный адрес нераспределенной области виртуальной памяти;
- поле cbData содержит общий размер в байтах нераспределенной области виртуальной памяти;
- поле cbOverhead содержит размер в байтах управляющей структуры, которая описывает нераспределенную область виртуальной памяти.

Анонимное объединение, которое расположено после поля wFlags, содержит две перекрывающиеся структуры типов Block и Region. Если в поле wFlags установлены значения PROCESS_HEAP_ENTRY_BUSY и PROCESS_HEAP_ENTRY_MOVEABLE, то анонимное объединение содержит структуру Block. Поля этой структуры описывают следующие значения:

- поле hMem содержит дескриптор распределенного и перемещаемого блока виртуальной памяти;
- поле dwReserved является зарезервированным и не используется.

Если же в поле wFlags установлено значение PROCESS_HEAP_REGION, то объединение содержит структуру Region, поля которой описывают следующие значения:

- поле dwCommittedSize содержит длину блока или длину управляющей структуры, которая описывает блок;
- поле dwUnCommittedSize содержит длину блока, который не распределен процессу;
- поле lpFirstBlock указывает на первый занятый блок памяти в куче;
- поле lpLastBlock указывает на первый неиспользуемый блок памяти в куче.

Теперь можно перейти к рассмотрению функции HeapWalk, которая позволяет последовательно просматривать информацию о блоках памяти в куче при помощи структур типа PROCESS_HEAP_ENTRY. Эта функция имеет следующий прототип:

```
BOOL  HeapWalk(
  HANDLE  hHeap,                  // дескриптор кучи
  LPPROCESS_HEAP_ENTRY  lpEntry  // адрес структуры PROCESS_HEAP_ENTRY
);
```

В случае удачного завершения функция HeapWalk возвращает значение TRUE и записывает состояние элемента кучи в структуру PROCESS_HEAP_ENTRY, адрес которой задается параметром lpEntry. В случае неудачи функция возвращает значение FALSE.

Чтобы начать просмотр элементов кучи, нужно установить значение поля lpData в структуре PROCESS_HEAP_ENTRY в значение NULL. Для продолжения просмотра состояния элементов кучи нужно последовательно вызывать функцию HeapWalk, не изменяя значения параметров hHeap, lpEntry и содержимого полей структуры PROCESS_HEAP_ENTRY. При вызове функции после просмотра последнего элемента кучи функция HeapWalk возвращает значение FALSE и последующий вызов функции GetLastError вернет значение ERROR_NO_MORE_ITEMS.

Теперь приведем в листинге 22.8 программу, которая проверяет состояние элементов кучи, используя для этого функцию HeapWalk.

Листинг 22.8. Проверка состояния элементов кучи

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hHeap;
  int  size = 4096;              // размер кучи
  int  *a = NULL, *b = NULL;     // указатели на массивы
  PROCESS_HEAP_ENTRY  phe;       // состояние элемента кучи

  // создаем кучу динамически
  hHeap = HeapCreate(HEAP_NO_SERIALIZE, size, 0);
  // распределяем память под массивы
  a = (int*)HeapAlloc(hHeap, NULL, 4 * sizeof(int));
```

```
  b = (int*)HeapAlloc(hHeap, NULL, 16 * sizeof(int));
  // инициализируем цикл проверки состояния кучи
  phe.lpData = NULL;
  // проверяем состояние элементов кучи
  while (HeapWalk(hHeap, &phe));
  {
    if (phe.wFlags & PROCESS_HEAP_REGION)
      cout << "PROCESS_HEAP_REGION flag is set." << endl;
    if (phe.wFlags & PROCESS_HEAP_UNCOMMITTED_RANGE)
      cout << "PROCESS_HEAP_UNCOMMITTED_RANGE flag is set" << endl;
    cout << "lpData = " << phe.lpData << endl;
    cout << "cbData = " << phe.cbData << endl;
    cout << endl;
  }
  // разрушаем кучу
  HeapDestroy(hHeap);

  return 0;
}
```

22.6. Уплотнение кучи

Если при работе с кучей распределяются и освобождаются блоки памяти разной длины, то со временем происходит фрагментация кучи, т. е. внутри кучи образуются свободные блоки. Это происходит потому, что длина свободного блока в куче не всегда соответствует длине распределяемого блока памяти. Фрагментация кучи приводит к двум последствиям. Во-первых, к неэффективному использованию памяти, т. к. внутри кучи находятся свободные неиспользуемые блоки виртуальной памяти, которые часто не могут быть использованы для распределения памяти из кучи из-за своей малой длины. Во-вторых, к более медленной работе с кучей, т. к. для распределения блока памяти требуется просмотр списка свободных блоков кучи. Чтобы частично избежать этих последствий, необходимо выполнить уплотнение кучи, которое заключается в соединении последовательных свободных блоков памяти в куче в один блок большей длины и освобождении больших блоков свободной виртуальной памяти.

Для уплотнения кучи используется функция `HeapCompact`, которая имеет следующий прототип:

```
  UINT  HeapCompact(
    HANDLE  hHeap,     // дескриптор кучи
```

```
    DWORD   dwFlags    // управляющие флаги
  );
```

В случае успешного завершения эта функция возвращает длину наибольшего свободного блока памяти в куче (в байтах), а в случае неудачи — ноль. Параметр `hHeap` указывает на дескриптор кучи, которую нужно уплотнить, а в параметре `dwFlags` может быть установлено значение:

- `HEAP_NO_SERIALIZE` — нет взаимного исключения при доступе к куче.

Если это значение установлено, то во время работы функции `HeapCompact` доступ других потоков к куче блокируется. В противном случае блокировка одновременного доступа потоков к куче не происходит. В заключение отметим, что эта функция работает только в операционных системах Windows NT/2000.

В листинге 22.9 приведена программа, которая уплотняет кучу, используя для этого функцию `HeapCompact`.

Листинг 22.9. Уплотнение кучи

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hHeap;
  int  *a = NULL;    // указатель на массив
  int  *b = NULL;    // указатель на массив
  int  *c = NULL;    // указатель на массив
  int  size = 4096;  // размер кучи
  UINT free;         // длина свободной памяти после уплотнения

  // создаем кучу динамически
  hHeap = HeapCreate(HEAP_NO_SERIALIZE, size, size);
  if (!hHeap)
  {
    cout << "Heap create failed." << endl;
    return GetLastError();
  }
    // распределяем память под массивы
  a = (int*)HeapAlloc(hHeap, NULL, 1024);
  cout << "a = " << a << endl;
  b = (int*)HeapAlloc(hHeap, NULL, 1024);
```

```
  cout << "b = " << b << endl;
  c = (int*)HeapAlloc(hHeap, NULL, 1024);
  cout << "c = " << c << endl << endl;

  // проверяем, сколько осталось свободного места в куче
  free = HeapCompact(hHeap, HEAP_NO_SERIALIZE);
  if (!free)
  {
    cout << "Heap compact failed." << endl;
    return GetLastError();
  }
  else
    cout << "Free: " << free << endl;

  // освобождаем первых два массива
  HeapFree(hHeap, NULL, b);
  HeapFree(hHeap, NULL, a);

  // уплотняем кучу
  free = HeapCompact(hHeap, HEAP_NO_SERIALIZE);
  if (!free)
  {
    cout << "Heap compact failed." << endl;
    return GetLastError();
  }
  else
    cout << "Free: " << free << endl;

  // разрушаем кучу
  HeapDestroy(hHeap);

  return 0;
}
```

Часть VII

Управление файлами

Глава 23

Общие концепции

23.1. Накопители на жестких магнитных дисках

Схематически накопитель на жестких магнитных дисках (НЖМД) с перемещаемыми головками показан на рис. 23.1. Основными элементами НЖМД являются круглые алюминиевые или некристаллические стекловидные пластины. Эти пластины нельзя согнуть и поэтому они называются *жесткими дисками*. Жесткие диски покрыты слоем ферромагнитного материала, который позволяет хранить информацию, используя направление магнитного поля. Жесткие диски также называются *жесткими магнитными дисками*. Жесткие магнитные диски закреплены на стержне, который вращается с большой скоростью. Данные записываются на поверхностях жестких магнитных дисков с помощью магнитных головок, которые расположены над каждой дисковой поверхностью. Магнитной головке доступны только те данные, которые находятся на участке дисковой поверхности под или над ней. Все магнитные головки закреплены на одном стержне, который, совершая вращательные движения, перемещает головки по радиусам магнитных дисков в обоих направлениях. Если магнитная головка не перемещается, то она описывает на дисковой поверхности окружность, которая называется *дорожкой*. Дорожки нумеруются от 0 до *n*, где дорожка с индексом 0 имеет наибольший радиус. Группа дорожек, находящихся под всеми магнитными головками в каком-то конкретном положении стержня с магнитными головками, называется *цилиндром*.

Теперь кратко поясним, как выполняются операции чтения и записи данных на магнитные диски. Но прежде дадим определение термину *доступ к данным* — под ним будем понимать операции записи данных на магнитные диски и чтения данных с магнитных дисков. Чтобы получить доступ к данным на магнитном диске, необходимо выполнить следующие операции:

- установить магнитные головки на соответствующий цилиндр;

- дождаться, пока под магнитной головкой окажется точка на вращающемся магнитном диске, с которой начинаются данные;
- прочитать или записать данные на магнитный диск во время его вращения.

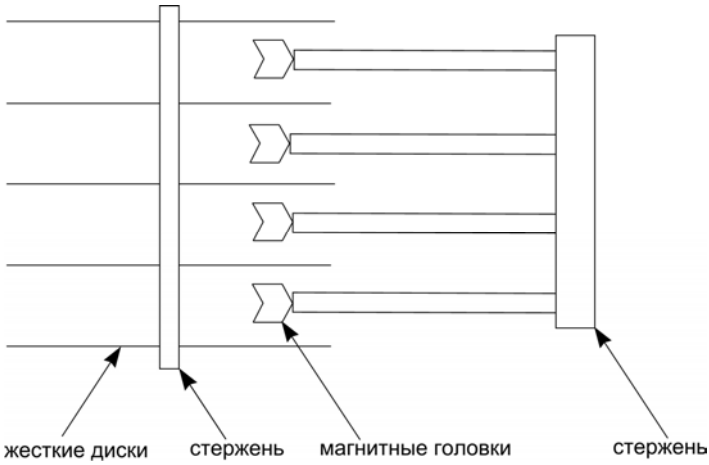


Рис. 23.1. Схема накопителя на жестких магнитных дисках

Так как каждая из этих операций связана с механическим движением, то общее время доступа к данным на магнитном диске достаточно велико по сравнению со скоростью работы интегральных схем.

23.2. Секторы и кластеры

Сектором называется наименьшая область (дуга) одной дорожки магнитного диска, которая может быть записана или считана магнитной головкой диска за его один полный поворот. Размер сектора равен 512 байт. Обычно плотность записи данных на диск является одинаковой для всех дорожек. Поэтому дорожки, находящиеся ближе к центру диска, содержат меньше секторов, чем дорожки, находящиеся ближе к краю диска. Одна дорожка жесткого диска может содержать от 380 до 700 секторов. Секторы каждой дорожки перенумерованы, причем эта нумерация начинается с 1. В начале каждого сектора хранится *заголовок* или *префикс*, который определяет начало и номер сектора. В конце каждого сектора хранится *заключение* или *суффикс*, который содержит контрольную сумму, необходимую для проверки целостности данных. Не так давно нумерация секторов одной дорожки не являлась последовательной. Секторы нумеровались через один сектор. Это было сделано для того, чтобы контроллер дискового устройства при доступе к жесткому магнитному диску не пропускал секторы при вращении. В настоящее время скорость работы контроллеров жестких дисков сравнима со скоростью

вращения диска, поэтому секторы каждой дорожки нумеруются последовательно.

Так как дорожки содержат разное количество секторов, то скорость обмена данными с накопителем на жестких магнитных дисках зависит от номера дорожки. Это вызвано тем, что при постоянной угловой скорости вращения стержня с дисками линейная скорость перемещения секторов относительно головки различается для разных дорожек. Эта скорость тем выше, чем дальше дорожка находится от центра диска. Поэтому диск разбивают на *зоны*. Одна зона содержит несколько цилиндров диска и на каждой из дорожек, входящих в одну зону, находится одинаковое количество секторов. Поэтому доступ к секторам одной зоны выполняется с одинаковой скоростью.

Теперь перейдем к кластерам. *Кластером* называется наименьшая область магнитного диска, которая может быть записана или прочитана операционной системой на диск. Обычно кластер состоит из нескольких секторов, имеющих последовательные номера.

23.3. Форматирование дисков

Прежде чем использовать жесткий магнитный диск для хранения информации, он должен быть отформатирован. Существуют три уровня форматирования жесткого магнитного диска:

- физическое форматирование или *форматирование низкого уровня*;
- *разбиение диска на разделы*;
- логическое форматирование или *форматирование высокого уровня*.

Форматирование низкого уровня (или низкоуровневое) заключается в разбиении дорожек на секторы. При этом формируются префикс, область данных и суффикс каждого сектора, а также интервалы между секторами и дорожками. Область данных сектора заполняется фиктивными данными или специальными тестовыми наборами данных.

При разбиении диска на разделы диск разбивается на области, которые также называются *разделами* или *логическими дисками*. В каждом разделе может быть установлена своя файловая система. В настоящее время операционные системы Windows используют файловые системы FAT16, FAT32 и NTFS.

При форматировании высокого уровня операционная система записывает в раздел структуры данных, необходимые для работы с файлами. В каждый раздел записывается загрузочный сектор (Boot Sector), который используется для загрузки операционной системы, таблицы размещения файлов и корневой каталог (Root Directory).

23.4. Функции файловой системы

Теперь перейдем к описанию файлов. *Логической записью* или *структурой* называется упорядоченное множество данных разных типов. Порядок следования этих данных называется *структурой записи*. На уровне прикладной программы *файл* представляет собой множество логических записей. На физическом уровне *файл* представляет собой поименованное множество секторов или кластеров, хранящихся на диске. Так как длина логической записи обычно не совпадает с длиной кластера, то кластер может содержать несколько логических записей или, наоборот, логическая запись может располагаться на нескольких кластерах. Часть операционной системы, которая обеспечивает доступ к файлам и выполняет связывание между логическими записями файла и их физическим представлением, называется *системой управления файлами* или *файловой системой*.

Для того чтобы выполнять операции доступа к логическим записям файла, с каждым файлом связывают *указатель файла*, который указывает на текущую логическую запись файла. После каждой операции записи или чтения логической записи файловая система передвигает указатель файла на следующую логическую запись. Для обеспечения доступа к файлам система управления файлами должна выполнять, по крайней мере, следующие функции:

- ❒ создание файла;
- ❒ удаление файла;
- ❒ открытие доступа к существующему файлу;
- ❒ закрытие доступа к существующему файлу;
- ❒ запись данных в файл;
- ❒ чтение данных из файла;
- ❒ установка указателя файла на нужную запись.

23.5. Каталоги

Каталогом называется файл, который содержит имена и местонахождение других файлов. Каталоги имеют древовидную структуру, в которой каждая вершина указывает на каталог, а каждый лист — на файл. Корень этой структуры называется корневым каталогом, который, как уже было сказано, создается системой на этапе форматирования высокого уровня и обычно обозначается символом \ (обратная косая). Файловая система обеспечивает следующие функции для работы с каталогами:

- ❒ создание каталога;
- ❒ удаление каталога;

- включение подкаталога в каталог;
- исключение подкаталога из каталога;
- включение файла в каталог;
- исключение файла из каталога.

23.6. Буферизация ввода-вывода

Буфером ввода-вывода называется область оперативной памяти, предназначенная для временного хранения записей файла. Обычно длина буфера выбирается кратной длине кластера. Буферы ввода-вывода предназначены для решения двух задач:

- устранение несоответствия между размером логической записи файла, определяемым в приложении, и размером кластера, который записывается на диск;
- снижение влияния внешних устройств на скорость работы процессора, которая значительно превышает скорость работы внешних устройств.

Для решения этих задач при выводе данных файловая система сначала полностью заполняет буфер логическими записями, а затем дает команду внешнему устройству на запись данных на диск. При вводе данных система управления файлами сначала заполняет буфер кластерами, а затем управляет чтением логических записей из буфера в программу пользователя.

Для ускорения ввода-вывода данных обычно используется несколько буферов ввода-вывода, которые организованы в кольцевую очередь. Во время работы пользовательского процесса с одним буфером файловая система параллельно осуществляет ввод или вывод данных в другие буферы.

23.7. Кэширование ввода-вывода

Рассмотрим два соседних уровня иерархии памяти, которые показаны на рис. 23.2. При этом предположим, что память верхнего уровня работает быстрее, чем память нижнего уровня. Считаем, что нужные данные хранятся в памяти нижнего уровня, а механизм управления памятью обеспечивает передачу на верхний уровень тех данных, к которым наиболее вероятно обращение программы пользователя. В таких случаях говорят, что верхний уровень памяти работает как *кэш* по отношению к нижнему уровню памяти. Вначале этот термин применялся только к памяти самого высокого уровня, которая располагается между регистрами процессора и оперативной памятью. В настоящее время этот термин используется в более широком смысле для обозначения упреждающего ввода данных.

Рис. 23.2. Иерархия памяти

Кэширование ввода данных подразумевает, что система выполняет упреждающее чтение данных с магнитного диска без ожидания следующей команды на чтение данных из приложения. Это сокращает время на чтение записей файла, если они читаются приложением последовательно.

В операционной системе Windows 2000 кэшированием ввода данных с диска занимается специальная программа ядра операционной системы, которая называется менеджером кэширования (Cash Manager).

Глава 24

Работа с файлами в Windows

24.1. Именование файлов в Windows

В операционных системах Windows полное имя файла представляется строкой, которая заканчивается пустым символом. Причем длина такой строки не может превышать `MAX_PATH` символов. Файловые системы FAT32 и NTFS поддерживают полные имена файлов длиной до 255 символов. Такие имена называются *длинными*. Файловая система FAT16, которая использовалась в операционной системе MS-DOS, поддерживает файловые имена длиной до 8 символов плюс 3 символа на расширение файла. Полное имя файла состоит из компонент (подстрок), каждая из которых разделяется символом \ (обратная косая). Каждая не последняя компонента полного имени файла задает имя каталога, в котором находится файл, а последняя компонента задает имя самого файла. Поэтому полное имя файла также часто называют *путем к файлу*, т. к. полное имя файла фактически описывает путь по дереву каталогов к его листу, который и представляет сам файл. При формировании пути к файлу нужно придерживаться определенных правил, которые перечислены ниже:

- имена каталогов и файлов не должны содержать символов, ASCII-коды которых находятся в диапазоне от 0 до 31 (это служебные символы);
- имена каталогов и файлов не должны содержать символы <, >, :, ", /, \ и |;
- имена каталогов и файлов могут содержать символы из расширенного множества, которое включает символы с кодами от 128 до 255;
- для обозначения текущего каталога в качестве компоненты пути используется символ . (точка);
- для обозначения родительского каталога для текущего каталога в качестве компоненты пути используются символы .. (две точки);
- в качестве компонент пути нельзя использовать имена устройств, как, например, `aux`, `con`, `lpt1` и `prn`.

Кроме того, отметим, что имена файлов нечувствительны к регистрам клавиатуры. То есть файловая система не различает имена файлов, которые отличаются только прописным или заглавным написанием букв. Например, имена `Demo`, `demo` и `DEMO` неразличимы.

24.2. Создание и открытие файлов

Для создания новых или открытия уже существующих файлов используется функция `CreateFile`, которая имеет следующий прототип:

```
HANDLE  CreateFile(
  LPCTSTR  lpFileName,             // имя файла
  DWORD    dwDesiredAccess,        // способ доступа
  DWORD    dwShareMode,            // режимы совместного использования
  LPSECUTITY_ATTRIBUTES  lpSecurutyAttributes,   // атрибуты защиты
  DWORD    dwCreationDisposition,  // создание или открытие файла
  DWORD    dwFlagsAndAttributes,   // флаги и атрибуты
  HANDLE   hTemplateFile           // файл атрибутов
);
```

В случае успешного завершения функция возвращает дескриптор созданного или открытого файла, а в случае неудачи — значение `INVALID_HANDLE_VALUE`.

В параметре `lpFileName` задается указатель на символьную строку, которая содержит полное имя создаваемого или открываемого файла. Если полное имя файла не указано, то файл с заданным именем создается или ищется в текущем каталоге.

Параметр `dwDesiredAccess` задает способ доступа к файлу и может принимать любую комбинацию следующих значений:

- `0` — приложение может только определять атрибуты устройства;
- `GENERIC_READ` — допускается только чтение данных из файла;
- `GENERIC_WRITE` — допускается только запись данных в файл.

Это общие или *родовые режимы* доступа к файлу. Существуют также и другие режимы, значения которых зависят от доступа, заданного при определении атрибутов защиты файла. Эти режимы будут рассмотрены в *разд. 45.5*, посвященном защите файлов. Пока же будем считать, что файл получает атрибуты защиты по умолчанию, что разрешает выполнять над ним все существующие операции.

Параметр `dwShareMode` задает режимы совместного доступа к файлу. Если значение этого параметра равно нулю, то файл не может использоваться для совместного доступа. Иначе параметр `dwShareMode` может принимать любую комбинацию следующих значений:

- `FILE_SHARE_READ` — файл может использоваться только для совместного чтения несколькими программами;

- FILE_SHARE_WRITE — файл может использоваться только для совместной записи несколькими программами;
- FILE_SHARE_DELETE — файл может использоваться несколькими программами при условии, что каждая из них имеет разрешение на удаление этого файла.

Отметим, что последнее значение может использоваться только в операционных системах Windows NT/2000.

Параметр lpSecurutyAttributes должен задавать атрибуты защиты файла. Пока этот параметр будем устанавливать в NULL. Это означает, что атрибуты защиты файла устанавливаются по умолчанию, т. е. дескриптор файла не является наследуемым и файл открыт для доступа всем пользователям.

Параметр dwCreationDisposition задает действия, которые нужно выполнить при создании или открытии файла. Этот параметр может принимать одно из следующих значений:

- CREATE_NEW — создать новый файл, если файл с заданным именем уже существует, то функция заканчивается неудачей;
- CREATE_ALWAYS — создать новый файл, если файл с заданным именем уже существует, то он уничтожается и создается новый файл;
- OPEN_EXISTING — открыть существующий файл, если файл с заданным именем не существует, то функция заканчивается неудачей;
- OPEN_ALWAYS — открыть файл, если файл с заданным именем не существует, то создается новый файл;
- TRUNCATE_EXISTING — открыть файл и уничтожить его содержимое, если файл с заданным именем не существует, то функция заканчивается неудачей.

Отметим, что в последнем случае вызывающий процесс должен иметь права записи в файл, т. е. в параметре dwDesiredAccess должен быть установлен флаг GENERIC_WRITE.

В параметре dwFlagsAndAttributes должны быть заданы флаги и атрибуты создаваемого или открываемого файла. Атрибуты файла управляют его свойствами и могут принимать любую комбинацию следующих значений:

- FILE_ATTRIBUTE_ARCHIVE — архивный файл, который содержит служебную информацию;
- FILE_ATTRIBUTE_ENCRYPTED — зашифрованный файл;
- FILE_ATTRIBUTE_HIDDEN — скрытый файл;
- FILE_ATTRIBUTE_NORMAL — обычный файл, который не имеет других атрибутов;
- FILE_ATTRIBUTE_NOT_CONTENT_INDEXED — содержимое файла не индексировано;

- FILE_ATTRIBUTE_OFFLINE — файл находится во вспомогательной памяти;
- FILE_ATTRIBUTE_READONLY — файл можно только читать;
- FILE_ATTRIBUTE_SYSTEM — файл используется операционной системой;
- FILE_ATTRIBUTE_TEMPORARY — файл используется для временного хранения данных.

Сделаем несколько замечаний относительно некоторых атрибутов файлов. Сначала отметим, что зашифрованные файлы не могут иметь также атрибут FILE_ATTRIBUTE_SYSTEM. Теперь заметим, что атрибут FILE_ATTRIBUTE_NORMAL должен использоваться только один, а не в комбинации с другими атрибутами.

Кроме того, в параметре dwFlagsAndAttributes может быть установлена любая комбинация следующих управляющих флагов:

- FILE_FLAG_WRITE_THROUGH — запись данных непосредственно на диск, не используя кэширования;
- FILE_FLAG_OVERLAPPED — обеспечивается асинхронное выполнение операций чтения и записи;
- FILE_FLAG_NO_BUFFERING — не использовать буферизацию при доступе к файлу;
- FILE_FLAG_RANDOM_ACCESS — программа предполагает выбирать записи из файла случайным образом;
- FILE_FLAG_SEQUENTIAL_SCAN — программа будет сканировать файл последовательно;
- FILE_FLAG_DELETE_ON_CLOSE — файл будет удален после того, как все дескрипторы этого файла будут закрыты;
- FILE_FLAG_BACKUP_SEMANTICS — резервный файл;
- FILE_FLAG_POSIX_SEMANTICS — доступ к файлу будет осуществляться по стандарту POSIX;
- FILE_FLAG_OPEN_REPARSE_POINT — при доступе к файлу используется системный фильтр;
- FILE_FLAG_OPEN_NO_RECALL — при использовании иерархической системы управления памятью файл не должен читаться в оперативную память, а оставаться на нижнем уровне иерархии.

Отметим, что флаг FILE_FLAG_BACKUP_SEMANTICS может использоваться только в операционных системах Windows NT/2000.

Параметр hTemplateFile используется при создании файла, атрибуты которого должны соответствовать атрибутам ранее созданного файла. В этом случае параметр hTemplateFile должен содержать дескриптор файла, атрибуты которого копируются в атрибуты создаваемого файла.

В остальных параметрах, кроме параметра `dwCreationDisposition`, может быть установлено значение 0. На платформах Windows 98/ME в этом параметре должно быть установлено значение `NULL`.

В заключение этого раздела скажем, что пример использования функции `CreateFile` приведен в *разд. 24.4*.

24.3. Закрытие и удаление файлов

Для закрытия доступа к файлу, как и для закрытия доступа к любому другому объекту ядра, используется функция `CloseHandle`, единственным параметром которой является дескриптор открытого файла.

Для физического удаления файла с диска используется функция `DeleteFile`, которая имеет следующий прототип:

```
BOOL  DeleteFile(
  LPCTSTR  lpFileName  // имя файла
);
```

Единственный параметр `lpFileName` является указателем на строку, которая указывает полный путь к файлу. При успешном завершении функция возвращает ненулевое значение, а при неудаче — `FALSE`.

В листинге 24.1 приведена программа, которая удаляет файл с именем demo_file.dat, который расположен в корневом каталоге на диске C:.

Листинг 24.1. Удаление файла

```
#include <windows.h>
#include <iostream.h>

int main()
{
  // удаляем файл
  if(!DeleteFile("C:\\demo_file.dat"))
  {
    cerr << "Delete file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();
    return 0;
  }
  cout << "The file is deleted." << endl;

  return 0;
}
```

24.4. Запись данных в файл

Для записи данных в файл служит функция `WriteFile`, причем отметим, что эта функция может использоваться как для синхронной, так и для асинхронной записи данных. В этом разделе будет рассмотрена только синхронная запись данных в файл. В этом случае данные записываются в файл последовательно — байт за байтом, и указатель файла передвигается по мере записи данных на новую позицию. Асинхронный ввод-вывод данных будет рассмотрен в *гл. 27*.

Теперь рассмотрим функцию `WriteFile` более подробно. Эта функция имеет следующий прототип

```
BOOL  WriteFile(
  HANDLE   hFile,                    // дескриптор файла
  LPCVOID  lpBuffer,                 // указатель на буфер данных
  DWORD    nNumberOfBytesToWrite,    // количество записываемых байтов
  LPDWORD  lpNumberOfBytesWritten,   // количество записанных байтов
  LPOVERLAPPED  lpOverlapped    // используется при асинхронной записи
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`.

Параметр `hFile` должен содержать дескриптор файла, причем файл должен быть открыт в режиме записи.

Параметр `lpBuffer` должен указывать на область памяти, в которую будут читаться данные.

Параметр `nNumberOfBytesToWrite` должен содержать количество байт, которые предполагается записать в файл посредством вызова функции `WriteFile`.

Параметр `lpNumberOfBytesWritten` должен содержать адрес памяти, в которую функция `WriteFile` поместит количество фактически записанных байт. При работе на платформе Windows 98 этот параметр должен иметь значение, отличное от `NULL`. При выполнении функции `WriteFile` операционная система записывает по этому адресу ноль, прежде чем выполнить запись данных в файл.

Так как в этом разделе не будет рассматриваться асинхронная запись данных в файл, то параметр `lpOverlapped` будет устанавливаться в `NULL`. Подробно асинхронный ввод-вывод данных будет рассмотрен в *гл. 27*.

В листинге 24.2 приведена программа, которая создает файл и записывает в него последовательность целых чисел.

Листинг 24.2. Создание файла и запись в него данных

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE hFile;

  // создаем файл для записи данных
  hFile = CreateFile(
    "C:\\demo_file.dat",    // имя файла
    GENERIC_WRITE,          // запись в файл
    0,                      // монопольный доступ к файлу
    NULL,                   // защиты нет
    CREATE_NEW,             // создаем новый файл
    FILE_ATTRIBUTE_NORMAL, // обычный файл
    NULL                    // шаблона нет
  );
  // проверяем на успешное создание
  if (hFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();
    return 0;
  }

  // пишем данные в файл
  for (int i = 0; i < 10; ++i)
  {
    DWORD dwBytesWrite;

    if (!WriteFile(
        hFile,              // дескриптор файла
        &i,                 // адрес буфера, откуда идет запись
        sizeof(i),          // количество записываемых байтов
        &dwBytesWrite,      // количество записанных байтов
```

```
        (LPOVERLAPPED)NULL   // запись синхронная
      ))
    {
      cerr << "Write file failed." << endl
        << "The last error code: " << GetLastError() << endl;
      CloseHandle(hFile);
      cout << "Press any key to finish.";
      cin.get();
      return 0;
    }
  }
  // закрываем дескриптор файла
  CloseHandle(hFile);

  cout << "The file is created and written." << endl;

  return 0;
}
```

24.5. Освобождение буферов файла

Часто несколько приложений имеют совместный доступ к одному и тому же файлу. При этом может потребоваться, чтобы приложение, которое читает данные из файла, имело доступ к последней версии этого файла. Для этого необходимо, чтобы приложение, которое изменяет содержимое файла, фиксировало изменение файла после обработки нужных записей. Так как не исключена возможность того, что последние обработанные записи хранятся в буфере файла, то в этом случае необходимо освободить буфер от записей. Выполнить эту операцию можно посредством функции `FlushFileBuffers`, которая имеет следующий прототип:

```
BOOL  FlushFileBuffers(
  HANDLE  hFile    // дескриптор файла
);
```

В случае удачного завершения эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`.

В листинге 24.3 приведена программа, в которой функция `FlushFileBuffers` используется для сброса данных из буферов в файл после записи половины файла.

Листинг 24.3. Освобождение буферов файла

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE hFile;

  // создаем файл для записи данных
  hFile = CreateFile(
    "C:\\demo_file.dat",      // имя файла
    GENERIC_WRITE,            // запись в файл
    FILE_SHARE_READ,          // разделяемое чтение файла
    NULL,                     // защиты нет
    CREATE_ALWAYS,            // создаем новый файл
    FILE_ATTRIBUTE_NORMAL,    // обычный файл
    NULL                      // шаблона нет
  );
  // проверяем на успешное создание
  if (hFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();
    return 0;
  }

  // пишем данные в файл
  for (int i = 0; i < 10; ++i)
  {
    DWORD dwBytesWrite;

    if (!WriteFile(
        hFile,                // дескриптор файла
        &i,                   // адрес буфера, откуда идет запись
        sizeof(i),            // количество записываемых байтов
        &dwBytesWrite,        // количество записанных байтов
```

```
        (LPOVERLAPPED)NULL   // запись синхронная
      ))
    {
      cerr << "Write file failed." << endl
        << "The last error code: " << GetLastError() << endl;
      CloseHandle(hFile);
      cout << "Press any key to finish.";
      cin.get();
      return 0;
    }
    // если достигли середины файла, то освобождаем буфер
    if(i == 5)
    {
      if(!FlushFileBuffers(hFile))
      {
        cerr << "Flush file buffers failed." << endl
          << "The last error code: " << GetLastError() << endl;
        CloseHandle(hFile);
        cout << "Press any key to finish.";
        cin.get();
        return 0;
      }
      // теперь можно просмотреть содержимое файла
      cout << "A half of the file is written." << endl
        << "Press any key to continue.";
      cin.get();
    }
  }
  // закрываем дескриптор файла
  CloseHandle(hFile);

  cout << "The file is created and written." << endl;

  return 0;
}
```

В заключение этого раздела отметим, что можно отменить режим буферизации файла, установив флаг `FILE_FLAG_NO_BUFFERING` в параметре `dwFlagsAndAttributes` функции `CreateFile`. Однако в этом случае длина

записываемых или считываемых данных из файла должна быть кратна размеру сектора. Например, в операционных системах Windows длина сектора равна 512 байт.

24.6. Чтение данных из файла

Для чтения данных из файла служит функция `ReadFile`, которая может использоваться как для синхронного, так и асинхронного чтения данных. В этом разделе будет рассмотрено только синхронное чтение данных из файла. В этом случае данные читаются из файла последовательно байт за байтом, и указатель файла передвигается по мере чтения данных на новую позицию. Асинхронный ввод-вывод данных будет рассмотрен в *гл. 27*.

Теперь более подробно рассмотрим функцию `ReadFile`, которая имеет следующий прототип:

```
BOOL  ReadFile(
  HANDLE   hFile,                 // дескриптор файла
  LPVOID   lpBuffer,              // указатель на буфер данных
  DWORD    nNumberOfBytesToRead,  // количество читаемых байтов
  LPDWORD  lpNumberOfBytesRead,   // количество прочитанных байтов
  LPOVERLAPPED  lpOverlapped   // используется при асинхронной записи
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`.

Параметр `hFile` должен содержать дескриптор файла, причем файл должен быть открыт в режиме чтения.

Параметр `lpBuffer` должен указывать на область памяти, из которой будут читаться данные.

Параметр `nNumberOfBytesToRead` должен содержать количество байт, которые предполагается читать из файла посредством вызова функции `ReadFile`.

Параметр `lpNumberOfBytesRead` должен содержать адрес памяти, в которую функция `ReadFile` поместит количество фактически прочитанных из файла байтов. Как и в случае функции `WriteFile`, при работе на платформе Windows 98 этот параметр должен иметь значение, отличное от `NULL`. При выполнении функции `ReadFile` операционная система записывает по этому адресу 0, прежде чем выполнить чтение данных из файла.

Так как в этом разделе не будет рассматриваться асинхронное чтение данных из файла, то параметр `lpOverlapped` будет устанавливаться в `NULL`. Подробно асинхронный ввод-вывод данных будет рассмотрен в *гл. 27*.

В листинге 24.4 приведена программа, которая читает данные из файла, созданного программой из листинга 24.2, и выводит эти данные на консоль.

Листинг 24.4. Открытие файла и чтение из него данных

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;

  // открываем файл для чтения
  hFile = CreateFile(
    "C:\\demo_file.dat",      // имя файла
    GENERIC_READ,             // чтение из файла
    0,                        // монопольный доступ к файлу
    NULL,                     // защиты нет
    OPEN_EXISTING,            // открываем существующий файл
    FILE_ATTRIBUTE_NORMAL,    // обычный файл
    NULL        // шаблона нет
  );
  // проверяем на успешное открытие
  if (hFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();
    return 0;
  }

  // читаем данные из файла
  for (;;)
  {
    DWORD  dwBytesRead;
    int    n;

    // читаем одну запись
    if (!ReadFile(
        hFile,                // дескриптор файла
        &n,                   // адрес буфера, куда читаем данные
        sizeof(n),            // количество читаемых байтов
```

```
        &dwBytesRead,        // количество прочитанных байтов
        (LPOVERLAPPED)NULL   // чтение синхронное
      ))
    {
      cerr << "Read file failed." << endl
        << "The last error code: " << GetLastError() << endl;
      CloseHandle(hFile);
      cout << "Press any key to finish.";
      cin.get();
      return 0;
    }
    // проверяем на конец файла
    if (dwBytesRead == 0)
      // если да, то выходим из цикла
      break;
    else
      // иначе выводим запись на консоль
      cout << n << ' ';
  }

  cout << endl;

  // закрываем дескриптор файла
  CloseHandle(hFile);

  cout << "The file is opened and read." << endl;

  return 0;
}
```

В связи с этой программой отметим обработку конца файла. При достижении конца файла и запросе на чтение записи функция `ReadFile` возвращает ненулевое значение и при этом устанавливает значение количества прочитанных байтов в 0.

24.7. Копирование файла

Для копирования файлов используется функция `CopyFile`, которая имеет следующий прототип:

```
BOOL  CopyFile(
  LPCTSTR  lpExistingFileName,   // имя существующего файла
```

```
    LPCTSTR  lpNewFileName,        // имя нового файла
    BOOL     bFailIfExists   // действия в случае существования файла
  );
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. При этом отметим, что функция `CopyFile` копирует для нового файла также и атрибуты доступа старого файла, но атрибуты безопасности не копируются.

Параметры этой функции имеют следующее назначение.

Параметр `lpExistingFileName` должен указывать на строку, содержащую имя копируемого файла.

Параметр `lpNewFileName` должен указывать на строку с именем файла, в который будет копироваться существующий файл. При этом отметим, что новый файл создается самой функцией `CopyFile`.

Параметр `bFailIfExists` определяет действия, которые нужно осуществить в случае, если файл, в который выполняется копирование, уже существует. Если значение этого параметра равно `FALSE`, то функция перезаписывает существующий файл. Если же значение этого параметра равно `TRUE`, то выполнение функции в этом случае заканчивается неудачей.

В листинге 24.5 приведена программа, которая выполняет копирование файла, используя функцию `CopyFile`.

Листинг 24.5. Копирование файла

```
#include <windows.h>
#include <iostream.h>

int main()
{
  // копируем файл
  if(!CopyFile("C:\\demo_file.dat", "C:\\new_file.dat", FALSE))
  {
    cerr << "Copy file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();
    return 0;
  }

  cout << "The file is copied." << endl;

  return 0;
}
```

В операционных системах семейства Windows NT (4.0\2000\XP) для копирования файлов может использоваться функция `CopyFileEx`, которая предоставляет более широкие возможности по копированию файлов.

24.8. Перемещение файла

Теперь разберем функцию `MoveFile`, которая служит для перемещения файлов. Перемещение файла отличается от копирования файла только тем, что старый файл после его перемещения удаляется.

Функция `MoveFile` имеет следующий прототип:

```
BOOL  MoveFile(
  LPCTSTR  lpExistingFileName,   // имя существующего файла
  LPCTSTR  lpNewFileName         // имя нового файла
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Отметим, что функция `MoveFile` сохраняет все атрибуты перемещаемого файла. Параметры этой функции имеют следующее назначение.

Параметр `lpExistingFileName` должен указывать на строку, содержащую имя перемещаемого файла.

Параметр `lpNewFileName` должен указывать на строку с именем файла, в который будет перемещаться существующий файл. При этом отметим, что новый файл создается самой функцией `MoveFile`.

В листинге 24.6 приведена программа, которая выполняет перемещение файла, используя функцию `MoveFile`.

Листинг 24.6. Перемещение файла

```
#include <windows.h>
#include <iostream.h>

int main()
{
  // перемещаем файл
  if(!MoveFile("C:\\demo_file.dat", "C:\\new_file.dat"))
  {
    cerr << "Move file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
```

```
    cin.get();
    return 0;
  }

  cout << "The file is moved." << endl;

  return 0;
}
```

В операционных системах семейства Windows NT (4.0\2000\XP) для перемещения файлов может также использоваться функция `MoveFileEx`, в которой можно задавать режимы перемещения файлов.

24.9. Замещение файла

В Windows 2000 и Windows XP определена функция `ReplaceFile`, которая предназначена для замещения файлов. Существенное отличие этой функции от функций `CopyFile` и `MoveFile` состоит в том, что она копирует в замещаемый файл не только атрибуты доступа, но также и атрибуты безопасности файла-заместителя. Отметим также, что функция `ReplaceFile` работает только с файлами, которые находятся на одном томе (логическом диске).

Функция `ReplaceFile` имеет следующий прототип:

```
BOOL  ReplaceFile(
  LPCTSTR  lpReplacedFileName,     // имя замещаемого файла
  LPCTSTR  lpReplacementFileName,  // имя файла-заместителя
  LPCTSTR  lpBackupFileName,   // имя резервной копии файла-заместителя
  DWORD    dwReplaceFlags,     // опции замещения
  LPVOID   lpExclude,          // не используется
  LPVOID   lpReserved          // не используется
);
```

Функция замещения файлов в случае успешного завершения возвращает ненулевое значение, а в случае неудачи — `FALSE`. Кратко опишем назначение параметров этой функции.

Параметр `lpReplacedFileName` должен указывать на строку, которая содержит имя замещаемого файла. При этом отметим, что функция `ReplaceFile` может как создавать новый файл, так и использовать уже существующий.

Параметр `lpReplacementFileName` должен указывать на строку, которая содержит имя файла-заместителя. Этот файл заменит файл, на который указывает параметр `lpReplacedFileName`. При этом отметим, что сам файл-заместитель удаляется с диска.

В параметре lpBackupFileName устанавливается адрес строки, которая содержит имя файла, который содержит резервную копию файла-заместителя. Если этот параметр установлен в NULL, то резервная копия файла-заместителя не создается.

В параметре dwReplaceFlags устанавливаются флаги, которые указывают режимы замещения файла. Можно установить любую комбинацию следующих флагов:

- REPLACEFILE_WRITE_THROUGH — освободить буферы перед выходом из функции;
- REPLACEFILE_IGNORE_MERGE_ERRORS — игнорировать ошибки при копировании данных.

Параметры lpExclude и lpReserved не используются. Поэтому они должны быть установлены в NULL.

В листинге 24.7 приведена программа, в которой показан пример использования функция ReplaceFile для замещения файла.

Листинг 24.7. Замещение файла

```
#include <windows.h>
#include <iostream.h>

int main()
{
  // перемещаем файл
  if(!ReplaceFile(
    "C:\\new_file.dat",         // имя замещаемого файла
    "C:\\demo_file.dat",        // имя файла-заместителя
    "C:\\back_file.dat",        // имя резервного файла
    REPLACEFILE_WRITE_THROUGH, // освободить буферы
    NULL, NULL                  // не используются
    ))
  {
    cerr << "Replace file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();
    return 0;
  }
```

```
  cout << "The file is replaced." << endl;

  return 0;
}
```

24.10. Работа с указателем позиции файла

Прежде чем разбираться с функцией для работы с указателем позиции файла, рассмотрим формат самого указателя позиции файла. Указатель позиции файла состоит из двух значений типа DWORD, которые будут называться старшая и младшая часть указателя позиции файла соответственно. Следовательно, указатель позиции имеет длину в 64 бита. Если длина файла не превышает двух гигабайт без двух байт, т. е. $2^{31} - 2$ байта, то в указателе позиции используется только его младшая часть, которая рассматривается как целое число со знаком, но при этом старшая часть указателя позиции должна быть установлена в NULL.

Для работы с указателем позиции файла служит функция SetFilePointer, которая имеет следующий прототип:

```
DWORD  SetFilePointer(
  HANDLE  hFile,           // дескриптор файла
  LONG    lDistanceToMove, // младшая часть сдвига указателя в байтах
  PLONG   lpDistanceToMoveHigh,  // указатель на старшую часть сдвига
                                 // указателя в байтах
  DWORD   dwMoveMethod           // начальная точка сдвига
);
```

В случае удачного завершения эта функция возвращает младшую часть новой позиции указателя файла, а по адресу, заданному параметром lpDistanceToMoveHigh, записывает старшую часть новой позиции указателя файла. Если функция устанавливает старшую часть указателя позиции в NULL, то младшая часть указателя позиции представлена положительным целым числом. В случае неудачного завершения функция SetFilePointer возвращает значение -1 и при этом устанавливает значение параметра lpDistanceToMoveHigh в NULL. Если же значение этого параметра не установлено в NULL, то возвращаемое значение -1 может быть и действительной младшей частью указателя позиции. В этом случае нужно проверить код последней ошибки, который возвращает функция GetLastError. Если этот код равен NO_ERROR, то ошибки нет, а в противном случае выполнение функции SetFilePointer завершилось неудачей. Теперь кратко опишем назначение параметров функции SetFilePointer.

Параметр hFile должен содержать дескриптор файла, причем сам файл должен быть открыт в режиме чтения или записи.

Параметр `lDistanceToMove` должен содержать младшую часть сдвига для указателя позиции файла. Если значение параметра `lpDistanceToMoveHigh` установлено в `NULL`, то значение этого параметра рассматривается как целое число со знаком. В случае положительного числа функция выполняет сдвиг указателя вперед на заданное количество байт, а в случае отрицательного числа выполняется сдвиг назад.

Параметр `lpDistanceToMoveHigh` должен содержать адрес старшей части сдвига для указателя позиции файла. Старшая и младшая части указателя позиции рассматриваются как целое число со знаком. Если значение этого параметра равно `NULL`, то сдвиг задается только младшей частью. Отметим, что в операционной системе Windows 98 значение этого параметра может быть равно только одному из следующих значений: `NULL`, `0` и `-1`.

Параметр `dwMoveMethod` задает начальную точку, от которой выполняется сдвиг указателя позиции. Этот параметр может принимать только одно из следующих значений:

- `FILE_BEGIN` — сдвиг от начала файла;
- `FILE_CURRENT` — сдвиг от текущей позиции файла;
- `FILE_END` — сдвиг от конца файла.

В листинге 24.8 приведена программа, которая читает запись файла, предварительно установив на эту запись указатель. Кстати, такое чтение записей файла называется прямым доступом к файлу. В общем случае прямой доступ к файлу подразумевает чтение записи с заданным значением ключа, который определяется содержимым одного или нескольких полей записи. Правда, в этом случае нужно знать зависимость указателя позиции файла от значения ключа записи.

Листинг 24.8. Установка указателя позиции файла при помощи функции `SetFilePointer`

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;     // дескриптор файла
  long    n;         // для номера записи
  long    p;         // для указателя позиции
  DWORD dwBytesRead; // количество прочитанных байт
  int     m;         // прочитанное число

  // открываем файл для чтения
```

```
hFile = CreateFile(
  "C:\\demo_file.dat",    // имя файла
  GENERIC_READ,           // чтение из файла
  0,                      // монопольный доступ к файлу
  NULL,                   // защиты нет
  OPEN_EXISTING,          // открываем существующий файл
  FILE_ATTRIBUTE_NORMAL, // обычный файл
  NULL                    // шаблона нет
);
// проверяем на успешное открытие
if (hFile == INVALID_HANDLE_VALUE)
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish.";
  cin.get();
  return 0;
}

// вводим номер нужной записи
cout << "Input a number from 0 to 9: ";
cin >> n;
// сдвигаем указатель позиции файла
p = SetFilePointer(hFile, n * sizeof(int), NULL, FILE_BEGIN);
if(p == -1)
{
  cerr << "Set file pointer failed." << endl
    << "The last error code: " << GetLastError() << endl;
  CloseHandle(hFile);
  cout << "Press any key to finish.";
  cin.get();
  return 0;
}
// выводим на консоль значение указателя позиции файла
cout << "File pointer: " << p << endl;
// читаем данные из файла
if (!ReadFile(
    hFile,              // дескриптор файла
    &m,                 // адрес буфера, куда читаем данные
    sizeof(m),          // количество читаемых байтов
```

```
      &dwBytesRead,      // количество прочитанных байтов
      (LPOVERLAPPED)NULL // чтение синхронное
    ))
  {
    cerr << "Read file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hFile);
    cout << "Press any key to finish.";
    cin.get();
    return 0;
  }
  // выводим прочитанное число на консоль
  cout << "The read number: " << m << endl;
  // закрываем дескриптор файла
  CloseHandle(hFile);

  return 0;
}
```

При помощи функции `SetFilePointer` можно также определить текущее состояние указателя позиции файла. Для этого нужно просто сдвинуть указатель файла от текущей позиции на нулевое количество байт. В результате функция `SetFilePointer` вернет текущее состояние указателя позиции файла.

Начиная с операционной системы Windows 2000, для работы с указателем позиции файла можно использовать функцию `SetFilePointerEx`, которая более проста в использовании и имеет следующий прототип:

```
BOOL  SetFilePointerEx(
  HANDLE  hFile,                    // дескриптор файла
  LARGE_INTEGER  liDistanceToMove,  // сдвиг в байтах
  PLARGE_INTEGER  lpNewFilePointer, // новый указатель позиции файла
  DWORD   dwMoveMethod              // начальная точка сдвига
);
```

В случае удачного завершения эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. Первый и последний параметры этой функции имеют такое же назначение, как и в функции `SetFilePointer`. Поэтому кратко опишем только оставшиеся параметры.

Параметр `liDistanceToMove` задает сдвиг указателя позиции файла, который рассматривается как целое число со знаком. В случае положительного числа указатель позиции сдвигается вперед, а в случае отрицательного — назад.

Параметр `lpNewFilePointer` должен указывать на объединение типа `LARGE_INTEGER`, в которое функция `SetFilePointerEx` вернет новое значение индикатора позиции файла. Объединение типа `LARGE_INTEGER` имеет следующий формат:

```
typedef union _LARGE_INTEGER
  {
    struct
    {
      DWORD LowPart;        // младшая часть
      LONG   HighPart;      // старшая часть
    };
    LONGLONG QuadPart;      // все части
  } LARGE_INTEGER, *PLARGE_INTEGER;
```

Если значение параметра `lpNewFilePointer` равно `NULL`, то новое значение индикатора позиции не будет возвращаться.

В листинге 24.9 приведена программа, которая для установки индикатора позиции использует функцию `SetFilePointerEx`, а затем читает из файла запись, на которую установлен индикатор позиции.

Листинг 24.9. Установка указателя позиции файла при помощи функции `SetFilePointerEx`

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;     // дескриптор файла
  int     n;         // для номера записи
  LARGE_INTEGER  p;  // для указателя позиции
  DWORD dwBytesRead; // количество прочитанных байтов
  int     m;         // прочитанное число

  // открываем файл для чтения
  hFile = CreateFile(
    "C:\\demo_file.dat",    // имя файла
    GENERIC_READ,           // чтение из файла
    0,                      // монопольный доступ к файлу
    NULL,                   // защиты нет
```

```
  OPEN_EXISTING,         // открываем существующий файл
  FILE_ATTRIBUTE_NORMAL, // обычный файл
  NULL                   // шаблона нет
);
// проверяем на успешное открытие
if (hFile == INVALID_HANDLE_VALUE)
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish.";
  cin.get();
  return 0;
}

// вводим номер нужной записи
cout << "Input a number from 0 to 9: ";
cin >> n;
// сдвигаем указатель позиции файла
if(!SetFilePointerEx(hFile, n * sizeof(int), &p, FILE_BEGIN))
{
  cerr << "Set file pointer failed." << endl
    << "The last error code: " << GetLastError() << endl;
  CloseHandle(hFile);
  cout << "Press any key to finish.";
  cin.get();
  return 0;
}
// выводим на консоль значение указателя позиции файла
cout << "File pointer: " << p << endl;
// читаем данные из файла
if (!ReadFile(
    hFile,             // дескриптор файла
    &m,                // адрес буфера, куда читаем данные
    sizeof(m),         // количество читаемых байтов
    &dwBytesRead,      // количество прочитанных байтов
    (LPOVERLAPPED)NULL // чтение синхронное
  ))
{
  cerr << "Read file failed." << endl
```

```
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hFile);
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  // выводим прочитанное число на консоль
  cout << "The read number: " << m << endl;
  // закрываем дескриптор файла
  CloseHandle(hFile);

  return 0;
}
```

24.11. Определение и изменение атрибутов файла

Узнать атрибуты файла можно при помощи функции `GetFileAttributes`, которая имеет следующий прототип:

```
DWORD  GetFileAttributes(
  LPCTSTR  lpFileName    // имя файла
);
```

В случае успешного завершения эта функция возвращает атрибуты файла, а в случае неудачи — значение `-1`. Единственный параметр этой функции должен содержать имя файла, а в возвращаемом значении устанавливаются атрибуты файла. Эти атрибуты можно проверить, используя следующие флаги:

- `FILE_ATTRIBUTE_ARCHIVE` — архивный файл;
- `FILE_ATTRIBUTE_COMPRESSED` — сжатый файл;
- `FILE_ATTRIBUTE_DIRECTORY` — файл является каталогом;
- `FILE_ATTRIBUTE_ENCRYPTED` — шифрованный файл;
- `FILE_ATTRIBUTE_HIDDEN` — скрытый файл;
- `FILE_ATTRIBUTE_NORMAL` — нормальный файл;
- `FILE_ATTRIBUTE_NOT_CONTENT_INDEXED` — файл не индексируется;
- `FILE_ATTRIBUTE_OFFLINE` — файл во внешней памяти;
- `FILE_ATTRIBUTE_READONLY` — файл только для чтения;

- FILE_ATTRIBUTE_REPARSE_POINT — файл требует интерпретации;
- FILE_ATTRIBUTE_SPARSE_FILE — разреженный файл;
- FILE_ATTRIBUTE_SYSTEM — системный файл;
- FILE_ATTRIBUTE_TEMPORARY — временный файл.

Более подробно атрибуты файла описаны в *разд. 24.2*. Другие атрибуты файла, которые не вошли в вышеприведенный список, можно узнать, используя функцию GetFileAttributesEx.

Изменить атрибуты файла можно при помощи функции SetFileAttributes, которая имеет следующий прототип:

```
BOOL  SetFileAttributes(
  LPCTSTR  lpFileName,          // имя файла
  DWORD    dwFileAttributes     // атрибуты файла
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — FALSE. Параметр lpFileName должен содержать имя файла, а в параметре dwFileAttributes можно установить следующие атрибуты файла:

- FILE_ATTRIBUTE_ARCHIVE — архивный файл;
- FILE_ATTRIBUTE_HIDDEN — скрытый файл;
- FILE_ATTRIBUTE_NORMAL — нормальный файл;
- FILE_ATTRIBUTE_NOT_CONTENT_INDEXED — файл не индексируется;
- FILE_ATTRIBUTE_OFFLINE — файл во внешней памяти;
- FILE_ATTRIBUTE_READONLY — файл только для чтения;
- FILE_ATTRIBUTE_SYSTEM — системный файл;
- FILE_ATTRIBUTE_TEMPORARY — временный файл.

При этом отметим, что если устанавливается флаг FILE_ATTRIBUTE_NORMAL, то он должен быть один, т. к. все остальные атрибуты, аннулируют этот атрибут.

В листинге 24.10 приведена программа, которая читает и изменяет атрибуты файла.

Листинг 24.10. Чтение и изменение атрибутов файла

```
#include <windows.h>
#include <iostream.h>

int main()
{
```

```
DWORD  file_attr;

// читаем атрибуты файла
file_attr = GetFileAttributes("C:\\demo_file.dat");
if(file_attr == -1)
{
  cerr << "Get file attributes failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish.";
  cin.get();
  return 0;
}
// проверяем, является ли файл нормальным
if(file_attr == FILE_ATTRIBUTE_NORMAL)
  cout << "This is a normal file." << endl;
else
{
  cout << "This is a not normal file." << endl;
  return 0;
}
// устанавливаем атрибут скрытого файла
if(!SetFileAttributes("C:\\demo_file.dat", FILE_ATTRIBUTE_HIDDEN))
{
  cerr << "Set file attributes failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
// Теперь можно проверить, что файл скрылся
cout << "Now the file is hidden." << endl
  << "Press any key to continue.";
cin.get();
// Обратно делаем файл обычным
if(!SetFileAttributes("C:\\demo_file.dat", FILE_ATTRIBUTE_NORMAL))
{
  cerr << "Set file attributes failed." << endl
    << "The last error code: " << GetLastError() << endl;
```

```
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  cout << "Now the file is again normal." << endl;

  return 0;
}
```

24.12. Определение и изменение размеров файла

Прежде чем описывать функции для работы с размером файла, скажем, что размер файла является целым числом и для его хранения требуются два значения типа DWORD или, другими словами, два двойных слова. Первое из этих двойных слов содержит старшую часть размера файла, а второе — младшую. Если размер файла входит только в младшую часть, то значение старшей устанавливается в NULL.

Определить размер файла можно при помощи функции GetFileSize, которая имеет следующий прототип:

```
DWORD  GetFileSize(
  HANDLE   hFile,          // дескриптор файла
  LPDWORD  lpFileSizeHigh  // указатель на старшую часть размера файла
);
```

В случае успешного завершения эта функция возвращает младшую часть размера файла, а по адресу, указанному в параметре lpFileSizeHigh, записывает старшую часть размера файла. В случае неудачи функция GetFileSize возвращает значение -1, если значение адреса, заданного параметром lpFileSizeHigh, установлено в NULL. Если же значение этого адреса не равно NULL и функция закончилась неудачей, то она возвращает значение -1 и функция кода последней ошибки GetLastError возвратит значение, отличное от NO_ERROR.

Отметим, что для правильной работы функции GetFileSize необходимо, чтобы файл был открыт в режиме чтения или записи.

В листинге 24.11 приведена программа, которая определяет размер файла, используя функцию GetFileSize.

Листинг 24.11. Определение размера файла при помощи функции `GetFileSize`

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;
  DWORD  dwFileSize;   // младшая часть размера файла

  // открываем файл для чтения
  hFile = CreateFile(
    "C:\\demo_file.dat",   // имя файла
    GENERIC_READ,          // чтение из файла
    0,                     // монопольный доступ к файлу
    NULL,                  // защиты нет
    OPEN_EXISTING,         // открываем существующий файл
    FILE_ATTRIBUTE_NORMAL, // обычный файл
    NULL                   // шаблона нет
  );
  // проверяем на успешное открытие
  if (hFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }

  // определяем размер файла
  dwFileSize = GetFileSize(hFile, NULL);
  if (dwFileSize == -1)
  {
    cerr << "Get file size failed." << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hFile);
```

```
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  // выводим размер файла
  cout << "File size: " << dwFileSize << endl;
  // закрываем дескриптор файла
  CloseHandle(hFile);

  return 0;
}
```

Определить размер файла можно также при помощи функции `GetFileSizeEx`, которая поддерживается только операционными системами Windows 2000 и Windows XP. Эта функция имеет следующий прототип:

```
BOOL  GetFileSizeEx(
  HANDLE  hFile,                // дескриптор файла
  PLARGE_INTEGER  lpFileSize    // размер файла
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`.

Параметр `hFile` этой функции должен содержать дескриптор файла, а параметр `lpFileSize` указывает на объединение типа `LARGE_INTEGER`, в которое функция `GetFileSizeEx` записывает размер файла. Формат объединения типа `LARGE_INTEGER` приведен в *разд. 24.10*.

В листинге 24.12 приведена программа, которая определяет размер файла, используя функцию `GetFileSizeEx`.

Листинг 24.12. Определение размера файла при помощи функции `GetFileSizeEx`

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;
  LARGE_INTEGERD  liFileSize;  // размер файла

  // открываем файл для чтения
```

```
  hFile = CreateFile(
    "C:\\demo_file.dat",    // имя файла
    GENERIC_READ,           // чтение из файла
    0,                      // монопольный доступ к файлу
    NULL,                   // защиты нет
    OPEN_EXISTING,          // открываем существующий файл
    FILE_ATTRIBUTE_NORMAL, // обычный файл
    NULL                    // шаблона нет
  );
  // проверяем на успешное открытие
  if (hFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }

  // определяем размер файла
  if (!GetFileSizeEx(hFile, &liFileSize))
    cerr << "Get file size failed." << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hFile);
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  // выводим размер файла
  cout << "File size: " << liFileSize.LowPart << endl;
  // закрываем дескриптор файла
  CloseHandle(hFile);

  return 0;
}
```

Изменить размер файла можно при помощи функции SetEndOfFile, которая имеет следующий прототип:

```
BOOL  SetEndOfFile(
  HANDLE  hFile    // дескриптор файла
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — FALSE. Единственным параметром функции является дескриптор файла, размер которого изменяется. При этом отметим, что файл должен быть открыт в режиме записи.

Функция SetEndOfFile работает следующим образом. Она передвигает маркер конца файла EOF на позицию, которую содержит указатель позиции файла. Если указатель позиции файла указывает на запись, которая не является последней, то все записи, которые находятся за текущей записью, отбрасываются. Таким образом, в этом случае содержимое файла урезается. Если же указатель позиции файла указывает за пределы файла, то объем файла расширяется за счет добавления новых кластеров к нему. При этом содержимое добавленных кластеров не определено. В листинге 24.13 приведен пример программы, которая уменьшает размер файла в два раза.

Листинг 24.13. Изменение размера файла

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;       // дескриптор файла
  DWORD   dwFileSize;  // размер файла
  long    p;           // указатель позиции

  // открываем файл для чтения
  hFile = CreateFile(
    "C:\\demo_file.dat",   // имя файла
    GENERIC_WRITE,         // запись в файл
    0,                     // монопольный доступ к файлу
    NULL,                  // защиты нет
    OPEN_EXISTING,         // открываем существующий файл
    FILE_ATTRIBUTE_NORMAL, // обычный файл
    NULL                   // шаблона нет
  );
```

```
// проверяем на успешное открытие
if (hFile == INVALID_HANDLE_VALUE)
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
// определяем размер файла
dwFileSize = GetFileSize(hFile, NULL);
if (dwFileSize == -1)
{
  cerr << "Get file size failed." << endl
    << "The last error code: " << GetLastError() << endl;
  CloseHandle(hFile);
  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
// выводим на консоль размер файла
cout << "Old file size: " << dwFileSize << endl;
// уменьшаем размер файла вдвое
dwFileSize /= 2;
// сдвигаем указатель позиции файла
p = SetFilePointer(hFile, dwFileSize, NULL, FILE_BEGIN);
if(p == -1)
{
  cerr << "Set file pointer failed." << endl
    << "The last error code: " << GetLastError() << endl;
  CloseHandle(hFile);
  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
// устанавливаем новый размер файла
```

```
  if (!SetEndOfFile(hFile))
  {
    cerr << "Set end of file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hFile);
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  // определяем новый размер файла
  dwFileSize = GetFileSize(hFile, NULL);
  if (dwFileSize == -1)
  {
    cerr << "Get file size failed." << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hFile);
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  // выводим на консоль размер файла
  cout << "New file size: " << dwFileSize << endl;
  // закрываем дескриптор файла
  CloseHandle(hFile);

  return 0;
}
```

24.13. Блокирование файла

Часто несколько приложений имеют совместный доступ к одному и тому же файлу. При этом может потребоваться, чтобы приложение, которое изменяет данные в файле, имело к этому файлу монопольный доступ. Для этого необходимо блокировать весь файл или только его часть для доступа другим приложениям. Для этих целей используется функция `LockFile`, которая имеет следующий прототип:

```
  BOOL  LockFile(
    HANDLE  hFile,                    // дескриптор файла
```

```
    DWORD   dwFileOffsetLow,          // младшая часть смещения
    DWORD   dwFileOffsetHigh,         // старшая часть смещения
    DWORD   nNumberOfBytesToLockLow,  // младшая часть количества байтов
    DWORD   nNumberOfBytesTiLockHigh  // старшая часть количества байтов
  );
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. Кратко опишем назначение параметров этой функции.

Параметр `hFile` должен содержать дескриптор файла. Причем этот файл должен быть открыт в режиме записи или чтения.

В параметрах `dwFileOffsetLow` и `dwFileOffsetHigh` должны быть установлены соответственно младшая и старшая части смещения от начала файла в байтах. Для операционной системы Windows 98 значение параметра `dwFileOffsetHigh` должно быть установлено в `0`.

В параметрах `nNumberOfBytesToLockLow` и `nNumberOfBytesTiLockHigh` должны быть установлены соответственно старшая и младшая части длины области файла, которая блокируется для монопольного доступа приложением.

Для отмены блокировки области файла используется функция `UnlockFile`, которая имеет следующий прототип:

```
  BOOL  UnlockFile(
    HANDLE  hFile,                    // дескриптор файла
    DWORD   dwFileOffsetLow,          // младшая часть смещения
    DWORD   dwFileOffsetHigh,         // старшая часть смещения
    DWORD   nNumberOfBytesToLockLow,  // младшая часть количества байтов
    DWORD   nNumberOfBytesTiLockHigh  // старшая часть количества байтов
  );
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Все параметры этой функции аналогичны параметрам функции `LockFile`.

В листинге 24.14 приведена программа, которая сначала блокирует доступ к файлу, а затем разблокирует его.

Листинг 24.14. Блокировка и разблокировка файла

```
#include <windows.h>
#include <iostream.h>

int main()
{
```

```
HANDLE  hFile;
DWORD  dwFileSize;

// открываем файл для записи
hFile = CreateFile(
  "C:\\demo_file.dat",    // имя файла
  GENERIC_WRITE,          // запись в файл
  0,                      // монопольный доступ к файлу
  NULL,                   // защиты нет
  OPEN_EXISTING,          // открываем существующий файл
  FILE_ATTRIBUTE_NORMAL, // обычный файл
  NULL                    // шаблона нет
);
// проверяем на успешное открытие
if (hFile == INVALID_HANDLE_VALUE)
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
// определяем размер файла
dwFileSize = GetFileSize(hFile, NULL);
if (dwFileSize == -1)
{
  cerr << "Get file size failed." << endl
    << "The last error code: " << GetLastError() << endl;
  CloseHandle(hFile);
  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
// блокируем файл
if (!LockFile(hFile, 0, 0, dwFileSize, 0))
{
```

```
    cerr << "Lock file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hFile);
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  cout << "Now the file is locked." << endl
    << "Press any key to continue." << endl;
  cin.get();
  // разблокируем файл
  if (!UnlockFile(hFile, 0, 0, dwFileSize, 0))
  {
    cerr << "Unlock file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hFile);
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  cout << "Now the file is unlocked." << endl
    << "Press any key to continue." << endl;
  cin.get();
  // закрываем дескриптор файла
  CloseHandle(hFile);

  return 0;
}
```

В заключение этого раздела сделаем несколько замечаний относительно использования функций `LockFile` и `UnlockFile`. Прежде всего, отметим, что если приложение выдает запрос на блокирование области файла, которая уже заблокирована самим приложением, то выполнение функции `LockFile` завершится неудачей. Кроме того, блокирование и разблокирование файлов должно совпадать по областям.

24.14. Получение информации о файле

Чтобы получить информацию о файле, можно использовать функцию GetFileInformationByHandle, которая имеет следующий прототип:

```
BOOL  GetFileInformationByHandle(
  HANDLE  hFile,       // дескриптор файла
  // указатель на информацию
  LPBY_HANDLE_FILE_INFORMATION  lpFileInformation
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — FALSE.

В параметре hFile этой функции должен быть установлен дескриптор файла, информацию о котором требуется получить. Отметим, что этот файл может быть открыт в любом режиме доступа.

Параметр lpFileInformation должен указывать на структуру типа BY_HANDLE_FILE_INFORMATION, в которую функция запишет информацию о файле. Эта структура имеет следующий формат:

```
typedef struct _BY_HANDLE_FILE_INFORMATION {
  DWORD  dwFileAttributes;       // атрибуты файла
  FILETIME  ftCreationTime;      // время создания файла
  FILETIME  ftLastAccessTime;    // время последнего доступа к файлу
  FILETIME  ftLastWriteTime;     // время последней записи в файл
  DWORD  dwVolumeSerialNumber;   // серийный номер тома
  DWORD  nFileSizeHigh;          // старшая часть размера файла
  DWORD  nFileSizeLow;           // младшая часть размера файла
  DWORD  nNumberOfLinks;         // количество ссылок на файл
  DWORD  nFileIndexHigh;         // старшая часть индекса файла
  DWORD  nFileIndexLow;          // младшая часть индекса файла
} BY_HANDLE_FILE_INFORMATION,  *LPBY_HANDLE_FILE_INFORMATION;
```

В листинге 24.15 приведена программа, которая получает информацию о файле и распечатывает ее.

Листинг 24.15. Получение информации о файле

```
#include <windows.h>
#include <iostream.h>

int main()
{
```

```
HANDLE  hFile;
BY_HANDLE_FILE_INFORMATION  bhfi;  // информация о файле

// открываем файл для чтения
hFile = CreateFile(
  "C:\\demo_file.dat",    // имя файла
  0,                      // получение информации о файле
  0,                      // монопольный доступ к файлу
  NULL,                   // защиты нет
  OPEN_EXISTING,          // открываем существующий файл
  FILE_ATTRIBUTE_NORMAL, // обычный файл
  NULL                    // шаблона нет
);
// проверяем на успешное открытие
if (hFile == INVALID_HANDLE_VALUE)
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
// получаем информацию о файле
if (!GetFileInformationByHandle(hFile, &bhfi))
{
  cerr << "Get file information by handle failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
// распечатываем информацию о файле
cout << "File attributes: " << bhfi.dwFileAttributes << endl
  << "Creation time: high date: "
    << bhfi.ftCreationTime.dwHighDateTime << endl
  << "Creation time: low date: "
    << bhfi.ftCreationTime.dwLowDateTime << endl
```

```
    << "Last access time: high date: "
      << bhfi.ftLastAccessTime.dwHighDateTime << endl
    << "Last access time: low date: "
      << bhfi.ftLastAccessTime.dwLowDateTime << endl
    << "Last write time: high date: "
      << bhfi.ftLastWriteTime.dwHighDateTime << endl
    << "Last write time: low date: "
      << bhfi.ftLastWriteTime.dwLowDateTime << endl
    << "Volume serial number: " << bhfi.dwVolumeSerialNumber << endl
    << "File size high: " << bhfi.nFileSizeHigh << endl
    << "File size low: " << bhfi.nFileSizeLow << endl
    << "Number of links: " << bhfi.nNumberOfLinks << endl
    << "File index high: " << bhfi.nFileIndexHigh << endl
    << "File index low: " << bhfi.nFileIndexLow << endl;

  // закрываем дескриптор файла
  CloseHandle(hFile);

  return 0;
}
```

Структура типа `BY_HANDLE_FILE_INFORMATION` содержит структуру типа `FILETIME`, которая служит для хранения времени. Эта структура имеет следующий формат:

```
typedef struct _FILETIME {
  DWORD  dwLowDateTime;    // младшая часть времени
  DWORD  dwHighDateTime;   // старшая часть времени
} FILETIME,  *PFILETIME;
```

Само время задается в интервалах, каждый из которых равен 100 наносекунд. Естественно, что такое время неудобно просматривать пользователю. Поэтому для перевода времени в более удобную форму существует функция `FileTimeToSystemTime`, которая имеет следующий прототип:

```
BOOL  FileTimeToSystemTime(
  CONST  FILETIME  *lpFileTime;  // указатель на время в формате "файл"
  LPSYSTEMTIME  lpSystemTime  // указатель на время в формате "система"
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`.

Параметр lpFileTime этой функции должен указывать на структуру типа FILETIME, которая содержит время в формате, используемом для хранения в файловой системе.

Параметр lpSystemTime должен указывать на структуру типа SYSTEMTIME, которая имеет следующий формат:

```
typedef struct  _SYSTEMTIME {
  WORD  wYear;       // год
  WORD  wMonth;      // месяц
  WORD  wDayOfWeek;  // день недели
  WORD  wDay;        // день
  WORD  wHour;       // час
  WORD  wMinute;     // минута
  WORD  wSecond;     // секунда
  WORD  wMilliseconds;     // миллисекунда
} SYSTEMTIME, *LPSYSTEMTIME;
```

В листинге 24.16 приведена программа, которая использует функцию FileTimeToSystemTime для перевода времени из формата файловой системы в системный формат.

Листинг 24.16. Преобразование времени в системный формат

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;
  BY_HANDLE_FILE_INFORMATION  bhfi;  // информация о файле
  SYSTEMTIME  st;    // системное время

  // открываем файл для чтения
  hFile = CreateFile(
    "C:\\demo_file.dat",    // имя файла
    0,                      // получение информации о файле
    0,                      // монопольный доступ к файлу
    NULL,                   // защиты нет
    OPEN_EXISTING,          // открываем существующий файл
    FILE_ATTRIBUTE_NORMAL, // обычный файл
    NULL                    // шаблона нет
```

```
);
// проверяем на успешное открытие
if (hFile == INVALID_HANDLE_VALUE)
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
// получаем информацию о файле
if (!GetFileInformationByHandle(hFile, &bhfi))
{
  cerr << "Get file information by handle failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
// переводим время создания файла в системное время
if (!FileTimeToSystemTime(&(bhfi.ftCreationTime), &st))
{
  cerr << "File time to system time failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
// распечатываем системное время
cout << "File creation time in system format: " << endl
  << "\tYear: " << st.wYear << endl
  << "\tMonth: " << st.wMonth << endl
  << "\tDay of week: " << st.wDayOfWeek << endl
  << "\tDay: " << st.wDay << endl
  << "\tHour: " << st.wHour << endl
  << "\tMinute: " << st.wMinute << endl
```

```
    << "\tSecond: " << st.wSecond << endl
    << "\tMilliseconds: " << st.wMilliseconds << endl;

  // закрываем дескриптор файла
  CloseHandle(hFile);

  return 0;
}
```

Определить тип файла можно при помощи функции `GetFileType`, которая имеет следующий прототип:

```
DWORD  GetFileType(
  HANDLE  hFile    // дескриптор файла
);
```

Единственным параметром этой функции является дескриптор файла. Функция `GetFileType` возвращает одно из следующих значений:

- `FILE_TYPE_UNKNOWN` — неизвестный тип файла;
- `FILE_TYPE_DISK` — дисковый файл;
- `FILE_TYPE_CHAR` — символьный файл;
- `FILE_TYPE_PIPE` — именованный или анонимный канал.

Отметим, что под символьным файлом обычно понимается принтер или консоль.

В листинге 24.17 приведена программа, которая определяет тип файла, используя функцию `GetFileType`.

Листинг 24.17. Определение типа файла

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;
  DWORD  dwFileType;

  // открываем файл для чтения
  hFile = CreateFile(
    "C:\\demo_file.dat",   // имя файла
```

```
    0,                      // получение информации о файле
    0,                      // монопольный доступ к файлу
    NULL,                   // защиты нет
    OPEN_EXISTING,          // открываем существующий файл
    FILE_ATTRIBUTE_NORMAL, // обычный файл
    NULL                    // шаблона нет
  );
  // проверяем на успешное открытие
  if (hFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  // определяем тип файла
  dwFileType = GetFileType(hFile);
    // распечатываем тип файла
  switch (dwFileType)
  {
  case FILE_TYPE_UNKNOWN:
    cout << "Unknown type file." << endl;
    break;
  case FILE_TYPE_DISK:
    cout << "Disk type file." << endl;
    break;
  case FILE_TYPE_CHAR:
    cout << "Char type file." << endl;
    break;
  case FILE_TYPE_PIPE:
    cout << "Pipe type file." << endl;
    break;
  default:
    break;
  }

  return 0;
}
```

На платформе Windows NT можно определить, является ли файл исполняемым. Для этого нужно использовать функцию `GetBinaryType`, которая имеет следующий прототип:

```
BOOL  GetBinaryType(
  LPCTSTR  lpApplicationName,  // имя приложения
  LPDWORD  lpBinaryType        // тип исполняемого файла
);
```

Если файл является исполняемым, то эта функция возвращает ненулевое значение, в противном случае — `FALSE`.

В параметре `lpApplicationName` устанавливается указатель на строку, которая содержит имя проверяемого файла.

Параметр `lpBinaryType` должен указывать на переменную типа `DWORD`, в которую функция `GetBinaryType` помещает тип исполняемого файла. Этот тип может принимать одно из следующих значений:

- `SCS_32BIT_BINARY` — приложение Win32;
- `SCS_DOS_BINARY` — приложение MS-DOS;
- `SCS_OS216_BINARY` — 16-битовое приложение OS/2;
- `SCS_PIF_BINARY` — PIF-файл;
- `SCS_POSIX_BINARY` — приложение POSIX;
- `SCS_WOW_BINARY` — 16-битовое приложение Windows.

В листинге 24.18 приведена программа, которая определяет тип исполняемого файла, используя функцию `GetBinaryType`.

Листинг 24.18. Определение типа исполняемого файла

```
#include <windows.h>
#include <iostream.h>

int main()
{
  DWORD  dwBinaryType;

  // определяем тип файла
  if(!GetBinaryType("C:\\temp.exe", &dwBinaryType))
  {
    cerr << "Get binary type failed." << endl
      << "The file may not be executable." << endl
```

```
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }

  // распечатываем тип файла
  if (dwBinaryType == SCS_32BIT_BINARY)
    cout << "The file is Win32 based application." << endl;
  else
    cout << "The file is not Win32 based application." << endl;

  return 0;
}
```

Глава 25

Работа с каталогами (папками) в Windows

25.1. Создание каталога

Для создания каталога используется функция `CreateDirectory`, которая имеет следующий прототип:

```
BOOL  CreateDirectory(
  LPCTSTR  lpPathName,      // имя каталога
  LPSECUTITY_ATTRIBUTES  lpSecurutyAttributes  // атрибуты защиты
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. В параметре `lpPathName` задается указатель на символьную строку, которая содержит имя создаваемого каталога, а параметр `lpSecurutyAttributes` задает атрибуты безопасности этого каталога.

В листинге 25.1 приведена программа, которая создает каталог.

Листинг 25.1. Пример создания каталога

```
#include <windows.h>
#include <iostream.h>

int main()
{
  // создаем каталог
  if (!CreateDirectory("C:\\demo_dir", NULL))
  {
    cerr << "Create directory failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
```

```
    cin.get();

    return 0;
  }

  cout << "The directory is created." << endl;

  return 0;
}
```

После создания каталога можно получить его дескриптор, используя для этого функцию `CreateFile` с установленным флагом `FILE_FLAG_BACKUP_SEMANTICS`.

Для создания подкаталогов можно использовать функцию `CreateDirectoryEx`, которая позволяет наследовать атрибуты другого каталога, который в этом случае называется шаблонным каталогом. В качестве шаблонного каталога можно указать родительский каталог, в котором создается текущий каталог. Функция `CreateDirectoryEx` имеет следующий прототип:

```
BOOL  CreateDirectoryEx(
  LPCTSTR  lpTemplateDirectory,    // имя шаблонного каталога
  LPCTSTR  lpNewDirectory,         // имя нового каталога
  LPSECUTITY_ATTRIBUTES  lpSecurutyAttributes  // атрибуты защиты
);
```

Эта функция отличается от функции `CreateDirectory` только наличием параметра `lpTemplateDirectory`, который содержит указатель на имя шаблонного каталога.

Например, программа из листинга 25.2 создает подкаталог в каталоге, созданном программой из листинга 25.1.

Листинг 25.2. Пример создания подкаталога

```
#include <windows.h>
#include <iostream.h>

int main()
{
  // создаем подкаталог
  if (!CreateDirectoryEx("C:\\demo_dir", "C:\\demo_dir\\demo_subdir", NULL))
  {
    cerr << "Create directory failed." << endl
```

```
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }

  cout << "The subdirectory is created." << endl;

  return 0;
}
```

25.2. Поиск файлов в каталоге

Для поиска файлов, находящихся в каталоге, используются функции `FindFirstFile` и `FindNextFile`. Причем функция `FindFirstFile` находит первый файл, имя которого удовлетворяет образцу поиска, а функция `FindNextFile` находит последующие такие файлы. При этом отметим, что в образцах имен файлов для поиска могут использоваться символы-заместители ? и *. Символ ? замещает один неизвестный символ в имени файла, а символ * — любую последовательность символов.

Рассмотрим функцию `FindFirstFile`, которая предназначена для нахождения первого файла в каталоге. Эта функция имеет следующий прототип:

```
HANDLE  FindFirstFile(
  LPCTSTR  lpFileName,     // образец имени для поиска
  LPWIN32_FIND_DATA  lpFindFileData  // адрес данных о файле
);
```

В случае успешного завершения функция `FindFirstFile` возвращает дескриптор для поиска файлов, который используется в дальнейшем функцией `FindNextFile`, а в случае неудачи — значение `INVALID_HANDLE_VALUE`.

В параметре `lpFileName` функции должен быть установлен указатель на символьную строку, которая содержит образец имени файла для поиска.

Параметр `lpFindFileData` должен указывать на структуру типа `WIN32_FIND_DATA`, в которую функция запишет о найденном файле. Эта структура имеет следующий формат:

```
typedef struct  _WIN32_FIND_DATA {
  DWORD  dwFileAttributes;       // атрибуты файла
  FILETIME  ftCreationTime;      // время создания файла
  FILETIME  ftLastAccessTime;    // время последнего доступа к файлу
```

```
  FILETIME  ftLastWriteTime;     // время последней записи в файл
  DWORD  nFileSizeHigh;     // старшая часть размера файла
  DWORD  nFileSizeLow;      // младшая часть размера файла
  DWORD  dwReserved0;       // тег для преобразования файла
  DWORD  dwReserved1;       // не используется
  CHAR  cFileName[ MAX_PATH ];      // длинное имя файла
  CHAR  cAlternateFileName[ 14 ];  // короткое имя файла
} WIN32_FIND_DATA,  *PWIN32_FIND_DATA,  *LPWIN32_FIND_DATA;
```

В случае успешного завершения функция `FindNextFile` записывает в эту структуру данные о первом найденном файле. Отметим, что в поле `dwFileAttributes` структуры `WIN32_FIND_DATA` могут быть установлены следующие флаги:

- `FILE_ATTRIBUTE_ARCHIVE` — архивный файл;
- `FILE_ATTRIBUTE_COMPRESSED` — сжатый файл;
- `FILE_ATTRIBUTE_DIRECTORY` — каталог;
- `FILE_ATTRIBUTE_ENCRYPTED` — зашифрованный файл;
- `FILE_ATTRIBUTE_HIDDEN` — скрытый файл;
- `FILE_ATTRIBUTE_NORMAL` — нормальный файл;
- `FILE_ATTRIBUTE_OFFLINE` — файл во внешней памяти;
- `FILE_ATTRIBUTE_READONLY` — файл предназначен только для чтения;
- `FILE_ATTRIBUTE_REPARSE_POINT` — файл содержит точку преобразования;
- `FILE_ATTRIBUTE_SPARSE_FILE` — разряженный файл;
- `FILE_ATTRIBUTE_SYSTEM` — системный файл;
- `FILE_ATTRIBUTE_TEMPORARY` — временный файл.

Теперь рассмотрим функцию `FindNextFile`, которая предназначена для нахождения в каталоге последующих файлов после первого найденного файла. Порядок перечисления файлов зависит от файловой системы. В файловой системе FAT файлы перечисляются в порядке их создания, а в файловых системах NTFS и CDFS — перечисляются в алфавитном порядке. Эта функция имеет следующий прототип:

```
BOOL  FindNextFile(
  HANDLE  hFindFile,     // дескриптор для поиска файлов
  LPWIN32_FIND_DATA  lpFindFileData    // адрес данных о файле
);
```

В случае успешного завершения функция `FindNextFile` возвращает ненулевое значение, а в случае неудачи — `FALSE`.

В параметре hFileFind этой функции должен быть установлен дескриптор для поиска файлов, который был получен вызовом функции FindFirstFile.

Параметр lpFindFileData должен содержать адрес структуры типа WIN32_FIND_DATA, в которую функция FindNextFile в случае своего успешного завершения поместит информацию о следующем найденном файле.

Особенно отметим, что как функция FindFirstFile, так и функция FindNextFile возвращают в структуре типа WIN32_FIND_DATA информацию как о файлах, так и о подкаталогах. В листинге 25.3 приведена программа, которая выводит на консоль информацию обо всех файлах и подкаталогах заданного каталога. Как видно из результатов, в этот список входят текущий и родительский каталоги.

Кроме того, отметим, что после завершения поиска файлов нужно вызвать функцию FindClose, которая закрывает дескриптор поиска файлов и имеет следующий прототип:

```
BOOL  FindClose(
  HANDLE  hFindFile  // дескриптор поиска файла
);
```

Функция FindClose имеет единственный параметр hFindFile, который должен содержать дескриптор поиска файла. В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — FALSE.

Листинг 25.3. Пример поиска файлов в каталоге

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFindFile;
  WIN32_FIND_DATA  fd;

  // находим первый файл
  hFindFile = FindFirstFile("C:\\demo_dir\\*", &fd);
  if (hFindFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Find first file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
```

```
    cin.get();

    return 0;
  }
  // выводим на консоль имя первого файла
  cout << "The first file name: " << fd.cFileName << endl;
  // находим следующий файл и выводим на консоль его имя
  while (FindNextFile(hFindFile, &fd))
    cout << "The next file name: " << fd.cFileName << endl;
  // закрываем дескриптор поиска
  FindClose(hFindFile);

  return 0;
}
```

Для более продвинутого поиска файлов может использоваться функция `FindFirstFileEx`, которая позволяет учитывать при поиске файла не только его имя, но также и его атрибуты. Отметим, что эта функция поддерживается только в операционных системах Windows NT/2000/XP.

25.3. Удаление каталога

Для удаления пустого каталога предназначена функция `RemoveDdirectory`, которая имеет следующий прототип:

```
BOOL  RemoveDirectory(
  LPCTSTR  lpPathName    // имя каталога
);
```

Единственный параметр этой функции должен указывать на символьную строку, содержащую имя удаляемого каталога. В случае успешного завершения функция `RemoveDdirectory` возвращает ненулевое значение, а в случае неудачи — `FALSE`.

В листинге 25.4 приведена программа, в которой удаляется каталог, используя функцию `RemoveDdirectory`. Еще раз отметим, что для успешного удаления каталог должен быть пуст.

Листинг 25.4. Пример удаления пустого каталога

```
#include <windows.h>
#include <iostream.h>

int main()
```

```
{
  // удалаем каталог
  if (!RemoveDirectory("C:\\demo_dir"))
  {
    cerr << "Remove directory failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }

  cout << "The directory is removed." << endl;

  return 0;
}
```

На практике каталог редко бывает пустым, поэтому прежде чем удалить сам каталог, нужно удалить все находящиеся в нем подкаталоги и файлы. Опишем, как удалять из каталога файлы. Этот же подход используется и для удаления файлов из подкаталогов. В листинге 25.5 приведена программа, которая удаляет из каталога файлы, а затем удаляет сам каталог.

Листинг 25.5. Пример удаления каталога с файлами

```
#include <windows.h>
#include <iostream.h>
#include <stdio.h>

int main()
{
  HANDLE  hFindFile;
  WIN32_FIND_DATA  fd;
  char  szFullFileName[MAX_PATH];

  // находим первый файл
  hFindFile = FindFirstFile("C:\\demo_dir\\*", &fd);
  if (hFindFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Find first file failed." << endl
```

```
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  // выводим на консоль имя первого файла
  cout << "The first file name: " << fd.cFileName << endl;
  // удаляем из каталога файлы
  while (FindNextFile(hFindFile, &fd))
  {
    // если это не подкаталог, то удаляем его
    if (!(fd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY))
    {
      // формируем имя файла
      sprintf(szFullFileName, "C:\\demo_dir\\%s", fd.cFileName);
      // удаляем файл
      if (!DeleteFile(szFullFileName))
      {
        cerr << "Delete file failed." << endl
          << "The last error code: " << GetLastError() << endl;
        cout << "Press any key to finish.";
        cin.get();

        return 0;
      }
      else
        cout << "The next file: " << fd.cFileName << " is deleted." << endl;
    }
    else
      cout << "The next directory: " << fd.cFileName << " is not deleted."
<< endl;
  }
  // закрываем дескриптор поиска
  if (!FindClose(hFindFile))
  {
            cout << "Find close failed." << endl;
            return ();
  }
  // удалаем каталог
  if (!RemoveDirectory("C:\\demo_dir"))
  {
    cerr << "Remove directory failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
```

```
    cin.get();

    return 0;
  }

  cout << "The directory is removed." << endl;

  // закрываем дескриптор поиска
  FindClose(hFindFile);

  return 0;
}
```

25.4. Перемещение каталога

Так же как и файлы, каталоги можно перемещать при помощи функции `MoveFile`. В этом случае на исполнение функции `MoveFile` накладывается одно ограничение, которое заключается в том, что можно перемещать только каталоги, которые находятся на одном томе. Напомним, что функция `MoveFile` имеет следующий прототип:

```
BOOL  MoveFile(
  LPCTSTR  lpExistingFileName,   // имя существующего файла
  LPCTSTR  lpNewFileName         // имя нового файла
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. В случае перемещения каталогов параметры `lpExistingFileName` и `lpNewFileName` должны указывать на строки, которые содержат имена перемещаемого и нового каталогов соответственно.

В листинге 25.6 приведена программа, которая выполняет перемещение каталога, используя функцию `MoveFile`.

Листинг 25.6. Перемещение каталога

```
#include <windows.h>
#include <iostream.h>

int main()
{
  // перемещаем каталог
```

```
  if(!MoveFile("C:\\demo_dir", "C:\\new_dir"))
  {
    cerr << "Move file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }

  cout << "The directory is moved." << endl;

  return 0;
}
```

25.5. Определение и установка текущего каталога

Каталог, из которого стартует приложение, называется *текущим каталогом* этого приложения. Как правило, в текущем каталоге находятся необходимые для работы приложения, исполняемые файлы и библиотеки. Определить имя текущего каталога можно при помощи функции `GetCurrentDirectory`, которая имеет следующий прототип:

```
DWORD  GetCurrentDirectory(
  DWORD  nBufferLength,    // длина буфера для имени каталога
  LPTSTR  lpBuffer         // адрес буфера для имени каталога
);
```

Если размер буфера достаточен для имени текущего каталога, то функция `GetCurrentDirectory` в случае успешного завершения возвращает количество символов, записанных в буфер, на который указывает параметр `lpBuffer`, а в случае неудачи — `0`. Если же размер буфера недостаточен для имени каталога, то функция `GetCurrentDirectory` возвращает необходимую длину буфера для имени текущего каталога, включая завершающий пустой символ.

В параметре `nBufferLength` должна быть задана длина буфера, куда функция `GetCurrentDirectory` записывает имя текущего каталога.

Параметр `lpBuffer` должен указывать на буфер, куда функция `GetCurrentDirectory` записывает имя текущего каталога.

Приложение может изменить имя текущего каталога, используя функцию SetCurrentDirectory, которая имеет следующий прототип:

```
BOOL  SetCurrentDirectory(
  LPCTSTR  lpPathName  // имя нового текущего каталога
);
```

В случае успешного завершения функция SetCurrentDirectory возвращает ненулевое значение, а в случае неудачи — FALSE.

Параметр lpPathName должен указывать на символьную строку, которая содержит имя нового текущего каталога.

В листинге 25.7 приведена программа, которая определяет имя текущего каталога, а затем устанавливает новый каталог в качестве текущего каталога.

Листинг 25.7. Определение и установка текущего каталога

```
#include <windows.h>
#include <iostream.h>

int main()
{
  DWORD  dwNumberOfChar;
  char  szDirName[MAX_PATH];

  // определяем имя текущего каталога
  dwNumberOfChar = GetCurrentDirectory(MAX_PATH, szDirName);
  if (dwNumberOfChar == 0)
  {
    cerr << "Get current directory failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  // выводим на консоль имя текущего каталога
  cout << "Current directory name: " << szDirName << endl;
  // устанавливаем текущий каталог для удаления из него файлов
  if (!SetCurrentDirectory("C:\\demo_dir"))
  {
```

```
    cerr << "Set current directory failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  // определяем имя нового текущего каталога
  dwNumberOfChar = GetCurrentDirectory(MAX_PATH, szDirName);
  if (dwNumberOfChar == 0)
  {
    cerr << "Get current directory failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  // выводим на консоль имя нового текущего каталога
  cout << "Current directory name: " << szDirName << endl;

  return 0;
}
```

25.6. Наблюдение за изменениями в каталоге

В операционных системах Windows предусмотрены функции, позволяющие следить за изменениями, происходящими в каталогах. Для этих целей предназначены функции `FindFirstChangeNotification` и `FindFirstChangeNotification`, которые отслеживают соответственно первое и последующие изменения, происходящие в каталоге.

Сначала рассмотрим функцию `FindFirstChangeNotification`, которая имеет следующий прототип:

```
HANDLE  FindFirstChangeNotification(
  LPCTSTR  lpPathName,       // имя каталога
  BOOL     bWatchSubtree,    // опция наблюдения
  DWORD    dwNotifyFilter    // условия фильтра
);
```

В случае успешного завершения функция FindFirstChangeNotification возвращает дескриптор для наблюдения за каталогом. Этот дескриптор может использоваться в функциях ожидания для отслеживания изменений в каталоге. В случае неудачи функция FindFirstChangeNotification возвращает значение INVALID_HANDLE_VALUE.

Параметр lpPathName должен указывать на символьную строку, содержащую имя каталога.

Параметр bWatchSubtree определяет структуру дерева каталогов, которые необходимо наблюдать. Если этот параметр равен FALSE, то система будет отслеживать только изменения в заданном каталоге. Если же значение этого параметра равно TRUE, то система будет отслеживать изменения как в заданном каталоге, так и в подкаталогах, содержащихся в нем.

Параметр dwNotifyFilter задает события, которые отслеживаются. Эти события соответствуют флагам, которые могут быть установлены в этом параметре. Возможно установить следующие флаги:

- FILE_NOTIFY_CHANGE_FILE_NAME — изменение имени файла;
- FILE_NOTIFY_CHANGE_DIR_NAME — изменение имени каталога;
- FILE_NOTIFY_CHANGE_ATTRIBUTES — изменение атрибутов;
- FILE_NOTIFY_CHANGE_SIZE — изменение размеров;
- FILE_NOTIFY_CHANGE_LAST_WRITE — изменение времени последней записи;
- FILE_NOTIFY_CHANGE_SECURITY — изменение атрибутов защиты.

Теперь рассмотрим функцию FindNextChangeNotification, которая предназначена для наблюдения за последующими изменениями, происходящими в каталоге. Эта функция имеет следующий прототип:

```
BOOL  FindNextChangeNotification(
  HANDLE  hChangeHandle    // дескриптор для наблюдения за изменениями
);
```

В случае успешного завершения функция FindNextChangeNotification возвращает ненулевое значение, а в случае неудачи — FALSE.

В параметре hChangeHandle должен быть установлен дескриптор для наблюдения за изменениями в каталоге, который был получен вызовом функции FindFirstChangeNotification.

После завершения наблюдения за изменениями в каталоге нужно вызвать функцию FindCloseChangeNotification, которая закрывает дескриптор наблюдения за изменениями в каталоге. Эта функция имеет следующий прототип:

```
BOOL  FindCloseChangeNotification(
  HANDLE  hChangeHandle    // дескриптор для наблюдения за изменениями
);
```

Функция `FindCloseChangeNotification` имеет единственный параметр `hChangeHandle`, который должен содержать дескриптор наблюдения за изменениями в каталоге. В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`.

В листинге 25.8 приведена программа, которая наблюдает за изменениями в каталоге. Программа отслеживает только два события, каждое из которых может быть вызвано или изменением имени файла, хранящегося в каталоге, или изменением размера каталога.

Листинг 25.8. Пример отслеживания изменений в каталоге

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hChangeHandle;

  // находим первое изменение в каталоге
  hChangeHandle = FindFirstChangeNotification(
    "C:\\demo_dir",    // имя каталога
    TRUE,              // отслеживать также подкаталоги
    FILE_NOTIFY_CHANGE_FILE_NAME // отслеживать изменение имени каталога
    | FILE_NOTIFY_CHANGE_SIZE    // и изменение его размера
  );

  if (hChangeHandle == INVALID_HANDLE_VALUE)
  {
    cerr << "Find first change notification failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  // нужно изменить каталог
  cout << "Wait for changes in the directory." << endl;
  // ждем первого изменения в каталоге
  if (WaitForSingleObject(hChangeHandle, INFINITE) == WAIT_OBJECT_0)
```

```
    cout << "First notification: the directory was changed." << endl;
  else
  {
    cerr << "Wait for single object failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  // находим второе изменение в каталоге
  if (!FindNextChangeNotification(hChangeHandle))
  {
    cerr << "Find next change notification failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  // ждем второго изменения в каталоге
  if (WaitForSingleObject(hChangeHandle, INFINITE) == WAIT_OBJECT_0)
    cout << "Next notification: the directory was changed." << endl;
  else
  {
    cerr << "Wait for single object failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  // закрываем дескриптор поиска
  FindCloseChangeNotification(hChangeHandle);

  return 0;
}
```

Часть VIII

Асинхронная обработка данных

Глава 26

Асинхронный вызов процедур

26.1. Механизм асинхронного вызова процедур

Асинхронной процедурой называется функция, которая исполняется асинхронно в контексте какого-либо потока. Для исполнения асинхронной процедуры необходимо выполнить три условия:

- определить асинхронную процедуру;
- определить поток, в контексте которого эта процедура будет выполняться;
- дать разрешение на выполнение асинхронной процедуры.

Рассмотрим подробнее каждое из этих условий.

Асинхронная процедура должна определяться функцией, которая имеет следующий прототип:

```
VOID  CALLBACK  имя_асинхронной_процедуры(DWORD  dwParam);
```

Отсюда видно, что асинхронная процедура не возвращает значения и должна иметь только один параметр.

Теперь рассмотрим, как определятся поток, в контексте которого исполняется асинхронная процедура. Для этого в операционных системах Windows существует функция `QueueUserAPC`, которая связывает асинхронную процедуру с потоком. Как работает эта функция, будет рассмотрено в *разд. 26.2*. Сейчас же скажем, что каждый поток имеет очередь асинхронных процедур и, вызвав функцию `QueueUserAPC`, мы помещаем нашу асинхронную процедуру в эту очередь. Очередь асинхронных процедур работает по алгоритму FIFO *(см. разд. 20.3)*. Фактически каждый поток имеет две очереди асинхронных процедур: одну пользовательскую, а вторую системную. В пользовательскую очередь асинхронная процедура ставится при помощи функции `QueueUserAPC`, а в системную очередь асинхронная процедура ставится операционной системой.

Наконец, рассмотрим, как вызывается асинхронная процедура. Для вызова асинхронной процедуры необходимо, чтобы поток находился в *настороженном* (alertable) состоянии. Перевести поток в настороженное состояние можно при помощи вызова функций `SleepEx`, `WaitForSingleObjectEx`, `WaitForMultipleObjectsEx` и `SignalObjectAndWait`. Все эти функции будут рассмотрены в следующих разделах. Если поток входит в настороженное состояние, то при следующем выделении кванта времени этому потоку сначала выполняются все асинхронные процедуры этого потока, затем настороженное состояние потока сбрасывается и начинает выполняться сам поток. Отметим, что системные асинхронные процедуры вызываются всегда перед передачей управления потоку.

26.2. Установка асинхронных процедур

Для установки асинхронной процедуры в очередь потока используется функция `QueueUserAPC`, которая имеет следующий прототип:

```
DWORD  QueueUserAPC(
  PAPCFUNC  pfnAPC,    // имя асинхронной процедуры
  HANDLE    hThread,   // дескриптор потока
  DWORD     dwData     // параметр асинхронной процедуры
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — 0. При этом отметим, что в случае неудачи не устанавливается код последней ошибки, который может быть получен вызовом функции `GetLastError`.

Параметр `pfnAPC` должен содержать имя функции, которая включается в очередь асинхронных процедур. Причем прототип этой функции должен совпадать с прототипом асинхронной процедуры, который был рассмотрен в предыдущем параграфе.

В параметре `hThread` должен быть установлен дескриптор потока, в очередь асинхронных процедур которого включается функция, заданная параметром `pfnAPC`.

В параметр `dwData` устанавливается значение, которое передается асинхронной процедуре при ее вызове.

Примеры установки и вызова асинхронных процедур будут рассмотрены в следующих разделах, посвященных функциям перевода потока в настороженное состояние.

26.3. Приостановка потока

Поток может перевести себя в состояние ожидания до истечения некоторого интервала времени или вызова асинхронной процедуры посредством вызова функции `SleepEx`, которая имеет следующий прототип:

```
DWORD  SleepEx(
  DWORD  dwMilliseconds,    // интервал времени
  BOOL   bAlertable         // режим завершения
);
```

В случае истечения заданного интервала времени эта функция возвращает значение 0. Если же завершение функции вызвано другими причинами, то функция `SleepEx` возвращает ненулевое значение. При этом, если возврат из функции происходит по причине завершения операции асинхронного ввода-вывода, то функция возвращает значение `WAIT_IO_COMPLETION`.

Параметр `dwMilliseconds` задает в миллисекундах интервал времени, на который может быть заблокирован поток. Для задания бесконечного интервала времени нужно использовать значение `INFINITE`.

Параметр `bAlertable` может принимать одно из двух значений — `FALSE` или `TRUE`. Если установлено значение `FALSE`, то поток не переходит в настороженное состояние. Следовательно в этом случае асинхронные процедуры не вызываются и поток блокируется на заданный интервал времени. Если же в этом параметре установлено значение `TRUE`, то поток переходит в настороженное состояние и блокируется до тех пор, пока не выполнится одно из следующих условий:

- закончится заданный интервал времени ожидания;
- асинхронная процедура поставлена в очередь потока;
- процедура завершения асинхронной операции ввода-вывода поставлена в очередь потока.

В программе из листинга 26.1 показано, как установить асинхронную процедуру и вызвать ее при помощи функции `SleepEx`.

Листинг 26.1. Вызов асинхронной процедуры путем приостановки потока

```
#define _WIN32_WINNT 0x0400  // версия не ниже, чем Windows NT 4.0

#include <windows.h>
#include <iostream.h>

HANDLE  hThread;   // дескриптор потока
```

```
DWORD  IDThread;   // идентификатор потока
DWORD  dwRet;      // возвращаемое значение при установке асинхронной
                   // процедуры

// процедура, которая вызывается асинхронно
void CALLBACK a_proc(DWORD p)
{
  int n;
  DWORD *ptr = (DWORD*)p;

  cout << "The asynchronous procedure is called." << endl;
  // ждем ввода целого числа
  cout << "Input integer: ";
  cin >> n;
  // увеличиваем счетчик
  *ptr += n;

  cout << "The asynchronous procedure is returned." << endl;
}

// поток, в очередь которого включается асинхронная процедура
DWORD WINAPI add(LPVOID ptr)
{
  // распечатываем начальное значение счетчика
  cout << "Initial count = " << *(DWORD*)ptr << endl;
  // ждем, пока выполнится асинхронная процедура
  SleepEx(INFINITE, TRUE);
  // распечатываем конечное значение счетчика
  cout << "Final count = " << *(DWORD*)ptr << endl;

  return 0;
}

// главный поток
int main()
{
  DWORD  count = 10;

  // запускаем поток
```

```
  hThread = CreateThread(NULL, 0, add, &count, 0, &IDThread);
  if (hThread == NULL)
    return GetLastError();

  Sleep(1000); // чтобы успел стартовать поток

  // устанавливаем асинхронную процедуру для потока
  dwRet = QueueUserAPC(a_proc, hThread, (DWORD) &count);
  if (!dwRet)
  {
    cout << "Queue user APC failed:" << dwRet << endl;
    return 0;
  }

  // ждем, пока поток add закончит работу
  WaitForSingleObject(hThread, INFINITE);
  // закрываем дескриптор потока
  CloseHandle(hThread);

  return 0;
}
```

В заключение этого раздела сделаем следующее замечание: включить асинхронную процедуру в очередь потока может как сам поток, так и любой другой поток. В нашем случае поток `main` включает асинхронную процедуру `a_proc` в очередь асинхронных процедур потока `add`.

26.4. Ожидание события

Для ожидания наступления некоторого события или выполнения асинхронной процедуры можно использовать функцию `WaitForSingleObjectEx`, которая имеет следующий прототип:

```
DWORD  WaitForSingleObjectEx(
  HANDLE  hHandle,         // дескриптор объекта
  DWORD   dwMilliseconds,  // временной интервал в миллисекундах
  BOOL    bAlertable       // режим завершения
);
```

В случае успешного завершения эта функция возвращает одно из следующих значений:

- `WAIT_OBJECT_0` — объект в сигнальном состоянии;
- `WAIT_ABANDONED` — забытый мьютекс;

- WAIT_IO_COMPLETION — асинхронная процедура поставлена в очередь;
- WAIT_TIMEOUT — истек временной интервал.

В случае неудачи функция WaitForSingleObjectEx возвращает -1.

Кратко опишем назначение параметров этой функции.

Параметр hHandle должен содержать дескриптор объекта, сигнальное состояние которого ожидает функция.

Параметр dwMilliseconds должен содержать временной интервал, заданный в миллисекундах, в течение которого функция ожидает перехода объекта в сигнальное состояние или включение асинхронной процедуры в очередь потока.

Параметр bAlertable может принимать одно из двух значений — FALSE или TRUE. Если установлено значение FALSE, то поток не переходит в настороженное состояние. Следовательно в этом случае асинхронные процедуры не вызываются и поток блокируется на заданный интервал времени. Если же в этом параметре установлено значение TRUE, то поток переходит в настороженное состояние и блокируется до тех пор, пока не выполнится одно из следующих условий:

- закончится заданный интервал времени ожидания;
- асинхронная процедура поставлена в очередь потока;
- процедура завершения асинхронной операции ввода-вывода поставлена в очередь потока.

В листинге 26.2 приведена программа, в которой функция WaitForSingleObjectEx используется для ожидания события или включения асинхронной процедуры в очередь потока.

Листинг 26.2. Ожидание события или вызова асинхронной процедуры

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include <iostream.h>

HANDLE  hEvent;    // дескриптор события
HANDLE  hThread;   // дескриптор потока
DWORD   IDThread;  // идентификатор потока
DWORD   dwRet;     // код возврата при установке асинхронной процедуры

// процедура, которая вызывается асинхронно
```

```
void CALLBACK a_proc(DWORD p)
{
  int n;
  DWORD *ptr = (DWORD*)p;

  cout << "The asynchronous procedure is called." << endl;
  // ждем ввода целого числа
  cout << "Input integer: ";
  cin >> n;
  // увеличиваем счетчик
  *ptr += n;

  cout << "The asynchronous procedure is returned." << endl;
}

// поток, в очередь которого включается асинхронная процедура
DWORD WINAPI add(LPVOID ptr)
{
  DWORD  dwRet;    // код возврата из функции ожидания

  // распечатываем начальное значение счетчика
  cout << "Initial count = " << *(DWORD*)ptr << endl;
  // ждем timeout, или наступление события,
  // или завершение асинхронной процедуры
  dwRet = WaitForSingleObjectEx(hEvent, 10, TRUE);
  switch (dwRet)
  {
  case WAIT_OBJECT_0:
    cout << "The event is signaled." << endl;
    break;
  case WAIT_IO_COMPLETION:
    cout << "The asynchronous procedure is done." << endl;
    break;
  case WAIT_TIMEOUT:
    cout << "Time out." << endl;
    break;
  default:
    cout << "Another case." << endl;
    break;
```

```
  }
  // распечатываем конечное значение счетчика
  cout << "Final count = " << *(DWORD*)ptr << endl;

  return 0;
}

// главный поток
int main()
{
  char c;
  DWORD  count = 10;

  // спрашиваем, что установить: событие или асинхронную процедуру
  cout << "Input: " << endl << "\te - to set event" << endl
    << "\ta - to set asynchronous procedure" << endl
    << "\tanother char - timeout" << endl
    << "->";
  cin >> c;

  // запускаем поток
  hThread = CreateThread(NULL, 0, add, &count, 0, &IDThread);
  if (hThread == NULL)
    return GetLastError();

  // создаем событие с автоматическим сбросом
  hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
  if (hEvent == NULL)
    return GetLastError();

  switch (c)
  {
  case 'e':
    // устанавливаем событие
    SetEvent(hEvent);
    break;
  case 'a':
    // устанавливаем асинхронную процедуру для потока
    dwRet = QueueUserAPC(a_proc, hThread, (DWORD) &count);
```

```
    if (!dwRet)
    {
      cout << "Queue user APC failed:" << dwRet << endl;
      return 0;
    }
    break;
  default:
    break;
  }

  // ждем, пока поток add закончит работу
  WaitForSingleObject(hThread, INFINITE);
  // закрываем дескриптор потока
  CloseHandle(hThread);

  return 0;
}
```

Для ожидания сигнального состояния нескольких объектов синхронизации можно использовать функцию `WaitForMultipleObjectsEx`, которая имеет следующий прототип:

```
DWORD  WaitForMultipleeObjectsEx(
  DWORD  nCount,           // количество объектов синхронизации
  CONST  HANDLE  lpHandles,  // массив дескрипторов объектов
  BOOL   bWaitAll,         // режим ожидания
  DWORD  dwMilliseconds,   // временной интервал в миллисекундах
  BOOL   bAlertable        // режим завершения
);
```

В случае успешного завершения эта функция возвращает одно из следующих значений:

- в интервале от значения `WAIT_OBJECT_0` до значения `WAIT_OBJECT_0 + nCount - 1`;
- в интервале от значения `WAIT_ABANDONED` до значения `WAIT_ABANDONED + nCount - 1`;
- `WAIT_IO_COMPLETION`;
- `WAIT_TIMEOUT`.

Значения `WAIT_IO_COMPLETION` и `WAIT_TIMEOUT` имеют тот же смысл, что и в функции `WaitForSingleObjectEx`, а первые два значения интерпретируются в зависимости от значения параметра `bWaitAll`.

Если параметр bWaitAll равен TRUE, то любое значение (от WAIT_OBJECT_0 до WAIT_OBJECT_0 + nCount - 1) означает, что все объекты синхронизации находятся в сигнальном состоянии. В противном случае возвращаемое значение минус WAIT_OBJECT_0 является индексом элемента массива, заданного параметром lpHandles, и этот индекс указывает на дескриптор объекта, который перешел в сигнальное состояние. Если таких объектов несколько, то индекс имеет наименьшее из возможных значений.

Далее, если значение параметра bWaitAll установлено в TRUE, то любое значение от WAIT_ABANDONED до WAIT_ABANDONED + nCount - 1 означает, что все объекты синхронизации находятся в сигнальном состоянии и, по крайней мере, один из них является забытым мьютексом. В противном случае возвращаемое значение минус WAIT_OBJECT_0 является индексом элемента массива, заданного параметром lpHandles, и этот индекс указывает на забытый мьютекс.

Теперь перейдем к параметрам функции WaitForMultipleObjectsEx.

Параметр nCount содержит количество объектов синхронизации, сигнальное состояние которых ожидает функция WaitForMultipleObjectsEx. Значение этого параметра не должно превышать величины MAXIMUM_WAIT_OBJECTS.

Параметр lpHandles должен указывать на массив дескрипторов объектов синхронизации. Количество элементов этого массива должно совпадать с величиной параметра nCount. Если хотя бы один из дескрипторов закрывается во время ожидания функции WaitForMultipleObjectsEx, то поведение этой функции неопределено.

Параметр bWaitAll задает режим ожидания функции WaitForMultipleObjectsEx. Если этот параметр установлен в TRUE, то функция ждет перехода в сигнальное состояние всех объектов синхронизации. Если же значение этого параметра установлено в FALSE, то функция ждет перехода в сигнальное состояние хотя бы одного объекта синхронизации.

Параметры dwMilliseconds и bAlertable имеют то же назначение, что и в функции WaitForSingleObjectEx.

26.5. Оповещение и ожидание события

Для оповещения о наступлении некоторого события и ожидании другого события или включения асинхронной процедуры в очередь потока используется функция SignalObjectAndWait, которая имеет следующий прототип:

```
DWORD  SignalObjectAndWait(
  HANDLE  hObjectToSignal,   // дескриптор сигнального объекта
  HANDLE  hObjectToWaitOn,   // дескриптор ожидаемого объекта
```

```
    DWORD   dwMilliseconds,    // временной интервал в миллисекундах
    BOOL    bAlertable         // режим завершения
  );
```

В случае успешного завершения эта функция возвращает одно из следующих значений:

- `WAIT_OBJECT_0` — объект в сигнальном состоянии;
- `WAIT_ABANDONED` — забытый мьютекс;
- `WAIT_IO_COMPLETION` — асинхронная процедура поставлена в очередь;
- `WAIT_TIMEOUT` — истек временной интервал.

В случае неудачи функция `WaitForSingleObjectEx` возвращает значение `-1`.

Опишем параметры этой функции.

Параметр `hObjectToSignal` должен содержать дескриптор объекта, сигнальное состояние которого устанавливается функцией `WaitForSingleObjectEx`. Таким объектом может быть событие, мьютекс или семафор.

Параметр `hObjectToWaitOn` должен содержать дескриптор объекта, сигнальное состояние которого ожидает функция.

Параметр `dwMilliseconds` должен содержать временной интервал, заданный в миллисекундах, в течение которого функция ожидает перехода объекта в сигнальное состояние или включение асинхронной процедуры в очередь потока.

Параметр `bAlertable` может принимать одно из двух значений: `FALSE` или `TRUE`. Если установлено значение `FALSE`, то поток не переходит в настороженное состояние. Следовательно, в этом случае асинхронные процедуры не вызываются и поток блокируется на заданный интервал времени. Если же в этом параметре установлено значение `TRUE`, то поток переходит в настороженное состояние и блокируется до тех пор, пока не выполнится одно из следующих условий:

- закончится заданный интервал времени ожидания;
- асинхронная процедура поставлена в очередь потока;
- процедура завершения асинхронной операции ввода-вывода поставлена в очередь потока.

Отметим, что функция `SignalObjectAndWait` работает только в операционных системах Windows NT версии 4.0 и старше. В листинге 26.3 приведена программа, в которой эта функция используется для оповещения о наступлении события и одновременном ожидании наступления другого события или включения асинхронной процедуры в очередь потока.

Листинг 26.3. Пример использования функции `SignalObjectAndWait`

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include <iostream.h>

HANDLE  hSignal;   // дескриптор события, о котором сигнализируем
HANDLE  hWait;     // дескриптор события, которое ждем
HANDLE  hThread;   // дескриптор потока
DWORD  IDThread;   // идентификатор потока
DWORD  dwRet;      // возвращаемое значение при установке асинхронной
                   // процедуры

// процедура, которая вызывается асинхронно
void CALLBACK a_proc(DWORD p)
{
  int n;
  DWORD *ptr = (DWORD*)p;

  cout << "The asynchronous procedure is called." << endl;
  // ждем ввода целого числа
  cout << "Input integer: ";
  cin >> n;
  // увеличиваем счетчик
  *ptr += n;

  cout << "The asynchronous procedure is returned." << endl;
}

// поток, в очередь которого включается асинхронная процедура
DWORD WINAPI add(LPVOID ptr)
{
  DWORD  dwRet;    // код возврата из функции ожидания

  // распечатываем начальное значение счетчика
  cout << "Initial count = " << *(DWORD*)ptr << endl;
  // ждем timeout, или наступление события,
  // или завершение асинхронной процедуры
```

```
  dwRet = SignalObjectAndWait(hSignal, hWait, 10, TRUE);
  switch (dwRet)
  {
  case WAIT_OBJECT_0:
    cout << "The event is signaled." << endl;
    break;
  case WAIT_IO_COMPLETION:
    cout << "The asynchronous procedure is done." << endl;
    break;
  case WAIT_TIMEOUT:
    cout << "Time out." << endl;
    break;
  default:
    cout << "Another case." << endl;
    break;
  }
  // распечатываем конечное значение счетчика
  cout << "Final count = " << *(DWORD*)ptr << endl;

  return 0;
}

// главный поток
int main()
{
  char c;
  DWORD  count = 10;

  // спрашиваем, что установить: событие или асинхронную процедуру
  cout << "Input: " << endl << "\te - to set event" << endl
    << "\ta - to set asynchronous procedure" << endl
    << "\tanother char - timeout" << endl
    << "->";
  cin >> c;

  // запускаем поток
  hThread = CreateThread(NULL, 0, add, &count, 0, &IDThread);
  if (hThread == NULL)
    return GetLastError();
```

```
  // создаем события с автоматическим сбросом
  hSignal = CreateEvent(NULL, FALSE, FALSE, NULL);
  if (hSignal == NULL)
    return GetLastError();
  hWait = CreateEvent(NULL, FALSE, FALSE, NULL);
  if (hWait == NULL)
    return GetLastError();

  // ждем сигнала
  WaitForSingleObject(hSignal, INFINITE);
  // запрашиваем вариант работы программы
  switch (c)
  {
  case 'e':
    // устанавливаем событие
    SetEvent(hWait);
    break;
  case 'a':
    // устанавливаем асинхронную процедуру для потока
    dwRet = QueueUserAPC(a_proc, hThread, (DWORD) &count);
    if (!dwRet)
    {
      cout << "Queue user APC failed:" << dwRet << endl;
      return 0;
    }
    break;
  default:
    break;
  }

  // ждем, пока поток add закончит работу
  WaitForSingleObject(hThread, INFINITE);
  // закрываем дескриптор потока
  CloseHandle(hThread);

  return 0;
}
```

Глава 27

Асинхронный доступ к данным

27.1. Концепция асинхронного ввода-вывода

Синхронный и асинхронный ввод-вывод концептуально нисколько не отличается от синхронного и асинхронного обмена данными, рассмотренного в *гл. 14*. При синхронной записи данных в файл поток, выдавший команду записи, блокируется до тех пор, пока данные не будут записаны в файл или буфер. При асинхронной записи данных в файл, после выдачи команды на запись данных, поток не блокируется, а продолжает свою работу. Соответственно, при синхронном чтении данных из файла поток, выдавший команду чтения, блокируется до тех пор, пока данные не будут прочитаны из файла. При асинхронном чтении данных такой блокировки не происходит, и поток продолжает свое исполнение. В операционных системах Windows асинхронный ввод-вывод также называется *перекрывающимся* (overlapped) *вводом-выводом*. Для асинхронного ввода-вывода данных в операционных системах Windows используются те же функции `ReadFile` и `WriteFile`, что и для синхронного ввода-вывода. Однако в этом случае файл должен быть открыт в режиме `FILE_FLAG_OVERLAPPED`.

Возникает вопрос: как поток узнает о завершении асинхронной операции чтения или записи? Это можно сделать двумя способами.

Первый способ заключается в том, что для этого можно использовать дескриптор файла, который устанавливается в несигнальное состояние после начала каждой асинхронной операции записи или чтения и переходит в сигнальное состояние после завершения асинхронной операции записи или чтения. Однако этот способ не может быть использован, если с одним файлом работает несколько асинхронных операций ввода-вывода. Так как в этом случае невозможно определить какая из асинхронных операций ввода-вывода завершилась.

Второй способ заключается в использовании специального события, которое устанавливается операционной системой в сигнальное состояние при завершении асинхронной операции ввода-вывода. Дескриптор этого события должен находиться в структуре типа OVERLAPPED, адрес которой передается в функции асинхронного ввода-вывода. Эта структура имеет следующий формат:

```
typedef struct  _OVERLAPPED {
  DWORD  Internal;       // для использования операционной системой
  DWORD  InternalHigh;   // для использования операционной системой
  DWORD  Offset;    // младшая часть смещения относительно начала файла
  DWORD  OffsetHigh;// старшая часть смещения относительно начала файла
  HANDLE hEvent;         // дескриптор события
}  OVERLAPPED,  *LPOVERLAPPED;
```

Поля Internal и InternalHigh этой структуры зарезервированы для использования операционной системой.

В полях Offset и OffsetHigh устанавливаются соответственно младшая и старшая части смещения относительно начала файла. Начиная с этого смещения, операционная система выполняет чтение или запись данных.

В параметре hEvent устанавливается дескриптор события, которое операционная система устанавливает в сигнальное состояние после завершения асинхронной операции ввода-вывода.

При одновременной работе нескольких асинхронных операций ввода-вывода, для каждой из них нужно определить свою структуру типа OVERLAPPED, в которой указать дескриптор отдельного события, связанного только с этой операцией ввода-вывода.

В следующих разделах будут рассмотрены примеры организации асинхронного доступа к файлам. Однако используемый подход может быть также применен и при передаче данных по именованным каналам.

27.2. Асинхронная запись данных

Асинхронная запись данных в файл работает только на платформах Windows NT/2000. Выполнить асинхронную запись данных можно при помощи функции WriteFile, которая имеет следующий прототип

```
BOOL  WriteFile(
  HANDLE   hFile,                    // дескриптор файла
  LPCVOID  lpBuffer,                 // указатель на буфер данных
  DWORD    nNumberOfBytesToWrite,    // количество записываемых байтов
  LPDWORD  lpNumberOfBytesWritten,   // количество записанных байтов
  LPOVERLAPPED  lpOverlapped   // используется при асинхронной записи
);
```

При асинхронной записи данных в файл эта функция возвращает ненулевое значение в том случае, если запись данных в файл уже завершилась до выхода из функции WriteFile. Если же функция возвращает FALSE, то нужно проверить код последней ошибки путем вызова функции GetLastError. Если эта функция возвращает значение ERROR_IO_PENDING, то это значит, что операция вывода данных еще не закончилась.

Кратко опишем назначение параметров этой функции. При этом отметим, что в отличие от синхронного вывода при асинхронном выводе также используется последний параметр lpOverlapped функции WriteFile.

Параметр hFile должен содержать дескриптор файла, причем файл должен быть открыт в режиме записи. Кроме того, должен быть установлен флаг FILE_FLAG_OVERLAPPED, который указывает на асинхронный режим ввода-вывода.

Параметр lpBuffer должен указывать на область памяти, в которую будут читаться данные.

Параметр nNumberOfBytesToWrite должен содержать количество байт, которые предполагается записать в файл посредством вызова функции WriteFile.

Параметр lpNumberOfBytesWritten должен содержать адрес памяти, в которую функция WriteFile поместит количество фактически записанных в файл байтов. При выполнении функции WriteFile операционная система записывает по этому адресу ноль, прежде чем выполнить запись данных в файл. При асинхронном выводе значение этого параметра может быть установлено в NULL. В этом случае для определения количества байт, которые функция записала в файл, можно использовать функцию GetOverlappedResult.

Параметр lpOverlapped должен указывать на структуру типа OVERLAPPED. Причем в этой структуре должны быть установлены поля Offset и OffsetHigh. В поле hEvent может быть помещен как дескриптор события, так и значение NULL. В первом случае система устанавливает событие в сигнальное состояние по завершении операции вывода. Во втором случае завершение операции вывода можно определить по дескриптору файла, который устанавливается в сигнальное состояние по завершении операции вывода, а при запуске новой операции вывода — сигнальное состояние файла сбрасывается.

В листинге 27.1 приведена программа, которая выполняет асинхронную запись данных в файл. В этой программе для сигнализации о завершении операции вывода используется дескриптор файла.

Листинг 27.1. Асинхронная запись данных в файл с использованием дескриптора файла

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
```

```
#include <iostream.h>

int main()
{
  HANDLE  hFile;     // дескриптор файла
  OVERLAPPED  ovl;   // структура управления асинхронным доступом к файлу

  // инициализируем структуру OVERLAPPED
  ovl.Offset = 0;      // младшая часть смещения равна 0
  ovl.OffsetHigh = 0;  // старшая часть смещения равна 0
  ovl.hEvent = 0;      // события нет

  // создаем файл для записи данных
  hFile = CreateFile(
    "C:\\demo_file.dat",    // имя файла
    GENERIC_WRITE,          // запись в файл
    FILE_SHARE_WRITE,       // совместный доступ к файлу
    NULL,                   // защиты нет
    OPEN_ALWAYS,            // открываем или создаем новый файл
    FILE_FLAG_OVERLAPPED,   // асинхронный доступ к файлу
    NULL                    // шаблона нет
  );
  // проверяем на успешное создание
  if (hFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    cout << "Press any key to finish.";
    cin.get();
    return 0;
  }

  // пишем данные в файл
  for (int i = 0; i < 10; ++i)
  {
    DWORD  dwBytesWrite;
    DWORD  dwRet;

    if (!WriteFile(
```

```
        hFile,           // дескриптор файла
        &i,              // адрес буфера, откуда идет запись
        sizeof(i),       // количество записываемых байтов
        &dwBytesWrite,   // количество записанных байтов
        &ovl             // запись асинхронная
      ))
    {
      dwRet = GetLastError();
      if (dwRet == ERROR_IO_PENDING)
        cout << "Write file pending." << endl;
      else
      {
        cout << "Write file failed." << endl
          << "The last error code: " << dwRet << endl;
        return 0;
      }
    }
    // ждем, пока завершится асинхронная операция записи
    WaitForSingleObject(hFile, INFINITE);
    // увеличивает смещение в файле
    ovl.Offset += sizeof(i);
  }
  // закрываем дескриптор файла
  CloseHandle(hFile);

  cout << "The file is written." << endl;

  return 0;
}
```

Теперь немного изменим программу из листинга 27.1 таким образом, чтобы для сигнализации о завершении асинхронной операции вывода использовалось событие. Измененная программа приведена в листинге 27.2.

Листинг 27.2. Асинхронная запись данных в файл с использованием события

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include <iostream.h>
```

```
int main()
{
  HANDLE  hFile;     // дескриптор файла
  HANDLE  hEndWrite; // дескриптор события
  OVERLAPPED  ovl;   // структура управления асинхронным доступом к файлу

  // создаем события с автоматическим сбросом
  hEndWrite = CreateEvent(NULL, FALSE, FALSE, NULL);
  if (hEndWrite == NULL)
    return GetLastError();

  // инициализируем структуру OVERLAPPED
  ovl.Offset = 0;          // младшая часть смещения равна 0
  ovl.OffsetHigh = 0;      // старшая часть смещения равна 0
  ovl.hEvent = hEndWrite;  // событие для оповещения завершения записи

  // создаем файл для записи данных
  hFile = CreateFile(
    "C:\\demo_file.dat",   // имя файла
    GENERIC_WRITE,         // запись в файл
    FILE_SHARE_WRITE,      // совместный доступ к файлу
    NULL,                  // защиты нет
    OPEN_ALWAYS,           // открываем или создаем новый файл
    FILE_FLAG_OVERLAPPED,  // асинхронный доступ к файлу
    NULL                   // шаблона нет
  );
  // проверяем на успешное создание
  if (hFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl
      << "The last error code: " << GetLastError() << endl;

    CloseHandle(hEndWrite);

    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
```

```
    // пишем данные в файл
  for (int i = 0; i < 10; ++i)
  {
    DWORD  dwBytesWrite;
    DWORD  dwRet;

    if (!WriteFile(
        hFile,           // дескриптор файла
        &i,              // адрес буфера, откуда идет запись
        sizeof(i),       // количество записываемых байтов
        &dwBytesWrite,   // количество записанных байтов
        &ovl             // запись асинхронная
      ))
    {
      dwRet = GetLastError();
      if (dwRet == ERROR_IO_PENDING)
        cout << "Write file pending." << endl;
      else
      {
        cout << "Write file failed." << endl
          << "The last error code: " << dwRet << endl;

        return 0;
      }
    }
    // ждем, пока завершится асинхронная операция записи
    WaitForSingleObject(hEndWrite, INFINITE);
    // увеличивает смещение в файле
    ovl.Offset += sizeof(i);
  }
  // закрываем дескрипторы
  CloseHandle(hFile);
  CloseHandle(hEndWrite);

  cout << "The file is written." << endl;

  return 0;
}
```

27.3. Асинхронное чтение данных

Асинхронное чтение данных из файла работает только в операционных системах Windows NT/2000. Выполнить асинхронное чтение данных можно при помощи функции `ReadFile`, которая имеет следующий прототип:

```
BOOL  ReadFile(
  HANDLE   hFile,                // дескриптор файла
  LPVOID   lpBuffer,             // указатель на буфер данных
  DWORD    nNumberOfBytesToRead, // количество читаемых байтов
  LPDWORD  lpNumberOfBytesRead,  // количество прочитанных байтов
  LPOVERLAPPED  lpOverlapped   // используется при асинхронной записи
);
```

При асинхронном чтении данных из файла эта функция возвращает ненулевое значение в том случае, если чтение данных из файла уже завершилось до выхода из функции `ReadFile`. Если же функция возвращает значение `FALSE`, то нужно проверить код последней ошибки путем вызова функции `GetLastError`. Если эта функция возвращает значение `ERROR_IO_PENDING`, то это значит, что операция ввода данных еще не закончилась.

Кратко опишем назначение параметров этой функции. При этом отметим, что, в отличие от синхронного ввода, при асинхронном вводе также используется последний параметр `lpOverlapped` функции `ReadFile`.

Параметр `hFile` должен содержать дескриптор файла, причем файл должен быть открыт в режиме чтения. Кроме того, должен быть установлен флаг `FILE_FLAG_OVERLAPPED`, который указывает на асинхронный режим ввода-вывода.

Параметр `lpBuffer` должен указывать на область памяти, из которой будут читаться данные.

Параметр `nNumberOfBytesToRead` должен содержать количество байт, которые предполагается читать из файла посредством вызова функции `ReadFile`.

Параметр `lpNumberOfBytesRead` должен содержать адрес памяти, в которую функция `ReadFile` поместит количество фактически прочитанных из файла байтов. Операционная система записывает по этому адресу ноль, прежде чем выполнить чтение данных из файла. При асинхронном вводе значение этого параметра может быть установлено в `NULL`. В этом случае для определения количества байтов, которые функция прочитала из файла, можно использовать функцию `GetOverlappedResult`.

Параметр `lpOverlapped` должен указывать на структуру типа `OVERLAPPED`. Причем в этой структуре должны быть установлены поля `Offset` и `OffsetHigh`. В поле `hEvent` может быть помещен как дескриптор события, так и значение `NULL`. В первом случае система устанавливает событие в сигнальное со-

стояние по завершении операции ввода. Во втором случае завершение операции ввода можно определить по дескриптору файла, который устанавливается в сигнальное состояние по завершении операции ввода, а при запуске новой операции ввода сигнальное состояние файла сбрасывается.

В листинге 27.3 приведена программа, которая выполняет асинхронное чтение данных из файла. В этой программе для сигнализации о завершении операции вывода используется дескриптор файла.

Листинг 27.3. Асинхронное чтение данных из файла с использованием дескриптора файла

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;     // дескриптор файла
  OVERLAPPED  ovl;   // структура управления асинхронным доступом к файлу

  // инициализируем структуру OVERLAPPED
  ovl.Offset = 0;        // младшая часть смещения равна 0
  ovl.OffsetHigh = 0;    // старшая часть смещения равна 0
  ovl.hEvent = 0;        // события нет

  // открываем файл для чтения
  hFile = CreateFile(
    "C:\\demo_file.dat",   // имя файла
    GENERIC_READ,          // чтение из файла
    FILE_SHARE_READ,       // совместный доступ к файлу
    NULL,                  // защиты нет
    OPEN_EXISTING,         // открываем существующий файл
    FILE_FLAG_OVERLAPPED,  // асинхронный ввод
    NULL                   // шаблона нет
  );
  // проверяем на успешное открытие
  if (hFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl
      << "The last error code: " << GetLastError() << endl;
```

```
  cout << "Press any key to finish.";
  cin.get();
  return 0;
}
// читаем данные из файла
for (;;)
{
  DWORD  dwBytesRead;
  DWORD  dwRet;
  int    n;

  // читаем одну запись
  if (!ReadFile(
      hFile,           // дескриптор файла
      &n,              // адрес буфера, куда читаем данные
      sizeof(n),       // количество читаемых байтов
      &dwBytesRead,    // количество прочитанных байтов
      &ovl             // чтение асинхронное
    ))
  {
    switch(dwRet = GetLastError())
    {
    case ERROR_IO_PENDING:
      cout << "Read file pending." << endl;
      break;
    case ERROR_HANDLE_EOF:
      cout << endl << "End of the file." << endl;
      // закрываем дескриптор файла
      CloseHandle(hFile);

      cout << "The file is read." << endl;
      return 1;
    default:
      cout << "Read file failed." << endl
        << "The last error code: " << dwRet << endl;
      return 0;
    }
  }
  // ждем, пока завершится асинхронная операция чтения
```

```
    WaitForSingleObject(hFile, INFINITE);
    // печатаем число
    cout << n << ' ';
    // увеличивает смещение в файле
    ovl.Offset += sizeof(n);
  }
}
```

Теперь немного изменим программу из листинга 27.4 таким образом, чтобы для сигнализации о завершении асинхронной операции ввода использовалось событие. Измененная программа приведена в листинге 27.4.

Листинг 27.4. Асинхронное чтение данных из файла с использованием события

```
#include <windows.h>
#include <iostream.h>


int main()
{
  HANDLE  hFile;     // дескриптор файла
  HANDLE  hEndRead;  // дескриптор события
  OVERLAPPED  ovl;   // структура управления асинхронным доступом к файлу

  // создаем события с автоматическим сбросом
  hEndRead = CreateEvent(NULL, FALSE, FALSE, NULL);
  if (hEndRead == NULL)
    return GetLastError();

  // инициализируем структуру OVERLAPPED
  ovl.Offset = 0;          // младшая часть смещения равна 0
  ovl.OffsetHigh = 0;      // старшая часть смещения равна 0
  ovl.hEvent = hEndRead;   // событие для оповещения завершения чтения

  // открываем файл для чтения
  hFile = CreateFile(
    "C:\\demo_file.dat",   // имя файла
    GENERIC_READ,          // чтение из файла
    FILE_SHARE_READ,       // совместный доступ к файлу
    NULL,                  // защиты нет
```

```
  OPEN_EXISTING,          // открываем существующий файл
  FILE_FLAG_OVERLAPPED,   // асинхронный ввод
  NULL                    // шаблона нет
);
// проверяем на успешное открытие
if (hFile == INVALID_HANDLE_VALUE)
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;

  CloseHandle(hEndRead);

  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
// читаем данные из файла
for (;;)
{
  DWORD  dwBytesRead;
  DWORD  dwRet;
  int    n;

  // читаем одну запись
  if (!ReadFile(
      hFile,           // дескриптор файла
      &n,              // адрес буфера, куда читаем данные
      sizeof(n),       // количество читаемых байтов
      &dwBytesRead,    // количество прочитанных байтов
      &ovl             // чтение асинхронное
    ))
  {
    switch(dwRet = GetLastError())
    {
    case ERROR_IO_PENDING:
      cout << "Read file pending." << endl;
      break;
    case ERROR_HANDLE_EOF:
```

```
        cout << endl << "End of the file." << endl;
        cout << "The file is read." << endl;

        // закрываем дескрипторы
        CloseHandle(hFile);
        CloseHandle(hEndRead);

        return 1;
      default:
        cout << "Read file failed." << endl
          << "The last error code: " << dwRet << endl;

        // закрываем дескрипторы
        CloseHandle(hFile);
        CloseHandle(hEndRead);

        return 0;
      }
    }
    // ждем, пока завершится асинхронная операция чтения
    WaitForSingleObject(hEndRead, INFINITE);
    // печатаем число
    cout << n << ' ';
    // увеличиваем смещение в файле
    ovl.Offset += sizeof(n);
  }

}
```

24.4. Блокирование файлов

На платформе Windows NT возможно выполнить асинхронное блокирование файла для монопольного доступа. Для этих целей предназначена функция `LockFileEx`, которая возвращает управление, не дожидаясь завершения операции блокирования файла. Эта функция имеет следующий прототип:

```
BOOL  LockFileEx(
  HANDLE  hFile,                    // дескриптор файла
  DWORD   dwFalgs,                  // опции блокирования
  DWORD   dwReserved,               // зарезервировано
```

```
  DWORD   nNumberOfBytesToLockLow,   // младшая часть количества байтов
  DWORD   nNumberOfBytesTiLockHigh,  // старшая часть количества байтов
  LPOVERLAPPED  lpOverlapped         // для асинхронной работы
);
```

При асинхронном блокировании файла эта функция возвращает ненулевое значение в том случае, если блокирование файла завершилось, до выхода из функции `LockFileEx`. Если же функция возвращает значение `FALSE`, то нужно проверить код последней ошибки путем вызова функции `GetLastError`. Если эта функция возвращает значение `ERROR_IO_PENDING`, то это значит, что операция блокирования файла еще не закончилась. Теперь перейдем к параметрам функции `LockFileEx`.

Параметр `hFile` должен содержать дескриптор файла. Причем этот файл должен быть открыт в режиме записи или чтения.

Параметр `dwFlags` задает режимы блокирования файла. В этом параметре может быть установлена любая комбинация из следующих двух флагов:

- `LOCKFILE_FAIL_IMMEDIATELY` — завершение функции в случае невозможности немедленной блокировки файла;
- `LOCKFILE_EXCLUSIVE_LOCK` — монопольный доступ к области файла.

Немного поясним значение флага `LOCKFILE_EXCLUSIVE_LOCK`. Если этот флаг установлен, то функция блокирует область файла для монопольного использования процессом. В противном случае область файла блокируется процессом только для записи, остальные же процессы могут читать данные из этой области файла.

Параметр `dwReserved` не используется и должен быть установлен в `0`.

В параметрах `dwFileOffsetLow` и `dwFileOffsetHigh` должны быть установлены младшая и старшая части соответственно смещения от начала файла в байтах.

Параметр `lpOverlapped` должен указывать на структуру типа `LPOVERLAPPED`. Поля `Offset` и `OffsetHigh` этой структуры должны содержать соответственно младшую и старшую части длины области файла, которая блокируется приложением. А в поле `hEvent` этой структуры может быть помещен как дескриптор события, так и значение `NULL`. В первом случае система устанавливает событие в сигнальное состояние по завершении операции блокирования. Во втором случае завершение операции блокирования можно определить по дескриптору файла, который устанавливается в несигнальное состояние перед исполнением операции блокирования и в сигнальное состояние по завершении этой операции.

Для асинхронной отмены блокировки области файла используется функция `UnlockFileEx`, которая имеет следующий прототип:

```
BOOL  UnlockFileEx(
  HANDLE  hFile,                   // дескриптор файла
```

```
    DWORD   dwReserved,                 // зарезервировано
    DWORD   nNumberOfBytesToLockLow,    // младшая часть количества байтов
    DWORD   nNumberOfBytesTiLockHigh,   // старшая часть количества байтов
    LPOVERLAPPED  lpOverlapped          // для асинхронной работы
  );
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`. Все параметры этой функции аналогичны параметрам функции `LockFileEx`.

В листинге 27.5 приведена программа, которая сначала асинхронно блокирует доступ к файлу, а затем разблокирует его.

Листинг 27.5. Асинхронное блокирование и разблокирование файла

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;
  DWORD  dwFileSize;
  DWORD  dwRet;
  OVERLAPPED  ovl;     // структура управления асинхронным доступом к файлу

  // инициализируем структуру OVERLAPPED
  ovl.Offset = 0;      // младшая часть смещения равна 0
  ovl.OffsetHigh = 0;  // старшая часть смещения равна 0
  ovl.hEvent = 0;      // события нет

  // открываем файл для записи
  hFile = CreateFile(
    "C:\\demo_file.dat",    // имя файла
    GENERIC_WRITE,          // запись в файл
    FILE_SHARE_WRITE,       // совместный доступ к файлу
    NULL,                   // защиты нет
    OPEN_EXISTING,          // открываем существующий файл
    FILE_FLAG_OVERLAPPED,   // асинхронный доступ к файлу
    NULL                    // шаблона нет
  );
  // проверяем на успешное открытие
```

```
if (hFile == INVALID_HANDLE_VALUE)
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish.";
  cin.get();
  return 0;
}
// определяем размер файла
dwFileSize = GetFileSize(hFile, NULL);
if (dwFileSize == -1)
{
  cerr << "Get file size failed." << endl
    << "The last error code: " << GetLastError() << endl;
  CloseHandle(hFile);
  cout << "Press any key to finish.";
  cin.get();
  return (0);
}
// блокируем файл
if (!LockFileEx(
    hFile,           // дескриптор файла
    LOCKFILE_EXCLUSIVE_LOCK, // монопольный доступ к файлу
    0,               // не используется
    0, dwFileSize,   // длина области
    &ovl))           // асинхронная блокировка
{
  dwRet = GetLastError();
  if (dwRet == ERROR_IO_PENDING)
    cout << "Lock file is pending." << endl;
  else
  {
    cout << "Lock file failed." << endl
      << "The last error code: " << dwRet << endl;
    CloseHandle(hFile);
    return 0;
  }
}
// ждем, пока завершится асинхронная операция записи
```

```
  WaitForSingleObject(hFile, INFINITE);

  cout << "Now the file is locked." << endl
    << "Press any key to continue." << endl;
  cin.get();
  // разблокируем файл
  if (!UnlockFileEx(hFile, 0, 0, dwFileSize, &ovl))
  {
    cerr << "Unlock file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hFile);
    cout << "Press any key to finish.";
    cin.get();
    return 0;
  }
  cout << "Now the file is unlocked." << endl
    << "Press any key to continue." << endl;
  cin.get();
  // закрываем дескриптор файла
  CloseHandle(hFile);

  return 0;
}
```

Теперь немного изменим программу из листинга 27.5 таким образом, чтобы для сигнализации о завершении асинхронной операции ввода использовалось событие. Измененная программа приведена в листинге 27.6.

Листинг 27.6. Асинхронное блокирование и разблокирование файла с использованием события

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;     // дескриптор файла
  HANDLE  hEndLock;  // дескриптор события
  DWORD  dwFileSize;
  DWORD  dwRet;
```

```
OVERLAPPED  ovl;   // структура управления асинхронным доступом к файлу

// создаем события с автоматическим сбросом
hEndLock = CreateEvent(NULL, FALSE, FALSE, NULL);
if (hEndLock == NULL)
  return GetLastError();

// инициализируем структуру OVERLAPPED
ovl.Offset = 0;          // младшая часть смещения равна 0
ovl.OffsetHigh = 0;      // старшая часть смещения равна 0
ovl.hEvent = hEndLock;   // событие для оповещения завершения блокирования

// открываем файл для записи
hFile = CreateFile(
  "C:\\demo_file.dat",   // имя файла
  GENERIC_WRITE,         // запись в файл
  0,                     // монопольный доступ к файлу
  NULL,                  // защиты нет
  OPEN_EXISTING,         // открываем существующий файл
  FILE_FLAG_OVERLAPPED,  // асинхронный доступ к файлу
  NULL                   // шаблона нет
);
// проверяем на успешное открытие
if (hFile == INVALID_HANDLE_VALUE)
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;

  CloseHandle(hFile);

  cout << "Press any key to finish.";
  cin.get();

  return 0;
}

// определяем размер файла
dwFileSize = GetFileSize(hFile, NULL);
if (dwFileSize == -1)
```

```
{
  cerr << "Get file size failed." << endl
    << "The last error code: " << GetLastError() << endl;

  CloseHandle(hFile);

  cout << "Press any key to finish.";
  cin.get();

  return 0;
}

// блокируем файл
if (!LockFileEx(
    hFile,          // дескриптор файла
    LOCKFILE_EXCLUSIVE_LOCK,   // монопольный доступ к файлу
    0,              // не используется
    0, dwFileSize, // длина области
    &ovl))          // асинхронная блокировка
{
  dwRet = GetLastError();
  if (dwRet == ERROR_IO_PENDING)
    cout << "The lock file is pending." << endl;
  else
  {
    cout << "Lock file failed." << endl
      << "The last error code: " << dwRet << endl;

    CloseHandle(hFile);
    CloseHandle(hEndLock);

    return 0;
  }
}

// ждем, пока завершится асинхронная операция записи
WaitForSingleObject(hEndLock, INFINITE);

cout << "Now the file is locked." << endl
```

```
    << "Press any key to continue." << endl;
  cin.get();

  // разблокируем файл
  if (!UnlockFileEx(hFile, 0, 0, dwFileSize, &ovl))
  {
    cerr << "Unlock file failed." << endl
      << "The last error code: " << GetLastError() << endl;

    CloseHandle(hFile);
    CloseHandle(hEndLock);

    cout << "Press any key to finish.";
    cin.get();

    return 0;
  }
  cout << "Now the file is unlocked." << endl
    << "Press any key to continue." << endl;
  cin.get();

  // закрываем дескрипторы
  CloseHandle(hFile);
  CloseHandle(hEndLock);
  return 0;
}
```

27.5. Определение состояния асинхронной операции ввода-вывода

Состояние, в котором находится асинхронная операция ввода-вывода, можно определить посредством функции `GetOverlappedResult`, которая имеет следующий прототип:

```
BOOL  GetOverlappedResult(
  HANDLE   hFile,              // дескриптор файла
  LPOVERLAPPED  lpOverlapped,  // для асинхронной работы
  LPDWORD  lpNumberOfBytesTransferred,  // количество переданных байт
  BOOL     bWait               // режим ожидания
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — FALSE.

Параметр hFile должен содержать дескриптор файла, для доступа к которому используется асинхронная операция ввода-вывода посредством вызова функций ReadFile и WriteFile. Если дескриптор описывает именованный канал, то посредством функции GetOverlappedResult можно также определить состояние операций ConnectNamedPipe и TransactNamedPipe.

Параметр lpOverlapped должен содержать адрес структуры типа OVERLAPPED, которая используется в асинхронной операции доступа к файлу.

Параметр lpNumberOfBytesTransferred должен указывать на переменную типа DWORD, в которую функция GetOverlappedResult поместит фактическое количество байт, переданных асинхронной операцией чтения или записи. Для функций ConnectNamedPipe и TransactNamedPipe это значение не определено.

В параметре bWait должно быть установлено одно из двух значений TRUE или FALSE, которое указывает на режим ожидания функции GetOverlappedResult. Если значение этого параметра равно FALSE, то функция GetOverlappedResult сразу после определения состояния асинхронной передачи данных вернет значение FALSE. В этом случае функция определения последней ошибки GetLastError вернет значение ERROR_IO_INCOMPLETE. Если же значение этого параметра равно TRUE, то функция GetOverlappedResult не вернет управление до тех пор, пока не завершится асинхронная операция передачи данных. При этом функция GetOverlappedResult определяет завершение асинхронной операции передачи данных посредством ожидания перехода в сигнальное состояние события, дескриптор которого задан в структуре типа OVERLAPPED.

В листинге 27.7 приведена программа, в которой функция GetOverlappedResult используется для определения состояния асинхронной операции записи данных в файл.

Листинг 27.7. Определение состояния асинхронной операции записи данных

```
#define _WIN32_WINNT 0x0400
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;       // дескриптор файла
  HANDLE  hEndWrite;   // дескриптор события
```

```
OVERLAPPED  ovl;      // структура управления асинхронным доступом к файлу

// создаем события с автоматическим сбросом
hEndWrite = CreateEvent(NULL, FALSE, FALSE, NULL);
if (hEndWrite == NULL)
  return GetLastError();

  // инициализируем структуру OVERLAPPED
ovl.Offset = 0;          // младшая часть смещения равна 0
ovl.OffsetHigh = 0;      // старшая часть смещения равна 0
ovl.hEvent = hEndWrite;  // событие для оповещения завершения записи

// создаем файл для записи данных
hFile = CreateFile(
  "C:\\demo_file.dat",   // имя файла
  GENERIC_WRITE,         // запись в файл
  FILE_SHARE_WRITE,      // совместный доступ к файлу
  NULL,                  // защиты нет
  OPEN_ALWAYS,           // открываем или создаем новый файл
  FILE_FLAG_OVERLAPPED,  // асинхронный доступ к файлу
  NULL                   // шаблона нет
);
// проверяем на успешное создание
if (hFile == INVALID_HANDLE_VALUE)
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;

  CloseHandle(hEndWrite);

  cout << "Press any key to finish.";
  cin.get();

  return 0;
}

// пишем данные в файл
for (int i = 0; i < 10; ++i)
{
```

```
DWORD  dwBytesWrite;
DWORD  dwNumberOfBytesTransferred;
DWORD  dwRet;

if (!WriteFile(
    hFile,          // дескриптор файла
    &i,             // адрес буфера, откуда идет запись
    sizeof(i),      // количество записываемых байтов
    &dwBytesWrite, // количество записанных байтов
    &ovl            // запись асинхронная
  ))
{
  dwRet = GetLastError();
  if (dwRet == ERROR_IO_PENDING)
    cout << "Write file pending." << endl;
  else
  {
    cout << "Write file failed." << endl
      << "The last error code: " << dwRet << endl;

    return 0;
  }
}

// проверяем состояние асинхронной операции записи
if (!GetOverlappedResult(
    hFile,        // дескриптор файла
    &ovl,         // адрес структуры для асинхронной работы
    &dwNumberOfBytesTransferred,  // количество переданных байтов
    FALSE         // не ждать завершения операции записи
  ))
{
  cout << "Get overlapped result failed." << endl
    << "The last error code: " << GetLastError() << endl;

  return 0;
}
else
  cout << "Number of bytes transferred: "
```

```
            << dwNumberOfBytesTransferred << endl;

    // ждем завершения асинхронной операции записи
    WaitForSingleObject(hEndWrite, INFINITE);
    // увеличиваем смещение в файле
    ovl.Offset += sizeof(i);
  }
  // закрываем дескрипторы
  CloseHandle(hFile);
  CloseHandle(hEndWrite);

  cout << "The file is written." << endl;

  return 0;
}
```

27.6. Отмена асинхронной операции ввода-вывода

Отменить асинхронную операцию передачи данных, которая еще не завершилась, можно посредством функции `CancelIo`, которая имеет следующий прототип:

```
BOOL  CancelIo(
  HANDLE  hFile    // дескриптор файла
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. Единственный параметр `hFile` этой функции должен содержать дескриптор файла, для которого отменяются все асинхронные операции ввода-вывода. Операция, отменяющаяся функцией `CancelIo`, немедленно прекращается, а сама функция ввода-вывода, которая инициировала эту операцию, завершается с кодом `ERROR_OPERATION_ABORTED`.

Отметим, что функция `CancelIo` отменяет только те асинхронные операции ввода-вывода для заданного файла, которые были запущены в потоке, вызвавшем эту функцию.

В листинге 27.8 приведена программа, в которой показано, как использовать функцию `CancelIo` для отмены незавершенных асинхронных операций записи данных в файл.

Листинг 27.8. Отмена незавершенных асинхронных операций записи данных

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;     // дескриптор файла
  HANDLE  hEndWrite; // дескриптор события
  OVERLAPPED  ovl;   // структура управления асинхронным доступом к файлу

  // создаем события с автоматическим сбросом
  hEndWrite = CreateEvent(NULL, FALSE, FALSE, NULL);
  if (hEndWrite == NULL)
    return GetLastError();

  // инициализируем структуру OVERLAPPED
  ovl.Offset = 0;          // младшая часть смещения равна 0
  ovl.OffsetHigh = 0;      // старшая часть смещения равна 0
  ovl.hEvent = hEndWrite;  // событие для оповещения завершения записи

  // создаем файл для записи данных
  hFile = CreateFile(
    "C:\\demo_file.dat",   // имя файла
    GENERIC_WRITE,         // запись в файл
    FILE_SHARE_WRITE,      // совместный доступ к файлу
    NULL,                  // защиты нет
    OPEN_ALWAYS,           // открываем или создаем новый файл
    FILE_FLAG_OVERLAPPED,  // асинхронный доступ к файлу
    NULL                   // шаблона нет
  );
  // проверяем на успешное создание
  if (hFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl
```

```
    << "The last error code: " << GetLastError() << endl;

  CloseHandle(hEndWrite);

  cout << "Press any key to finish.";
  cin.get();

  return 0;
}

// пишем данные в файл
for (int i = 0; i < 10; ++i)
{
  DWORD  dwBytesWrite;

  if (!WriteFile(
      hFile,           // дескриптор файла
      &i,              // адрес буфера, откуда идет запись
      sizeof(i),       // количество записываемых байтов
      &dwBytesWrite,   // количество записанных байтов
      &ovl             // запись асинхронная
    ))
  {
    cout << "Write file failed." << endl
      << "The last error code: " << GetLastError() << endl;

    return 0;
  }
  // увеличиваем смещение в файле
  ovl.Offset += sizeof(i);
}

// отменяем незавершенные асинхронные операции записи
if (!CancelIo(hFile))
{
  cout << "Concel io failed." << endl
    << "The last error code: " << GetLastError() << endl;

  return 0;
```

```
  }
  // закрываем дескрипторы
  CloseHandle(hFile);
  CloseHandle(hEndWrite);

  cout << "The file is written." << endl;

  return 0;
}
```

Проверить завершение асинхронной операции передачи данных можно при помощи макрокоманды `HasOverlappedIoCompleted`, которая определена следующим образом:

```
#define  HasOverlappedIoCompleted(lpOverlapped)
  ((lpOverlapped)->Internal != STATUS_PENDING)
```

и может рассматриваться как функция

```
BOOL  HasOverlappedIoCompleted(
  LPOVERLAPPED  lpOverlapped
);
```

В случае завершения асинхронной операции передачи данных эта команда возвращает значение `TRUE`, иначе — `FALSE`. Единственным параметром макрокоманды `HasOverlappedIoCompleted` является указатель на структуру типа `OVERLAPPED`, которая используется для управления асинхронной операцией передачи данных. Отметим, что до вызова этой макрокоманды нужно убедиться, что асинхронная операция передачи данных уже стартовала.

В листинге 27.9 приведена программа, в которой для определения завершения асинхронной операции записи в файл используется макрокоманда `HasOverlappedIoCompleted`.

Листинг 27.9. Определение завершения асинхронной операции записи данных

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;       // дескриптор файла
  HANDLE  hEndWrite;   // дескриптор события
```

```
OVERLAPPED  ovl;      // структура управления асинхронным доступом к файлу

// создаем события с автоматическим сбросом
hEndWrite = CreateEvent(NULL, FALSE, FALSE, NULL);
if (hEndWrite == NULL)
  return GetLastError();

// инициализируем структуру OVERLAPPED
ovl.Offset = 0;          // младшая часть смещения равна 0
ovl.OffsetHigh = 0;      // старшая часть смещения равна 0
ovl.hEvent = hEndWrite;  // событие для оповещения завершения записи

// создаем файл для записи данных
hFile = CreateFile(
  "C:\\demo_file.dat",   // имя файла
  GENERIC_WRITE,         // запись в файл
  FILE_SHARE_WRITE,      // совместный доступ к файлу
  NULL,                  // защиты нет
  OPEN_ALWAYS,           // открываем или создаем новый файл
  FILE_FLAG_OVERLAPPED,  // асинхронный доступ к файлу
  NULL                   // шаблона нет
);
// проверяем на успешное создание
if (hFile == INVALID_HANDLE_VALUE)
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;

  CloseHandle(hEndWrite);

  cout << "Press any key to finish.";
  cin.get();

  return 0;
}

// пишем данные в файл
for (int i = 0; i < 10; ++i)
```

```
  {
    DWORD  dwBytesWrite;

    if (!WriteFile(
        hFile,          // дескриптор файла
        &i,             // адрес буфера, откуда идет запись
        sizeof(i),      // количество записываемых байтов
        &dwBytesWrite, // количество записанных байтов
        &ovl            // запись асинхронная
      ))
    {
      cout << "Write file failed." << endl
        << "The last error code: " << GetLastError() << endl;

      return 0;
    }

    // проверяем завершение асинхронной операции записи
    if (HasOverlappedIoCompleted(&ovl))
      cout << "Write file is completed." << endl;
    else
      cout << "Write file is not completed." << endl;

    // ждем завершения асинхронной операции записи
    WaitForSingleObject(hEndWrite, INFINITE);
    // увеличиваем смещение в файле
    ovl.Offset += sizeof(i);
  }
  // закрываем дескрипторы
  CloseHandle(hFile);
  CloseHandle(hEndWrite);

  cout << "The file is written." << endl;

  return 0;
}
```

27.7. Процедуры завершения ввода-вывода

Возможен другой подход к оповещению потока о завершении асинхронной операции записи или чтения данных из файла. В этом случае операционная система не устанавливает событие, дескриптор которого задан в структуре типа `OVERLAPPED`, а, по завершении асинхронной операции ввода-вывода, вызывает функцию, которая называется *процедурой завершения ввода-вывода* (*FileIOCompletionRoutine*). Такой подход к оповещению завершения асинхронных операций ввода-вывода используется в функциях `WriteFileEx` и `ReadFileEx`. При этом отметим, что процедура завершения ввода-вывода вызывается только в том случае, если поток находится в настороженном состоянии. В противном случае процедура завершения ввода-вывода включается в очередь асинхронных процедур потока. Кроме того, процедуры завершения ввода-вывода могут вызываться в порядке отличном от порядка завершения асинхронных операций ввода-вывода.

Процедура завершения ввода-вывода определяется программистом, но при этом она должна иметь следующий прототип:

```
VOID  CALLBACK  процедура_завершения_ввода_вывода(
  DWORD  dwErrorCode,          // код завершения операции ввода-вывода
  DWORD  dwNumberOfBytesTransferred,   // количество переданных байтов
  LPOVERLAPPED  lpOverlapped   // информация о вводе-выводе
);
```

Параметры процедуры ввода-вывода имеют следующее назначение.

В параметре `dwErrorCode` процедура завершения ввода-вывода получает код завершения асинхронной операции ввода-вывода. Этот код может принимать одно из двух значений:

- `0` — ввод-вывод завершился успешно;
- `ERROR_HANDLE_EOF` — операция `ReadFileEx` достигла конца файла.

В параметре `dwNumberOfBytesTransferred` процедура завершения ввода-вывода поучает количество переданных байтов в асинхронной операции ввода-вывода. Если последняя завершилась ошибкой, то значение этого параметра равно нулю.

В параметре `lpOverlapped` процедура завершения ввода-вывода получает адрес структуры типа `OVERLAPPED`, которая использовалась в асинхронной операции ввода-вывода.

27.8. Асинхронная запись данных с процедурами завершения

Для асинхронной записи данных в файл, используя процедуры завершения ввода-вывода, предназначена функция `WriteFileEx`, которая имеет следующий прототип:

```
BOOL  WriteFileEx(
  HANDLE   hFile,                 // дескриптор файла
  LPCVOID  lpBuffer,              // буфер данных
  DWORD    nNumberOfBytesToWrite, // количество записываемых байтов
  LPOVERLAPPED  lpOverlapped,     // асинхронная запись
  // адрес процедуры завершения
  LPOVERLAPPED_COMPLETION_ROUTINE  lpCompletionRoutine
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`.

В параметре `hFile` должен быть установлен дескриптор файла, в который записываются данные. При этом файл должен быть открыт в режиме асинхронной записи данных, т. е. должен быть установлен флаг `FILE_FLAG_OVERLAPPED`.

Параметр `lpBuffer` должен содержать адрес буфера, в который будут записываться данные.

Параметр `nNumberOfBytesToWrite` должен содержать количество байтов, записываемых в файл.

Параметр `lpOverlapped` должен указывать на структуру типа `OVERLAPPED`, которая используется при асинхронной записи в файл.

Параметр `lpCompletionRoutine` должен указывать на процедуру завершения асинхронной операции ввода-вывода.

В листинге 27.10 приведена программа, в которой показано, как можно записать данные в файл, используя функцию `WriteFileEx`.

Листинг 27.10. Асинхронная запись данных в файл

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include <iostream.h>

// процедура завершения ввода-вывода
```

```
VOID  CALLBACK completion_routine(
  DWORD  dwErrorCode,                // код возврата
  DWORD  dwNumberOfBytesTransferred, // количество переданных байтов
  LPOVERLAPPED  lpOverlapped         // асинхронная передача данных
    )
{
  cout << "Completion routine parameters: " << endl
    << "\tErrorCode: " << dwErrorCode << endl
    << "\tNumber of bytes transferred: " << dwNumberOfBytesTransferred <<
endl
    << "\tOffsets: " << (*lpOverlapped).OffsetHigh << ' '
    << (*lpOverlapped).Offset << endl;
}

// главная программа
int main()
{
  HANDLE  hFile;    // дескриптор файла
  OVERLAPPED  ovl;  // структура управления асинхронным доступом к файлу

  // инициализируем структуру OVERLAPPED
  ovl.Offset = 0;      // младшая часть смещения равна 0
  ovl.OffsetHigh = 0;  // старшая часть смещения равна 0

  // создаем файл для записи данных
  hFile = CreateFile(
    "C:\\demo_file.dat",   // имя файла
    GENERIC_WRITE,         // запись в файл
    FILE_SHARE_WRITE,      // совместный доступ к файлу
    NULL,                  // защиты нет
    OPEN_ALWAYS,           // открываем или создаем новый файл
    FILE_FLAG_OVERLAPPED,  // асинхронный доступ к файлу
    NULL                   // шаблона нет
  );
  // проверяем на успешное создание
  if (hFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl
```

```
    << "The last error code: " << GetLastError() << endl;

  cout << "Press any key to finish.";
  cin.get();

  return 0;
}

// пишем данные в файл
for (int i = 0; i < 10; ++i)
{
  DWORD  dwRet;

  if (!WriteFileEx(
      hFile,       // дескриптор файла
      &i,          // адрес буфера, откуда идет запись
      sizeof(i),   // количество записываемых байтов
      &ovl,        // запись асинхронная
      completion_routine // процедура завершения
    ))
  {
    dwRet = GetLastError();
    if (dwRet == ERROR_IO_PENDING)
      cout << "Write file pending." << endl;
    else
    {
      cout << "Write file failed." << endl
        << "The last error code: " << dwRet << endl;

      return 0;
    }
  }
  // ждем, пока сработает асинхронная процедура
  // завершения операции вывода
  SleepEx(INFINITE, TRUE);
  // увеличивает смещение в файле
  ovl.Offset += sizeof(i);
}
```

```
  // закрываем дескриптор файла
  CloseHandle(hFile);

  cout << "The file is written." << endl;

  return 0;
}
```

27.9. Асинхронное чтение данных с процедурами завершения

Для асинхронного чтения данных из файла, используя процедуры завершения ввода-вывода, предназначена функция `ReadFileEx`, которая имеет следующий прототип:

```
BOOL  ReadFileEx(
  HANDLE   hFile,                 // дескриптор файла
  LPCVOID  lpBuffer,              // буфер данных
  DWORD    nNumberOfBytesToRead,  // количество читаемых байтов
  LPOVERLAPPED  lpOverlapped,     // асинхронная запись
  // адрес процедуры завершения
  LPOVERLAPPED_COMPLETION_ROUTINE  lpCompletionRoutine
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`.

Параметры функции `ReadFileEx` имеют следующее назначение.

В параметре `hFile` должен быть установлен дескриптор файла, из которого читаются данные. При этом файл должен быть открыт в режиме асинхронного чтения данных, т. е. должен быть установлен флаг `FILE_FLAG_OVERLAPPED`.

Параметр `lpBuffer` должен содержать адрес буфера, из которого будут читаться данные.

Параметр `nNumberOfBytesToRead` должен содержать количество байтов, читаемых из файла.

Параметр `lpOverlapped` должен указывать на структуру типа `OVERLAPPED`, которая используется при асинхронном чтении из файла.

Параметр `lpCompletionRoutine` должен указывать на процедуру завершения асинхронной операции ввода-вывода.

В листинге 27.11 приведена программа, в которой показано, как можно читать данные из файла, используя функцию `ReadFileEx`.

Листинг 27.11. Асинхронное чтение данных из файла

```
#include <windows.h>
#include <iostream.h>

// процедура завершения ввода-вывода
VOID  CALLBACK completion_routine(
    DWORD  dwErrorCode,                // код возврата
    DWORD  dwNumberOfBytesTransferred, // количество переданных байтов
    LPOVERLAPPED  lpOverlapped         // асинхронная передача данных
    )
{
  cout << "Completion routine parameters: " << endl
    << "\tErrorCode: " << dwErrorCode << endl
    << "\tNumber of bytes transferred: " << dwNumberOfBytesTransferred << endl
    << "\tOffsets: " << (*lpOverlapped).OffsetHigh << ' '
    << (*lpOverlapped).Offset << endl;
}

// главная программа
int main()
{
  HANDLE  hFile;     // дескриптор файла
  OVERLAPPED  ovl;   // структура управления асинхронным доступом к файлу

  // инициализируем структуру OVERLAPPED
  ovl.Offset = 0;      // младшая часть смещения равна 0
  ovl.OffsetHigh = 0;  // старшая часть смещения равна 0

  // открываем файл для чтения
  hFile = CreateFile(
    "C:\\demo_file.dat",   // имя файла
    GENERIC_READ,          // чтение из файла
    FILE_SHARE_READ,       // совместный доступ к файлу
    NULL,                  // защиты нет
    OPEN_EXISTING,         // открываем существующий файл
    FILE_FLAG_OVERLAPPED,  // асинхронный ввод
    NULL                   // шаблона нет
  );
```

```
// проверяем на успешное открытие
if (hFile == INVALID_HANDLE_VALUE)
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;

  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
// читаем данные из файла
for (;;)
{
  DWORD  dwRet;
  int    n;

  // читаем одну запись
  if (!ReadFileEx(
      hFile,       // дескриптор файла
      &n,          // адрес буфера, куда читаем данные
      sizeof(n),   // количество читаемых байтов
      &ovl,        // чтение асинхронное
      completion_routine   // процедура завершения чтения
    ))
  {
    switch(dwRet = GetLastError())
    {
    case ERROR_IO_PENDING:
      cout << "Read file pending." << endl;
      break;
    case ERROR_HANDLE_EOF:
      cout << endl << "End of the file." << endl;
      cout << "The file is read." << endl;
      // закрываем дескриптор файла
      CloseHandle(hFile);

      return 0;
    default:
```

```
        cout << "Read file failed." << endl
          << "The last error code: " << dwRet << endl;

        // закрываем дескриптор файла
        CloseHandle(hFile);

        return 0;
      }
    }
    // ждем, пока сработает асинхронная процедура завершения чтения
    SleepEx(INFINITE, TRUE);
    // печатаем число
    cout << n << endl;
    // увеличивает смещение в файле
    ovl.Offset += sizeof(n);
  }
}
```

Глава 28

Порты завершения

28.1. Концепция порта завершения

Порт завершения ввода-вывода — это, фактически, объект синхронизации, который оповещает параллельно работающие потоки о завершении асинхронных операций доступа к файлам или именованным каналам. В дальнейшем мы будем говорить только о файлах, подразумевая, что все сказанное справедливо и для именованных каналов.

Для работы порта завершения ввода-вывода к нему необходимо подключить файлы, которые должны быть открыты в режиме асинхронного доступа. Причем для каждого файла устанавливается свой номер, который в этом случае называется *ключом завершения*. При завершении асинхронной операции доступа к файлу операционная система посылает в порт завершения пакет, который содержит информацию о завершившейся асинхронной операции ввода-вывода, содержащую и ключ файла, для которого завершилась эта операция. Порт содержит очередь пакетов, оповещающих о завершении асинхронных операций ввода-вывода. Обслуживается эта очередь по алгоритму FIFO (first in — first out), т. е. вошедший первым — выходит первым. Для создания портов завершения ввода-вывода и подключения к нему файлов в операционных системах Windows используется функция `CreateIoCompletionPort`. Порт завершения ввода-вывода удаляется, когда закрывается его дескриптор.

Поток может узнать о том, что завершилась асинхронная операция ввода-вывода, вызвав функцию `GetQueuedCompletionStatus`, в которой указывается дескриптор порта. Если очередь пакетов порта завершения не пуста, то поток получает пакет из этой очереди. В противном случае поток блокируется. Каждый порт содержит очередь заблокированных потоков, которая обслуживается по алгоритму LIFO (lst in — first out), т. е. вошедший последним — выходит первым. При этом отметим, что порт завершения может одновременно обслуживать только заданное количество потоков, которое

определяется одним из параметров функции CreateIoCompletionPort. Остальные потоки, которые превысили этот предел, блокируются системой, до тех пор, пока количество обслуживаемых потоков не станет меньше заданного предела.

Другой особенностью порта завершения является то, что он может получать пакеты не только от системы, по завершении асинхронной операции ввода-вывода, но также и от приложения. Для посылки пакетов в порт завершения предназначена функция PostQueuedCompletionStatus. Этот прием может использоваться для оповещения потоков о внешних событиях.

В заключение этого раздела отметим, что работа с портами завершения ввода-вывода поддерживается только в операционных системах Windows NT/2000/XP.

28.2. Создание порта завершения

Для создания порта завершения ввода-вывода и подключения файлов к этому порту используется функция CreateIoCompletionPort, которая имеет следующий прототип:

```
HANDLE  CreateIoCompletionPort(
  HANDLE  hFile,                          // дескриптор файла
  HANDLE  hExistingCompletionPort,        // дескриптор порта завершения
  ULONG   *ulCompletionKey,               // ключ завершения
  DWORD   dwNumberOfConcurrentThreads  // количество параллельных потоков
);
```

В случае успешного завершения эта функция возвращает дескриптор созданного порта завершения ввода-вывода, а в случае неудачи — NULL.

В параметре hFile должен содержать должен быть установлен дескриптор файла, который подключается к порту завершения ввода-вывода. Причем этот файл должен быть открыт в режиме асинхронного ввода-вывода. То есть должен быть установлен флаг FILE_FLAG_OVERLAPPED.

Если создается новый порт, то в параметре hFile можно установить значение INVALID_HANDLE_VALUE. Тогда будет создан порт без подключенного к нему файла. В этом случае в параметре hExistingCompletionPort должно быть установлено значение NULL. Значение же параметра ulCompletionKey не важно, т. к. не используется функцией.

В параметре hExistingCompletionPort должен быть установлен дескриптор порта завершения, к которому подключается файл. Как уже было сказано, если создается новый порт завершения, то значение этого параметра должно быть установлено в NULL.

В параметре ulCompletionKey должен быть задан адрес числа без знака, которое будет обозначать номер файла. Это число называется ключом завер-

шения и используется для идентификации файла, для которого завершилась асинхронная операция ввода-вывода.

В параметре `dwNumberOfConcurrentThreads` задается максимальное количество потоков, которые могут одновременно подключиться к порту завершения ввода-вывода. Если это значение равно нулю, то максимальное количество потоков равно количеству процессоров в системе. Каждый новый вызов функции `CreateIoCompletionPort` задает новое максимальное количество потоков.

Пример использования функции `CreateIoCompletionPort` будет приведен в *разд. 28.4*, после изучения остальных функций, предназначенных для работы с портом завершения.

28.3. Получение пакета из порта завершения

Для получения пакета из порта завершения ввода-вывода используется функция `GetQueuedCompletionStatus`, которая имеет следующий прототип:

```
BOOL GetQueuedCompletionStatus(
  HANDLE   hCompletionPort,      // дескриптор порта завершения
  LPDWORD  lpNumberOfBytes,      // количество переданных байтов
  ULONG    *ulCompletionKey,     // ключ завершения
  LPOVERLAPPED  *lpOverlapped,   // адрес структур управления
                                 // асинхронным доступом
  DWORD    dwMilliseconds        // интервал ожидания
);
```

В случае успешного завершения функция `GetQueuedCompletionStatus` возвращает ненулевое значение, а в случае неудачи — `FALSE`. Параметры этой функции имеют следующее назначение.

В параметре `hCompletionPort` должен быть установлен дескриптор порта, к которому поток обращается за пакетом.

В параметре `lpNumberOfBytes` должен быть установлен адрес переменной, в которую в случае успешного завершения функция поместит количество переданных байт.

В параметре `ulCompletionKey` должен быть задан адрес переменной типа `unsigned long`, в которую функция `GetQueuedCompletionStatus` поместит ключ завершения, т. е. номер файла, для которого завершилась асинхронная операции ввода-вывода. Посылку пакетов в порт завершения ввода-вывода могут инициировать следующие функции: `ConnectNamedPipe`, `DeviceIoControl`, `LockFileEx`, `ReadDirectoryChangesW`, `ReadFile`, `TransactNamedPipe`,

WaitCommEvent, WriteFile. При этом отметим, что приложение может отменить посылку пакета в порт завершения ввода-вывода, если установит младший бит в дескрипторе события, который задан в структуре OVERLAPPED, обрабатываемой асинхронной операцией ввода-вывода.

В параметре lpOverlapped должен быть задан адрес на переменную, в которую функция GetQueuedCompletionStatus поместит адрес структуры типа OVERLAPPED, для завершенной асинхронной операции ввода-вывода.

Параметр dwMilliseconds должен содержать количество миллисекунд, которые определяют время ожидания функцией GetQueuedCompletionStatus завершение асинхронной операции ввода-вывода. Если в течение этого времени в очередь не поставлен ни один пакет, то функция завершается неудачей, а по адресу, заданному параметром lpOverlapped, будет записано значение NULL.

Пример использования функции GetQueuedCompletionStatus будет приведен в следующем разделе, после изучения функции PostQueuedCompletionStatus.

28.4. Посылка пакета в порт завершения

Для посылки приложением пакета в порт завершения ввода-вывода используется функция PostQueuedCompletionStatus, которая имеет следующий прототип:

```
BOOL PostQueuedCompletionStatus(
  HANDLE  hCompletionPort,              // дескриптор порта завершения
  WORD    lpNumberOfBytesTransferred,   // количество переданных байтов
  ULONG   *ulCompletionKey,             // ключ завершения
  LPOVERLAPPED  *lpOverlapped,          // асинхронный доступ
);
```

В случае успешного завершения функция PostQueuedCompletionStatus возвращает ненулевое значение, а в случае неудачи — FALSE.

В параметре hCompletionPort должен быть установлен дескриптор порта, в который приложение посылает пакет. Остальные параметры могут принимать произвольные значения. Эти значения будут переданы через соответствующие выходные параметры функции GetQueuedCompletionStatus в поток, который получит пакет, вызвав эту функцию.

В листинге 28.1 приведена программа, в которой поток main посылает пакеты в порт завершения ввода-вывода, используя функцию PostQueuedCompletionStatus, а поток thread получает пакеты из этого порта, используя функцию GetQueuedCompletionStatus.

Поясним работу этой программы чуть подробнее. Программа состоит из двух потоков, работающих с портом завершения ввода-вывода. Поток main

создает порт завершения ввода-вывода и подключает к нему файл, в который записываются данные. Поток `thread` получает пакеты о завершении асинхронных операций записи данных в этот файл. Отметим следующий момент, поток `thread` завершает свою работу после получения пакета от потока `main`. Для этого считаем, что пакет с ключом файла, равным нулю, оповещает о завершении операции записи данных в файл.

Листинг 28.1. Работа с портом завершения ввода-вывода

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include <iostream.h>

HANDLE  hCompletionPort;   // порт завершения

DWORD WINAPI thread(LPVOID)
{
  int  i = 0;      // количество полученных пакетов
  DWORD  dwNumberOfBytes;      // количество переданных байтов
  ULONG    ulCompletionKey;    // ключ файла
  LPOVERLAPPED  lpOverlapped;  // указатель на структуру типа OVERLAPPED

  cout << "The thread is started." << endl;

  // подключаем поток к порту
  while (GetQueuedCompletionStatus(
    hCompletionPort,   // дескриптор порта завершения
    &dwNumberOfBytes,  // количество переданных байтов
    &ulCompletionKey,  // ключ файла
    &lpOverlapped,     // указатель на структуру типа OVERLAPPED
    INFINITE))         // бесконечное ожидание
//  {---тело цикла---
    // проверяем пакет на завершение вывода
    if (ulCompletionKey == 0)
    {
      cout << endl << "The thread is finished." << endl;
      break;
    }
    else
```

```
      cout << "\tPacket: " << ++i << endl
        << "Number of bytes: " << dwNumberOfBytes << endl
        << "Completion key: " << ulCompletionKey << endl;
//  }---конец тела цикла---

  return 0;
}

int main()
{
  HANDLE  hFile;     // дескриптор файла
  OVERLAPPED  ovl;   // структура управления асинхронным доступом к файлу
  ULONG    ulKey;    // ключ файла
  HANDLE  hThread;   // массив для дескрипторов потоков
  DWORD  dwThreadID; // массив для идентификаторов потоков

  // инициализируем структуру OVERLAPPED
  ovl.Offset = 0;      // младшая часть смещения равна 0
  ovl.OffsetHigh = 0;  // старшая часть смещения равна 0
  ovl.hEvent = 0;      // события нет

  // запрашиваем ключ файла
  cout << "Input a number for file key (not zero): ";
  cin >> ulKey;
  if (ulKey == 0)
  {
    cout << "The file key can't be equal to zero." << endl
      << "Press any key to exit." << endl;

    return 0;
  }

  // создаем файл для записи данных
  hFile = CreateFile(
    "C:\\demo_file.dat",   // имя файла
    GENERIC_WRITE,         // запись в файл
    FILE_SHARE_WRITE,      // совместный доступ к файлу
    NULL,                  // защиты нет
    OPEN_ALWAYS,           // открываем или создаем новый файл
    FILE_FLAG_OVERLAPPED,  // асинхронный доступ к файлу
```

```
  NULL                    // шаблона нет
);
// проверяем на успешное создание
if (hFile == INVALID_HANDLE_VALUE)
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish." << endl;

  cin.get();
  return 0;
}

// создаем порт завершения и подключаем к нему файл
hCompletionPort = CreateIoCompletionPort(
    hFile,   // дескриптор файла
    NULL,    // новый порт
    ulKey,   // ключа файла
    1        // один поток
  );

// проверяем на успешное создание
if (hCompletionPort == NULL)
{
  cerr << "Create completion port failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish." << endl;

  cin.get();
  return 0;
}

// запускаем поток
hThread = CreateThread(NULL, 0, thread, NULL, 0, &dwThreadID);

// пишем данные в файл
for (int i = 0; i < 10; ++i)
{
  DWORD  dwBytesWrite;
  DWORD  dwRet;
```

```
    if (!WriteFile(
        hFile,        // дескриптор файла
        &i,           // адрес буфера, откуда идет запись
        sizeof(i),    // количество записываемых байтов
        &dwBytesWrite,   // количество записанных байтов
        &ovl          // запись асинхронная
      ))
    {
      dwRet = GetLastError();
      if (dwRet == ERROR_IO_PENDING)
        cout << "Write file pending." << endl;
      else
      {
        cout << "Write file failed." << endl
          << "The last error code: " << dwRet << endl;
        return 0;
      }
    }
    // ждем, пока завершится асинхронная операция записи
    WaitForSingleObject(hFile, INFINITE);
    // увеличивает смещение в файле
    ovl.Offset += sizeof(i);
  }
  // посылаем пакет с командой на завершение потока
  PostQueuedCompletionStatus(
    hCompletionPort,   // дескрипор потока
    0,        // нет передачи
    0,        // ключ завершения
    NULL);    // нет структуры типа OVERLAPPED
  // ждем завершения потока
  WaitForSingleObject(hThread, INFINITE);
  // закрываем дескрипторы
  CloseHandle(hFile);
  CloseHandle(hCompletionPort);
  CloseHandle(hThread);

  cout << "The file is written." << endl;
  return 0;
}
```

Глава 29

Работа с ожидающим таймером

29.1. Ожидающий таймер

Ожидающим таймером (waitable timer) называется объект синхронизации, который переходит в сигнальное состояние при наступлении заданного момента времени. Создаются ожидающие таймеры функцией `CreateWaitableTimer`. После создания ожидающего таймера нужно установить тот момент времени, когда ожидающий таймер перейдет в сигнальное состояние. Для установки такого момента времени используется функция `SetWaitableTimer`. Момент перехода ожидающего таймера в сигнальное состояние может быть определен двумя способами: используя абсолютное системное время или временной интервал, который отсчитывается от текущего момента времени.

Если ожидающий таймер ждет момента для перехода в сигнальное состояние, то говорят, что он находится в *активном состоянии*. В противном случае таймер находится в *пассивном* или *неактивном состоянии*. В пассивном состоянии ожидающий таймер находится после его создания. В активное состояние таймер переводится посредством вызова функции `SetWaitableTimer`. Для перевода таймера из активного состояния в пассивное состояние поток может использовать функцию `CancelWaitableTimer`.

Для ожидания сигнального состояния ожидающего таймера поток использует обычные функции ожидания перехода объекта в сигнальное состояние. По способу перехода в несигнальное состояние ожидающие таймеры подразделяются на таймеры с ручным сбросом и таймеры с автоматическим сбросом, которые также называются *таймерами синхронизации*. Таймер с ручным сбросом переходит из сигнального состояния в несигнальное только посредством установки нового момента (времени) для этого таймера посредством вызова функции `SetWaitableTimer`. Таймер синхронизации переходит в несигнальное состояние посредством срабатывания функции ожидания, параметром которой является дескриптор этого таймера.

Существуют два типа ожидающих таймеров: периодические ожидающие таймеры и непериодические ожидающие таймеры. *Периодические ожидающие таймеры*, перейдя в сигнальное состояние, повторяют этот переход через заданные интервалы времени до тех пор, пока не будут переустановлены функцией `SetWaitableTimer` или отменены функцией `CancelWaitableTimer`. *Непериодический ожидающий таймер* переходит в сигнальное состояние только один раз при наступлении заданного момента.

После завершения использования таймера его дескриптор нужно закрыть при помощи функции `CloseHandle`.

29.2. Создание ожидающего таймера

Создается ожидающий таймер посредством функции `CreateWaitableTimer`, которая имеет следующий прототип:

```
HANDLE  CreateWaitableTimer(
  LPSECURITY_ATTRIBUTES  lpTimerAttributes,  // атрибуты безопасности
  BOOL     bManualReset,   // переход в несигнальное состояние
  LPCTSTR  lpTimerName     // имя таймера
);
```

В случае успешного завершения функция `CreateWaitableTimer` возвращает дескриптор вновь созданного или уже существующего таймера. В последнем случае функция `GetLastError` вернет значение `ERROR_ALREADY_EXISTS`. В случае неудачи функция возвращает значение `NULL`.

В параметре `lpTimerAttributes` устанавливаются атрибуты безопасности ожидающего таймера. Если значение этого параметра установлено в `NULL`, то атрибуты безопасности устанавливаются по умолчанию и дескриптор таймера не наследуется дочерними процессами.

Параметр `bManualReset` может принимать одно из двух значений `TRUE` или `FALSE`. Если установлено значение `FALSE`, то создается таймер синхронизации. В противном случае создается периодический таймер.

Параметр `lpTimerName` должен указывать на символьную строку, которая служит именем файла. Если значение этого параметра равно `NULL`, то создается безымянный таймер.

В следующих разделах будут приведены примеры программ, использующих ожидающие таймеры. В этих программах и будет показано, как создаются ожидающие таймеры, используя функцию `CreateWaitableTimer`.

29.3. Установка ожидающего таймера

Для установки времени перехода таймера в сигнальное состояние и других характеристик ожидающего таймера используется функция `SetWaitableTimer`, которая имеет следующий прототип:

```
BOOL  SetWaitableTimer(
  HANDLE  hTimer,                    // дескриптор таймера
  const  LARGE_INTEGER  pDueTime,    // время перехода в сигн. состояние
  LONG    lPeriod,                   // период времени
  PTIMERAPCROUTINE  pfnCompletionRoutine,  // процедура завершения
  LPVOID  lpArgToCompletionRoutine, // аргумент процедуры завершения
  BOOL    bResume                    // управление питанием
);
```

В случае успешного завершения функция `SetWaitableTimer` возвращает ненулевое значение, а в случае неудачи — `FALSE`.

Параметр `hTimer` должен содержать дескриптор ожидающего таймера.

Параметр `pDueTime` должен содержать адрес структуры типа `LARGE_INTRGER` с заданным моментом, при наступлении которого таймер перейдет в сигнальное состояние. Структура типа `LARGE_INTEGER` является целым числом длиной в 64 бита. Если это целое число имеет положительное значение, то считается, что задано абсолютное время. Если же это число отрицательное, то считается, что задан интервал от текущего значения системного времени. В обоих случаях единица измерения времени равна 100 наносекунд. Напомним, что 1 наносекунда равна 10^{-9} секунды.

Значение параметра `lPeriod` определяет, является ли ожидающий таймер периодическим. Если значение этого параметра равно нулю, то таймер — не периодический. Если же значение этого параметра больше нуля, то таймер является периодическим и величина периода времени, через который таймер переходит в сигнальное состояние, равна значению этого параметра. Единица измерения времени периода равна 1 миллисекунде. Напомним, что 1 миллисекунда = 10^{-3} секунды. Если значение параметра `lPeriod` меньше нуля, то вызов функции заканчивается неудачей.

Параметр `pfnCompletionRoutine` должен указывать на функцию завершения, которая вызывается после перехода ожидающего таймера в сигнальное состояние. При этом отметим, что в этом случае поток, вызвавший функцию `SetWaitableTimer`, должен находиться в настороженном состоянии. Если процедуры завершения нет, то этот параметр может быть установлен в `NULL`.

Параметр `lpArgToCompletionRoutine` может содержать единственный аргумент, который передается функции завершения.

В параметре bResume устанавливается режим управления питанием компьютера после перехода таймера в сигнальное состояние. Этот параметр может принимать значения FALSE или TRUE. Если задано значение TRUE, то компьютер переключается в режим экономии энергии. В противном случае переключение не происходит.

В листинге 29.1 приведена программа, в которой создается ожидающий таймер и устанавливается момент, определяющий переход таймера в сигнальное состояние.

Листинг 29.1. Создание и установка ожидающего таймера

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include <iostream.h>

#define _SECOND 10000000   // одна секунда для ожидающего таймера

HANDLE  hTimer;    // ожидающий таймер

DWORD WINAPI thread(LPVOID)
{
  // ждем сигнал от ожидающего таймера
  WaitForSingleObject(hTimer, INFINITE);
  // выводим сообщение
  cout << "\aThe timer is signaled." << endl;

  return 0;
}

int main()
{
  HANDLE  hThread;
  DWORD  IDThread;

  _int64  qwTimeInterval;    // время задержки для таймера

  // создаем ожидающий таймер
```

```
  hTimer = CreateWaitableTimer(NULL, FALSE, NULL);
  if (hTimer == NULL)
    return GetLastError();

  // время задержки для таймера = 3 секунды
  qwTimeInterval = -3 * _SECOND;

  // инициализируем таймер
  if (!SetWaitableTimer(
    hTimer,  // дескриптор таймера
    (LARGE_INTEGER*)&qwTimeInterval,   // временной интервал
    0,       // непериодический таймер
    NULL,    // процедуры завершения нет
    NULL,    // параметров к этой процедуре нет
    FALSE    // режим сбережения энергии не устанавливать
    ))
  {
    cout << "Set waitable timer failed." << endl
      << "The last error code: " << GetLastError() << endl;

    return 0;
  }

  // запускаем поток
  hThread = CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
  if (hThread == NULL)
    return GetLastError();

  // ждем, пока поток закончит работу
  WaitForSingleObject(hThread, INFINITE);
  // закрываем дескрипторы
  CloseHandle(hThread);
  CloseHandle(hTimer);

  return 0;
}
```

29.4. Отмена ожидающего таймера

Для перевода таймера в неактивное состояние предназначена функция `CancelWaitableTimer`, которая имеет следующий прототип:

```
BOOL  CancelWaitableTimer(
  HANDLE  hTimer     // дескриптор таймера
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. Единственным параметром этой функции является дескриптор таймера, который переходит в неактивное состояние.

В листинге 29.2 приведена программа, в которой таймер переводится в неактивное состояние посредством вызова функции `CancelWaitableTimer`.

Листинг 29.2. Перевод ожидающего таймера в неактивное состояние

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include <iostream.h>

#define _SECOND 10000000   // одна секунда для ожидающего таймера

HANDLE  hHandle[2];  // событие для выхода из цикла и ожидающий таймер

DWORD WINAPI thread(LPVOID)
{
  DWORD  dwRetCode;  // код возврата из функции ожидания

  for (;;)
  {
    // ждем сигнал от ожидающего таймера или на выход из цикла
    dwRetCode = WaitForMultipleObjects(2, hHandle, FALSE, INFINITE);
    // определяем индекс дескриптора
    dwRetCode -= WAIT_OBJECT_0;

    switch (dwRetCode)
    {
```

```
    case 0:    // сработало событие
      cout << "The timer is canceled." << endl;
      return 0;
    case 1:    // сработал таймер
      cout << "\aThe waitable timer is signaled." << endl;
      break;
    default:
      cout << "Default statement." << endl;
      return 0;
    }
  }
}

int main()
{
  HANDLE  hThread;
  DWORD  IDThread;

  _int64  qwTimeInterval;    // время задержки для таймера

  // создаем события с автоматическим сбросом
  hHandle[0] = CreateEvent(NULL, FALSE, FALSE, NULL);
  if (hHandle[0] == NULL)
    return GetLastError();

  // создаем ожидающий таймер
  hHandle[1] = CreateWaitableTimer(
      NULL,    // защиты нет
      FALSE,   // автоматический сброс
      NULL     // имени нет
    );
  if (hHandle[1] == NULL)
    return GetLastError();

  // время задержки для таймера = 2 секунды
  qwTimeInterval = -2 * _SECOND;

  // инициализируем таймер
  if (!SetWaitableTimer(
```

```
    hHandle[1],  // дескриптор таймера
    (LARGE_INTEGER*)&qwTimeInterval,  // временной интервал
    2000,    // период = 2 секунды
    NULL,    // процедуры завершения нет
    NULL,    // параметров к этой процедуре нет
    FALSE    // режим сбережения энергии не устанавливать
    ))
  {
    cout << "Set waitable timer failed." << endl
      << "The last error code: " << GetLastError() << endl;

    return 0;
  }

  // запускаем поток
  hThread = CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
  if (hThread == NULL)
    return GetLastError();

  // ждем команды на снятие ожидающего таймера
  char  c;
  cout << "Input any char to cancel waitable timer." << endl;
  cin >> c;

  // снимаем ожидающий таймер
  CancelWaitableTimer(hHandle[1]);

  // даем потоку сигнал на завершение работы
  SetEvent(hHandle[0]);

  // ждем, пока поток закончит работу
  WaitForSingleObject(hThread, INFINITE);
  // закрываем дескрипторы
  CloseHandle(hHandle[0]);
  CloseHandle(hHandle[1]);

  return 0;
}
```

29.5. Открытие существующего ожидающего таймера

Для открытия уже существующего ожидающего таймера используется функция OpenWaitableTimer, которая имеет следующий прототип:

```
HANDLE  OpenWaitableTimer(
  DWORD    dwDesiredAccess,  // режимы доступа
  BOOL     bInheritHandle,   // режим наследования
  LPCTSTR  lpTimerName       // имя таймера
);
```

В случае успешного завершения функция возвращает дескриптор уже существующего таймера, а в случае неудачи — значение NULL.

В параметре dwDesiredAccess устанавливаются режимы доступа к открываемому таймеру. В этом параметре может быть установлена любая комбинация следующих флагов:

- TIMER_ALL_ACCESS — произвольный доступ к таймеру;
- TIMER_MODIFY_STATE — можно только изменять состояние таймера, используя функции SetWaitableTimer и CancelWaitableTimer;
- TIMER_QUERY_STATE — не используется;
- SYNCHRONIZE — можно использовать таймер только в функциях ожидания.

Отметим, что режим SYNCHRONIZE работает только в операционных системах Windows NT/2000/XP.

Параметр bInheritHandle может принимать значения TRUE или FALSE. Если устанавливается значение TRUE, то дескриптор таймера наследуется дочерними процессами, в противном случае дескриптор ненаследуемый.

Параметр lpTimerName должен указывать на имя открываемого таймера, который должен быть предварительно создан другим процессом.

В листинге 29.3 приведена программа родительского процесса, который создает ожидающий таймер и запускает дочерний процесс.

Листинг 29.3. Программа родительского процесса, который создает ожидающий таймер

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include <iostream.h>
```

```
#define _SECOND 10000000   // одна секунда для ожидающего таймера

int main()
{
  HANDLE  hTimer;          // ожидающий таймер

  _int64  qwTimeInterval;  // время задержки для таймера

  // создаем ожидающий таймер
  hTimer = CreateWaitableTimer(NULL, FALSE, "demo_timer");
  if (hTimer == NULL)
    return GetLastError();

  // время задержки для таймера = 2 секунды
  qwTimeInterval = -2 * _SECOND;

  // инициализируем таймер
  if (!SetWaitableTimer(
    hTimer,  // дескриптор таймера
    (LARGE_INTEGER*)&qwTimeInterval,  // временной интервал
    0,       // непериодический таймер
    NULL,    // процедуры завершения нет
    NULL,    // параметров к этой процедуре нет
    FALSE    // режим сбережения энергии не устанавливать
    ))
  {
    cout << "Set waitable timer failed." << endl
      << "The last error code: " << GetLastError() << endl;

    return 0;
  }

  // создаем новый консольный процесс
  char lpszAppName[] = "C:\\DemoProcess.exe";

  STARTUPINFO si;
  PROCESS_INFORMATION piApp;

  ZeroMemory(&si, sizeof(STARTUPINFO));
```

```
  si.cb = sizeof(STARTUPINFO);

  if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
      CREATE_NEW_CONSOLE, NULL, NULL, &si, &piApp))
  {
    cout << "The new process is not created." << endl
      << "Check a name of the process." << endl
      << "Press any key to finish." << endl;

    cin.get();

    return 0;
  }

  // ждем завершения созданного процесса
  WaitForSingleObject(piApp.hProcess, INFINITE);
  // закрываем дескрипторы этого процесса в текущем процессе
  CloseHandle(piApp.hThread);
  CloseHandle(piApp.hProcess);

  // закрываем таймер
  CloseHandle(hTimer);

  return 0;
}
```

В листинге 29.4 приведена программа дочернего процесса, который открывает ожидающий таймер, созданный родительским процессом, и ждет перехода этого таймера в сигнальное состояние.

Листинг 29.4. Открытие существующего ожидающего таймера дочерним процессом

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include <iostream.h>



int main()
```

```
{
  HANDLE  hTimer;    // ожидающий таймер

  // открываем существующий ожидающий таймер
  hTimer = OpenWaitableTimer(
    TIMER_ALL_ACCESS,
    FALSE,
    "demo_timer"
    );
  if (hTimer == NULL)
  {
    cout << "Open waitable timer failed." << endl
      << "The last error code: " << GetLastError() << endl;
    return 0;
  }

  // ждем сигнал от ожидающего таймера
  WaitForSingleObject(hTimer, INFINITE);
  // выводим сообщение
  cout << "\aThe timer is signaled." << endl;
  // закрываем таймер
  CloseHandle(hTimer);
  // ждем команды на выход из процесса
  cout << "Press any key to exit." << endl;
  cin.get();

  return 0;
}
```

29.6. Процедуры завершения ожидания

При установке характеристик ожидающего таймера посредством функции `SetWaitableTimer` можно определить процедуру завершения, которая будет вызвана при переходе таймера в сигнальное состояние. Эта процедура ставится в очередь асинхронных процедур потока и вызывается только в том случае, если поток находится в настороженном состоянии. Отметим, что вызов функции `CancelWaitableTimer` удаляет процедуру завершения ожидающего таймера из очереди асинхронных процедур. Подробно механизм вызова асинхронных процедур рассмотрен в *гл. 26*.

Процедура завершения ожидающего таймера имеет следующий прототип:

```
VOID  CALLBACK  имя_процедуры_завершения (
  LPVOID  lpArgToCompletionRoutine,  // параметр
  DWORD   dwTimerLowValue,           // младшее значение таймера
  DWORD   dwTimerHighValue           // старшее значение таймера
);
```

Эта процедура не возвращает значения и получает три параметра.

В параметре `lpArgToCompletionRoutine` процедура завершения получает единственный аргумент, который был задан при вызове функции `SetWaitableTimer` в аналогичном параметре.

В параметрах `dwTimerLowValue` и `dwTimerHighValue` процедура завершения получает, соответственно, младшую и старшую части из структуры типа `LARGE_INTEGER`, которая задает момент перехода ожидающего таймера в сигнальное состояние.

В листинге 29.5 приведена программа, в которой вызывается процедура завершения при переходе таймера в сигнальное состояние. Отметим, что, в этой программе для перевода потока в настороженное состояние используется функция `SleepEx`.

Листинг 29.5. Пример вызова процедуры завершения ожидающего таймера

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include <iostream.h>

#define _SECOND 10000000   // одна секунда для ожидающего таймера

VOID CALLBACK completion_proc(LPVOID lpArg,
         DWORD dwTimerLowValue, DWORD dwTimerHighValue)
{
  // выводим сообщение
  cout << "\aThe timer is signaled." << endl;
  // распечатываем аргументы
  cout << "Arguments: " << (char*)lpArg << endl
    << "Timer low value: " << dwTimerLowValue << endl
    << "Timer high value: " << dwTimerHighValue << endl;
}

int main()
```

```
{
  HANDLE  hTimer;            // ожидающий таймер

  _int64  qwTimeInterval;    // время задержки для таймера

  // создаем ожидающий таймер
  hTimer = CreateWaitableTimer(NULL, FALSE, NULL);
  if (hTimer == NULL)
    return GetLastError();

  // время задержки для таймера = 1 секунда
  qwTimeInterval = -1 * _SECOND;

  // инициализируем таймер
  if (!SetWaitableTimer(
    hTimer,              // дескриптор таймера
    (LARGE_INTEGER*)&qwTimeInterval,   // временной интервал
    0,                   // непериодический таймер
    completion_proc,     // процедура завершения
    "Demo parameters.",  // параметров к этой процедуре нет
    FALSE                // режим сбережения энергии не устанавливать
    ))
  {
    cout << "Set waitable timer failed." << endl
      << "The last error code: " << GetLastError() << endl;

    return 0;
  }
  // переводим поток в настороженное состояние
  SleepEx(INFINITE, TRUE);

  // закрываем таймер
  CloseHandle(hTimer);

  return 0;
}
```

Часть IX

Динамически подключаемые библиотеки

Глава 30

Отображение файлов в память

30.1. Концепция механизма отображения файлов в память

Прежде чем перейти к описанию динамически подключаемых библиотек, рассмотрим механизм, который позволяет динамически выполнять это подключение. Этот механизм называется *отображением содержимого файла* (file mapping) в виртуальную память процесса.

Как уже говорилось в *гл. 20*, посвященной организации виртуальной памяти, операционная система создает файлы подкачки с виртуальными страницами, которые система отображает в адресные пространства процессов. В операционных системах Windows реализован механизм, который позволяет отображать в адресное пространство процесса не только содержимое файлов подкачки, но и содержимое обычных файлов. То есть в этом случае файл или его часть рассматривается как набор виртуальных страниц процесса, которые имеют последовательные логические адреса. Файл, отображенный в адресное пространство процесса, называется *представлением* или *видом файла* (file view). После отображения файла в адресное пространство процесса доступ к виду может осуществляться с помощью указателя, как к обычным данным в адресном пространстве процесса.

Несколько процессов могут одновременно отображать один и тот же файл в свое адресное пространство. В этом случае операционная система обеспечивает согласованность содержимого файла для всех процессов, если доступ к этим данным осуществляется как к области виртуальной памяти процесса. То есть для доступа к файлу, который отображен в память, не используется функция `WriteFile`. Такая согласованность данных, хранящихся в файле, отображенном в память несколькими процессами, называется *когерентностью данных*. Однако следует отметить, что когерентность данных для файла, отображенного в память, не поддерживается в том случае, если этот файл ото-

бражается в адресное пространство процессов, которые выполняются на других компьютерах в локальной сети.

Из вышесказанного можно сделать вывод, что основным назначением механизма отображения файлов в память является загрузка программы (файла с расширением exe) на выполнение, в адресном пространстве процесса, и динамическое подключение библиотек функций во время выполнения этой программы.

Кроме того, механизм отображения файлов в память позволяет осуществлять обмен данными между процессами, принимая во внимание то, что система обеспечивает когерентность данных в файле, отображаемом в память.

Теперь кратко опишем общую последовательность действий, которые необходимо выполнить для работы с отображаемым в память файлом. Эти действия могут быть разбиты на следующие шаги:

- открыть файл, который будет отображаться в память;
- создать объект ядра, который выполняет отображение файла;
- отобразить файл или его часть в адресное пространство процесса;
- выполнить необходимую работу с видом файла;
- отменить отображение файла;
- закрыть объект ядра для отображения файла;
- закрыть файл, который отображался в память.

Все эти шаги и функции, необходимые для их выполнения, будут рассмотрены в следующих разделах.

30.2. Создание и открытие объекта, отображающего файл

Если в память отображается существующий файл, то первым делом этот файл должен быть открыт для доступа, используя функцию `CreateFile`. Это делается для того, чтобы получить дескриптор файла, т. к. в дальнейшем этот дескриптор используется при создании объекта, отображающего файл в память процесса. Если отображаемый файл используется просто для обмена данными между процессами, то создавать для этого специальный файл на диске не обязательно. Для этого можно использовать файлы подкачки операционной системы. Как это делается, будет рассмотрено подробно в *разд. 30.4*.

После того как файл открыт, создается объект, отображающий этот файл в память. Под объектом, отображающим файл в память, можно понимать объект ядра операционной системы, который выполняет отображение файла в адресное пространство процесса. Можно также представить, что этот объ-

ект позволяет рассматривать файл, отображаемый в память, как файл подкачки. Для создания этого объекта используется функция `CreateFileMapping`, которая имеет следующий прототип:

```
HANDLE  CreateFileMapping(
  HANDLE   hFile,              // дескриптор файла
  LPSECURITY_ATTRIBUTES,  lpAttributes   // атрибуты защиты
  DWORD    flProtect,          // флаги доступа к файлу
  DWORD    dwMaximumSizeHigh,  // старшее двойное слово размера объекта
  DWORD    dwMaximumSizeLow,   // младшее двойное слово размера объекта
  LPCTSTR  lpName              // имя объекта отображения
);
```

В случае успешного завершения эта функция возвращает дескриптор объекта, отображающего файл в память, а в случае неудачи — `NULL`. Параметры этой функции имеют следующее назначение.

Параметр `hFile` должен содержать дескриптор открытого файла, для которого будет создаваться объект, отображающий этот файл в память процесса.

Параметр `lpAttributes`, как обычно, указывает на атрибуты защиты для объекта, отображающего файл в память. В этом разделе этот параметр будет всегда устанавливаться в `NULL`, задавая, тем самым, атрибуты защиты по умолчанию. То есть в этом случае объект отображения не является наследуемым и принадлежит создавшему его пользователю.

Параметр `flProtect` содержит флаги, которые задают режимы доступа к виду файла в памяти процесса. Этот параметр может принимать одно из следующих значений:

- `PAGE_READONLY` — из вида файла можно только читать данные;
- `PAGE_READWRITE` — разрешает чтение и запись данных в вид файла;
- `PAGE_WRITECOPY` — разрешает чтение и запись данных в вид файла, но при записи создается новая копия вида файла.

Как можно видеть, значения этого параметра совпадают со значениями соответствующего параметра функции `VirtualAlloc`, которая распределяет виртуальную память процессу. Отметим, что режимы доступа к объекту, отображающему файл в память, должны соответствовать режимам доступа к файлу, для которого создается этот объект отображения.

Кроме того, в параметре `flProtect` может быть установлена любая комбинация флагов, которые определяют атрибуты секций исполняемых файлов, заданных в переносимом формате (portable executable files). Эти флаги рассматриваться не будут.

Параметры `dwMaximumSizeHigh` и `dwMaximumSizeLow` определяют соответственно значение старшей и младшей частей, которые в совокупности задают

размер объекта, отображающего файл в память. Если эти значения установлены в 0, то объект, отображающий файл в память, имеет размер, равный размеру файла. Отметим, что если размер этого объекта будет меньше размера файла, то система не сможет отобразить весь файл в память. Если же размер объекта, отображающего файл, больше чем размер файла, то размер файла увеличивается до размера объекта.

Последний параметр `lpName` используется для задания имени объекта, отображающего файл в память. Как обычно, это имя используется для доступа к одному и тому же объекту в разных процессах. Если процесс пытается получить доступ к уже созданному объекту, отображающему файл, то флаги доступа, установленные в параметре `flProtect`, должны соответствовать флагам доступа, уже установленным в существующем объекте, отображающем файл.

Пример использования функции `CreateFileMapping` будет приведен в следующем разделе. А в заключение этого раздела отметим, что, как обычно, после завершения работы с объектом, отображающим файл в память, его дескриптор нужно закрыть, используя функцию `CloseHandle`.

30.3. Отображение файла в память

После того как создан объект, отображающий файл в память, файл или его часть должна быть отображена в память процесса. Другими словами, должен быть создан вид файла или его части в адресном пространстве процесса. Для отображения файла или его части в адресное пространство процесса используется функция `MapViewOfFile`, которая имеет следующий прототип:

```
LPVOID  MapViewOfFile(
  HANDLE  hFileMappingObject, // дескриптор объекта, отображающего файл
  DWORD   dwDesiredAccess,    // режим доступа
  DWORD   dwFileOffsetHigh,   // старшее двойное слово смещения
  DWORD   dwFileOffsetLow,    // младшее двойное слово смещения
  SIZE_T  dwNumberOfBytesToMap  // количество отображаемых байт
);
```

В случае успешного завершения функция возвращает указатель на вид файла в адресном пространстве процесса, а случае неудачи — `NULL`. Параметры этой функции имеют следующее назначение.

Параметр `hFileMappingObject` должен содержать дескриптор объекта, отображающего файл в память, который был предварительно создан функцией `CreateFileMapping`.

Параметр `dwDwsiredAccess` задает режим доступа к виду файла и может принимать одно из следующих значений:

- `FILE_MAP_WRITE` — чтение и запись в вид файла;

- `FILE_MAP_READ` — только чтение из вида файла;
- `FILE_MAP_ALL_ACCESS` — чтение и запись в вид файла;
- `FILE_MAP_COPY` — при записи в вид файла создается его копия, а исходный файл не изменяется.

Следует отметить, что установленное в этом параметре значение должно соответствовать режиму доступа, который установлен для объекта, отображающего файл в память.

Параметры `dwFileOffsetHigh` и `dwFileOffsetLow` задают смещение от начала файла или, другими словами, первый байт файла, начиная с которого файл отображается в память. Это смещение задается в байтах и должно быть кратно гранулярности распределения виртуальной памяти в системе (allocation granularity), которая может быть определена при помощи вызова функции `GetSystemInfo`. Единственным параметром этой функции является указатель на структуру `SYSTEM_INFO`, в поле `dwAllocationGranularity` которой, функция `GetSystemInfo` помещает гранулярность (в байтах). Это значение зависит от архитектуры компьютера и в большинстве случаев равно 64 Кбайт.

Параметр `dwNumberOfBytesToMap` задает количество байт, которые будут отображаться в память из файла. Если значение этого параметра равно нулю, то в память будет отображен весь файл.

Если необходимо отобразить файл в адресное пространство процесса, начиная с некоторого заданного виртуального адреса, то для этой цели нужно использовать функцию `MapViewOfFileEx`, которая имеет следующий прототип:

```
LPVOID  MapViewOfFileEx(
  HANDLE  hFileMappingObject, // дескриптор объекта, отображающего файл
  DWORD   dwDesiredAccess,    // режим доступа
  DWORD   dwFileOffsetHigh,   // старшее двойное слово смещения
  DWORD   dwFileOffsetLow,    // младшее двойное слово смещения
  SIZE_T  dwNumberOfBytesToMap, // количество отображаемых байтов
  LPVOID  lpBaseAddress       // начальный адрес отображения файла
);
```

Последний параметр этой функции указывает на начальный виртуальный адрес вида отображаемого файла. Этот параметр может быть установлен в `NULL`. В этом случае система сама выберет начальный адрес загрузки. Остальные параметры этой функции соответствуют параметрам функции `MapViewOfFile`.

После окончания работы с видом файла в памяти нужно отменить отображение файла в адресное пространство процесса. Отмена отображения файла освобождает виртуальные адреса процесса. Следует особо отметить, что если

отображение файла в адресное пространство процесса не отменено, то система продолжает держать отображаемый файл открытым до тех пор, пока существует его вид, независимо от того, закрыт этот файл функцией `CloseHandle` или нет. Для отмены отображения файла в память используется функция `UnmapViewOfFile`, которая имеет следующий прототип:

```
BOOL  UnmapViewOfFile(LPCVOID  lpBaseAddress);
```

В случае успешного завершения возвращает ненулевое значение, а в случае неудачи — `FALSE`. Единственным параметром этой функции является начальный адрес вида файла. Этот адрес должен быть предварительно получен одной из функций `MapViewOfFile` или `MapViewOfFileEx`.

В листинге 30.1 приведена программа, которая отображает файл в память. Предварительно создадим файл, который представляет собой массив целых чисел. После этого увеличим значения элементов этого массива на десять, используя вид файла. Затем выведем содержимое измененного файла на экран.

Листинг 30.1. Работа с видом файла

```
#include <windows.h>
#include <fstream.h>

int main()
{
  int  a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  char  file_name[] = "Demo.bin";
  HANDLE  hFile, hMapping;   // дескрипторы файла и объекта отображения
  int    *ptr;               // для указателя на массив

  // открываем файл для вывода
  ofstream    out(file_name, ios::out | ios::binary);
  if (!out)
  {
    cerr << "File constructor failed." << endl;
    return 0;
  }
  // выводим исходный массив в файл и на консоль
  cout << "Initial array: ";
  for (int i = 0; i < 10; ++i)
  {
    out.write((char*)&a[i], sizeof(int));
```

```
    cout << a[i] << ' ';
  }
  cout << endl;
  // закрываем выходной файл
  out.close();
// ----------------------------------------------------------------------
  // открываем файл для отображения в память
  hFile = CreateFile(file_name, GENERIC_READ | GENERIC_WRITE,
    0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
  if (hFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl;
    return GetLastError();
  }
  // открываем объект, отображающий файл в память
  hMapping = CreateFileMapping(
        hFile,     // дескриптор открытого файла
        NULL,      // атрибуты защиты по умолчанию
        PAGE_READWRITE,  // режим доступа
        0, 0,      // размер объекта отображения равен размеру файла
        NULL);     // имя не используем
  if (!hMapping)
  {
    cerr << "Create file mapping failed." << endl;
    return GetLastError();
  }
  // создаем вид файла
  ptr = (int*)MapViewOfFile(
      hMapping,    // дескриптор объекта отображения
      FILE_MAP_WRITE,    // режим доступа к виду
      0, 0,        // отображаем файл с начала
      0);          // отображаем весь файл
  // изменяем значения элементов массива
  for (i = 0; i < 10; ++i)
    ptr[i] += 10;
  // отменяем отображение файла в память
  if (!UnmapViewOfFile(ptr))
```

```
  {
    cerr << "Unmap view of file failed." << endl;
    return GetLastError();
  }
  // закрываем объект отображения файла в память
  if (!CloseHandle(hMapping))
  {
    cerr << "Close file failed." << endl;
    return GetLastError();
  }
  // закрываем файл
  if (!CloseHandle(hFile))
  {
    cerr << "Close file failed." << endl;
    return GetLastError();
  }
// ----------------------------------------------------------------------
  // открываем файл для ввода
  ifstream    in(file_name, ios::in | ios::binary);
  if (!in)
  {
    cerr << "File constructor failed." << endl;
    return 0;
  }
  // вводим финальный массив из файла и выводим на консоль
  cout << "Final array: ";
  for (i = 0; i < 10; ++i)
  {
    in.read((char*)&a[i], sizeof(int));
    cout << a[i] << ' ';
  }
  cout << endl;
  // закрываем входной файл
  in.close();

  return 0;
}
```

30.4. Обмен данными между процессами через отображаемый в память файл

Так как один и тот же файл может быть отображен в память несколькими процессами, и система поддерживает когерентность таких отображений, то механизм отображения файлов в память может использоваться для обмена данными между процессами. Кроме того, в операционных системах Windows все остальные механизмы обмена данными между процессами базируются на отображении файлов в память. Поэтому можно сказать, что отображение файлов в память обеспечивает наилучшую производительность по сравнению со всеми остальными способами обмена данными между процессами.

Так как безразлично, какой файл используется для обмена данными между процессами, то в этом случае лучше использовать файл подкачки страниц. В программах из листингов 30.2, 30.3 показывается, как передать данные другому процессу через файл подкачки страниц. Первая программа создает объект, отображающий файл подкачки, а затем записывает в вид файла подкачки массив целых чисел. После этого она запускает вторую программу, которая будет читать созданный массив, через отображаемый в память файл, используя тот же объект, отображающий файл в память. Отметим, что для обращения к одному и тому же объекту, отображающему файл, используется имя этого объекта.

Листинг 30.2. Передача данных через отображаемый в память файл

```
#include <windows.h>
#include <fstream.h>

int main()
{
  char    MappingName[] = "MappingName";
  HANDLE  hMapping;      // дескриптор объекта, отображающего файл
  int    *ptr;           // для указателя на массив
  const int    n = 10;   // размерность массива

  cout << "This is a parent process." << endl;
  // открываем объект отображения файла в память
  hMapping = CreateFileMapping(
        INVALID_HANDLE_VALUE,  // файл подкачки страниц
        NULL,            // атрибуты защиты по умолчанию
        PAGE_READWRITE,  // режим доступа: чтение и запись
```

```
        0,                // старшее слово = 0
        n * sizeof(int), // младшее слово = длине массива
        MappingName);     // имя объекта отображения
  if (!hMapping)
  {
    cerr << "Create file mapping failed." << endl;
    return GetLastError();
  }
  // создаем вид файла
  ptr = (int*)MapViewOfFile(
      hMapping,         // дескриптор объекта отображения
      FILE_MAP_WRITE,   // режим доступа к виду
      0,0,              // отображаем файл с начала
      0);               // отображаем весь файл
  // инициализируем массив и выводим его на консоль
  cout  << "Array: ";
  for (int i = 0; i < n; ++i)
  {
    ptr[i] = i;
    cout << ptr[i] << ' ';
  }
  cout << endl;
//---------------------------------------------------------------------
// создаем процесс, который будет читать данные из отображаемого
// в память файла
  char lpszAppName[] = "C:\\ConsoleProcess.exe";

  STARTUPINFO si;
  PROCESS_INFORMATION piApp;

  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb = sizeof(STARTUPINFO);

  // создаем новый консольный процесс
  if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
      CREATE_NEW_CONSOLE, NULL, NULL, &si, &piApp))
  {
    cerr << "Create process failed." << endl;
    return GetLastError();
```

```
  }
  // ждем завершения созданного прцесса
  WaitForSingleObject(piApp.hProcess, INFINITE);
  // закрываем дескрипторы этого процесса в текущем процессе
  CloseHandle(piApp.hThread);
  CloseHandle(piApp.hProcess);
//-----------------------------------------------------------------------
  // отменяем отображение файла в память
  if (!UnmapViewOfFile(ptr))
  {
    cerr << "Unmap view of file failed." << endl;
    return GetLastError();
  }
  // закрываем объект отображения файла в память
  if (!CloseHandle(hMapping))
  {
    cerr << "Close file failed." << endl;
    return GetLastError();
  }
  // ждем команду на завершение процесса
  char  c;
  cout << "Input any char to exit: ";
  cin >> c;

  return 0;
}
```

Листинг 30.3. Прием данных через отображаемый в память файл

```
#include <windows.h>
#include <fstream.h>

int main()
{
  char  MappingName[] = "MappingName";
  HANDLE  hMapping;      // дескриптор объекта, отображающего файл
  int    *ptr;           // для указателя на массив
  const int    n = 10;   // размерность массива
```

```
cout << "This is a child process." << endl;
// открываем объект отображения файла в память
hMapping = CreateFileMapping(
      INVALID_HANDLE_VALUE,  // файл подкачки страниц
      NULL,            // атрибуты защиты по умолчанию
      PAGE_READWRITE,  // режим доступа: чтение и запись
      0,               // старшее слово = 0
      n * sizeof(int), // младшее слово = длине массива
      MappingName);    // имя объекта отображения
if (!hMapping)
{
  cerr << "Create file mapping failed." << endl;
  return GetLastError();
}
// создаем вид файла
ptr = (int*)MapViewOfFile(
    hMapping,        // дескриптор объекта отображения
    FILE_MAP_WRITE,  // режим доступа к виду
    0,0,             // отображаем файл с начала
    0);              // отображаем весь файл
// выводим массив из вида на консоль
cout  << "Array: ";
for (int i = 0; i < n; ++i)
  cout << ptr[i] << ' ';
cout << endl;
// отменяем отображение файла в память
if (!UnmapViewOfFile(ptr))
{
  cerr << "Unmap view of file failed." << endl;
  return GetLastError();
}
// закрываем объект отображения файла в память
if (!CloseHandle(hMapping))
{
  cerr << "Close file failed." << endl;
  return GetLastError();
}
// ждем команду на завершение процесса
char  c;
```

```
  cout << "Input any char to exit: ";
  cin >> c;
  return 0;
}
```

30.5. Сброс вида в файл

Для записи на диск содержимого вида файла используется функция `FlushViewOfFile`, которая имеет следующий прототип:

```
BOOL  FlushViewOfFile(
  LPCVOID  lpBaseAddress,          // базовый адрес вида
  SIZE_T   dwNumberOfButesToFlush  // количество записываемых байтов
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — `FALSE`.

Параметр `lpBaseAddress` этой функции должен содержать начальный адрес, который принадлежит диапазону адресов одного из видов и, начиная с которого, содержимое вида записывается на диск, а параметр `dwNumberOfButesToFlush` — количество записываемых байт. Если значение второго параметра равно нулю, то на диск записывается весь вид файла, начиная с базового адреса и до конца этого вида.

Следует отметить, что функция `FlushViewOfFile` гарантирует, что данные были записаны на диск локального компьютера, но не гарантирует запись этих данных в вид на удаленном компьютере в локальной сети.

Далее, в листингах 30.4, 30.5, приведены две программы, которые иллюстрируют использование функции `FlushViewOfFile`. Первая программа создает файл, затем отображает его в память и изменяет его содержимое. После этого запускается вторая программа, которая читает содержимое отображенного в память файла. Чтобы вторая программа прочитала измененный файл, вид перед ее запуском сбрасывается на диск при помощи вызова функции `FlushViewOfFile`.

Листинг 30.4. Сброс вида в файл на диске

```
#include <windows.h>
#include <fstream.h>

int main()
{
```

```
  int  a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  char  file_name[] = "C:\\Demo.bin";
  char  mapping_name[] = "MappingName";
  HANDLE  hFile, hMapping;   // дескрипторы файла и объекта отображения
  int  *ptr;                 // для указателя на массив

  // открываем файл для вывода
  ofstream  out(file_name, ios::out | ios::binary);
  if (!out)
  {
    cerr << "File constructor failed." << endl;
    return 0;
  }
  // выводим исходный массив в файл и на консоль
  cout << "Initial array: ";
  for (int i = 0; i < 10; ++i)
  {
    out.write((char*)&a[i], sizeof(int));
    cout << a[i] << ' ';
  }
  cout << endl;
  // закрываем выходной файл
  out.close();
//---------------------------------------------------------------------------
  // открываем файл для отображения в память
  hFile = CreateFile(file_name, GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE,
    NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
  if (hFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl;
    return GetLastError();
  }
  // открываем объект, отображающий файл в память
  hMapping = CreateFileMapping(
        hFile,           // дескриптор открытого файла
        NULL,            // атрибуты защиты по умолчанию
        PAGE_READWRITE,  // режим доступа
        0, 0,      // размер объекта отображения равен размеру файла
```

```
      mapping_name);   // имя объекта отображения
  if (!hMapping)
  {
    cerr << "Create file mapping failed." << endl;
    return GetLastError();
  }
  // создаем вид файла
  ptr = (int*)MapViewOfFile(
      hMapping,        // дескриптор объекта отображения
      FILE_MAP_WRITE,  // режим доступа к виду
      0, 0,            // отображаем файл с начала
      0);              // отображаем весь файл
  // изменяем значения элементов массива
  for (i = 0; i < 10; ++i)
    ptr[i] += 10;
  // сбрасываем весь вид на диск
  if (!FlushViewOfFile(ptr, 0))
  {
    cerr << "Flush view of file failed." << endl;
    return GetLastError();
  }
//-----------------------------------------------------------------------
// создаем процесс, который будет читать данные из отображаемого
// в память файла
  char lpszAppName[] = "C:\\ConsoleProcess.exe";

  STARTUPINFO si;
  PROCESS_INFORMATION piApp;

  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb = sizeof(STARTUPINFO);

  // создаем новый консольный процесс
  if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
      CREATE_NEW_CONSOLE, NULL, NULL, &si, &piApp))
  {
    cerr << "Create process failed." << endl;
    return GetLastError();
  }
```

```
  // ждем завершения созданного прцесса
  WaitForSingleObject(piApp.hProcess, INFINITE);
  // закрываем дескрипторы этого процесса в текущем процессе
  CloseHandle(piApp.hThread);
  CloseHandle(piApp.hProcess);
//---------------------------------------------------------------------------
// отменяем отображение файла в память
  if (!UnmapViewOfFile(ptr))
  {
    cerr << "Unmap view of file failed." << endl;
    return GetLastError();
  }
  // закрываем объкт отображения файла в память
  if (!CloseHandle(hMapping))
  {
    cerr << "Close file failed." << endl;
    return GetLastError();
  }
  // закрываем файл
  if (!CloseHandle(hFile))
  {
    cerr << "Close file failed." << endl;
    return GetLastError();
  }
  // ждем команду на завершение процесса
  char  c;
  cout << "Input any char to exit: ";
  cin >> c;

  return 0;
}
```

Листинг 30.5. Чтение данных из файла, который отображен в память другим процессом

```
#include <windows.h>
#include <fstream.h>

int main()
```

```
{
  char  file_name[] = "C:\\Demo.bin";
  int    a[10];

  // открываем файл для ввода
  ifstream  in(file_name, ios::in | ios::binary);
  if (!in)
  {
    cerr << "File constructor failed." << endl;
    return 0;
  }
  // вводим финальный массив из файла и выводим на консоль
  cout << "Final array: ";
  for (int i = 0; i < 10; ++i)
  {
    in.read((char*)&a[i], sizeof(int));
    cout << a[i] << ' ';
  }
  cout << endl;
  // закрываем входной файл
  in.close();

  // ждем команду на завершение процесса
  char  c;
  cout << "Input any char to exit: ";
  cin >> c;

  return 0;
}
```

Глава 31

Динамически подключаемые библиотеки

31.1. Концепция динамически подключаемых библиотек

Динамически подключаемая библиотека (DLL, Dynamic Link Library) представляет собой программный модуль, который может быть загружен в виртуальную память процесса как статически, во время создания исполняемого модуля процесса, так и динамически, во время исполнения процесса операционной системой. Программный модуль, оформленный в виде DLL, хранится на диске в виде файла, который имеет расширение dll, и может содержать как функции, так и данные. Для загрузки DLL в память используется механизм отображения файлов в память.

Динамически подключаемые библиотеки предназначены, главным образом, для разработки функционально-замкнутых библиотек функций, которые могут использоваться разными приложениями. Это позволяет снизить затраты на разработку программного обеспечения, т. к. один и тот же программный код может использоваться разными разработчиками. Кроме того, динамически подключаемые библиотеки позволяют уменьшить объем используемой физической памяти при одновременной работе нескольких приложений, которые используют одну и ту же библиотеку. Это достигается благодаря *механизму проецирования* DLL в виртуальную память процессов, т. к. в этом случае все приложения разделяют один и тот же экземпляр исполняемого кода DLL, загруженный в физическую память.

В связи с последним предложением следует сделать небольшое замечание. Исполняемые файлы и файлы динамических библиотек, т. е. файлы с расширениями exe и dll соответственно разбиты на разделы, каждый из которых содержит данные только определенного типа. Один из этих разделов содержит только исполняемый код приложения или динамически подключаемой библиотеки. Вот этот раздел и хранится в физической памяти в одном

экземпляре и отображается в адресное пространство всех процессов. Те же разделы, которые содержат данные, хранятся для каждого процесса в отдельном экземпляре. Поэтому получается, что процессы совместно разделяют исполняемый код из динамически подключаемой библиотеки, но каждый процесс имеет свой набор переменных из этой библиотеки. Как сделать переменные разделяемыми между процессами можно прочитать, например, в книге Джеффри Рихтера "Windows для профессионалов".

31.2. Создание DLL

Создаются DLL подобно обычным исполняемым модулям, но при этом в среде разработки Visual Studio необходимо выбрать проект типа Win32 Dynamic-Link Library. Как и каждая программа на языке программирования C++, динамически подключаемая библиотека должна иметь главную функцию, которая отмечает точку входа в программу при ее исполнении операционной системой. В отличие от исполняемых модулей, в которых эта функция называется `main`, в DLL главная функция называется `DllMain` и вызывается операционной системой при загрузке DLL в адресное пространство процесса и при создании этим процессом нового потока. Главное назначение функции `DllMain` заключается в инициализации DLL при ее загрузке, а также захвате и освобождении необходимых ресурсов при создании и завершении нового потока в процессе. Эта функция имеет следующий прототип:

```
BOOL  WINAPI  DllMain(
  HINSTANCE  hinstDLL,      // дескриптор DLL
  DWORD      fdwReason,     // флаг причины вызова функции DllMain
  LPVOID     lpvReserved    // зарезервировано Windows
);
```

При успешном завершении функция `DllMain` должна вернуть значение `TRUE`, а в случае неудачи — значение `FALSE`. Параметры функции `DllMain` имеют следующее назначение.

В параметре `hinstDLL` операционная система Windows передает дескриптор DLL, который фактически равен виртуальному адресу, по которому загружена DLL.

Параметр `fdwReason` может иметь одно из следующих значений, которое указывает на причину, по которой операционная система вызывает функцию `DllMain`:

- `DLL_PROCESS_ATTACH` — DLL загружена в адресное пространство процесса;
- `DLL_THREAD_ATTACH` — в процессе создан новый поток и функция `DllMain` вызывается в контексте этого потока;

- DLL_THREAD_DETACH — в процессе завершается поток и функция DllMain вызывается в контексте этого потока;
- DLL_PROCESS_DETACH — DLL выгружается из адресного пространства процесса.

Параметр lpvReserved совместно со значением параметра fdwReason отмечает способ загрузки и выгрузки DLL из адресного пространства процесса.

Если значение параметра fdwReason равно DLL_PROCESS_ATTACH, то значение параметра lpvReserved, равное NULL, указывает на то, что DLL загружается динамически при помощи одной из функций LoadLibrary или LoadLibraryEx. Любое другое значение параметра lpvReserved в этом случае указывает на то, что DLL загружается статически.

Если значение параметра fdwReason равно DLL_PROCESS_DETACH, то значение параметра lpvReserved, равное NULL, указывает на то, что DLL выгружается динамически при помощи функции FreeLibrary. Любое другое значение параметра lpvReserved в этом случае указывает на то, что DLL выгружается при завершении процесса.

Сделаем следующее замечание относительно функции DllMain. Если эта функция не реализована в DLL, то компилятор автоматически сгенерирует эту функцию, которая всегда будет возвращать значение TRUE.

Теперь рассмотрим, как оформляются функции и переменные, которые DLL предоставляет в использование своим клиентам. Во-первых, заметим, что такие функции и переменные называются *экспортируемыми*. Для того чтобы сделать функцию или переменную экспортируемой, нужно определить их с модификатором extern "C" и квалификатором __declspec(dllexport). Кроме того, экспортируемая переменная должна быть инициализирована. Модификатор extern "C" указывает компилятору на то, что функция или переменная должна иметь имя в стиле языка программирования C. То есть имя функции или переменной не будет искажаться путем добавления к нему обозначений типов данных из сигнатуры функции или определения переменной. Модификатор __declspec(dllexport) указывает компилятору на то, что данная функция или переменная будет экспортироваться из DLL.

В листинге 31.1 приведен код DLL, в которой определяются переменная count и две функции Add и Sub. Переменная count исполняет роль счетчика, а функции Add и Sub соответственно прибавляют и вычитают из счетчика некоторое число.

Листинг 31.1. Создание DLL

```
#include <windows.h>

// главная функция
```

```
BOOL WINAPI DllMain(HINSTANCE hDll, DWORD dwReason, LPVOID lpReserved)
{
  switch (dwReason)
  {
  case DLL_PROCESS_ATTACH:   // загрузка DLL
    break;
  case DLL_THREAD_ATTACH:    // создание потока
    break;
  case DLL_THREAD_DETACH:    // завершение потока
    break;
  case DLL_PROCESS_DETACH:   // отключение DLL
    break;
  }

  return TRUE;
}

extern "C" __declspec(dllexport) int count = 0;

extern "C" __declspec(dllexport) int Add(int n)
{
  count += n;

  return count;
}

extern "C" __declspec(dllexport) int Sub(int n)
{
  count -= n;

  return count;
}
```

31.3. Динамическая загрузка и отключение DLL

Для динамической загрузки DLL в виртуальную память процесса используются две функции `LoadLibrary` и `LoadLibraryEx`. Функция `LoadLibraryEx` отличается от функции `LoadLibrary` только тем, что позволяет управлять режимом загрузки библиотеки. Рассмотрим эти функции подробнее.

Для загрузки динамически подключаемой библиотеки используется функция LoadLibrary, которая имеет следующий прототип:

```
HMODULE  LoadLibrary(
  LPCTSTR  lpFileName  // имя файла
);
```

В случае успешного завершения эта функция возвращает дескриптор загруженного модуля, а в случае неудачи — NULL.

Единственный параметр этой функции указывает на строку, содержащую имя загружаемого файла. Если эта строка содержит полный путь к файлу, то именно этот файл и загружается. Если же указано только имя файла, то сначала система определяет, не был ли загружен этот модуль ранее. Если да, то система возвращает дескриптор уже загруженного модуля, в противном случае — для загрузки используется следующий алгоритм поиска.

1. Просматривается каталог, из которого запущено приложение.
2. Просматривается текущий каталог процесса.
3. Просматривается системный каталог Windows.
4. Просматривается каталог Windows.
5. Просматриваются каталоги, которые указаны в переменной окружения PATH.

При этом следует учитывать, что если тип файла не указан, то система по умолчанию считает, что этот файл имеет расширение dll. Для того чтобы отметить, что файл не имеет типа, нужно после имени файла просто указать точку.

Если поиск DLL завершился успехом, то библиотека загружается в виртуальную память процесса, а затем вызывается ее функция DllMain с параметром fdwReason, равным значению DLL_PROCESS_ATTACH. Если эта функция возвращает значение FALSE, то она опять вызывается системой, но со значением параметра fdwReason равным DLL_PROCESS_DETACH. После этого DLL выгружается.

Для загрузки динамически подключаемых библиотек, а также исполняемых модулей используется функция LoadLibraryEx, которая имеет следующий прототип:

```
HMODULE  LoadLibraryEx(
  LPCTSTR  lpFileName,   // имя файла
  HANDLE   hFile,        // зарезервировано
  DWORD    dwFlags       // флаги управления загрузкой
);
```

В случае успешного завершения возвращает дескриптор загруженного модуля, а в случае неудачи — NULL.

Как и в случае с функцией `LoadLibrary`, параметр `lpFileName` должен содержать указатель на имя файла. Поиск загружаемого модуля выполняется по тому же алгоритму, что и функции `LoadLibrary`, при условии, что в параметре `dwFlags` не задан флаг, указывающий на альтернативный алгоритм поиска загружаемого модуля.

Параметр `hFile` зарезервирован для дальнейшего использования системой и должен быть всегда установлен в `NULL`.

Параметр `dwFlags` задает флаги управления загрузкой модуля и может принимать одно из следующих значений:

- `DONT_RESOLVE_DLL_REFERENCES` — в операционных системах Windows NT/2000 после загрузки DLL не вызывается функция `DllMain`, а также не загружаются дополнительные модули, которые требуются для выполнения этой DLL;
- `LOAD_LIBRARY_AS_DATAFILE` — в этом случае модуль загружается как файл данных, при этом не выполняются никакие действия по настройке и вызову программ. Отметим, что, в этом случае операционная система Windows 98 разрешает использовать возвращаемый дескриптор модуля только в функциях управления ресурсами;
- `LOAD_WITH_ALTERED_SEARCH_PATH` — в этом случае при поиске загружаемого модуля в алгоритме поиска не просматривается каталог, из которого запущено приложение.

Для отключения DLL от процесса используется функция `FreeLibrary`, которая имеет следующий прототип:

```
BOOL  FreeLibrary(
  HMODULE  hModule   // дескриптор DLL
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — значение `FALSE`.

В единственном параметре `hModule` этой функции должен быть установлен дескриптор выгружаемого модуля, который был предварительно получен функциями `LoadLibrary` или `LoadLibraryEx`.

Перед выгрузкой DLL из адресного пространства процесса операционная система вызовет функцию `DllMain` с параметром `fdwReason` равным значению `DLL_PROCESS_ATTACH`.

Для одновременного завершения потока и отключения DLL используется функция `FreeLibraryAndExitThread`, которая имеет следующий прототип:

```
VOID  FreeLibraryAndExitThread(
  HMODULE  hModule,      // дескриптор DLL
  DWORD    dwExitCode    // код возврата для потока
);
```

Эта функция эквивалентна последовательному вызову следующих двух функций:

```
FreeLibrary(hModule);
ExitThread(dwExitCode);
```

Вызов этой функции имеет смысл только в потоке динамической библиотеки, который одновременно завершает свою работу и выгружает саму эту библиотеку.

Сделаем следующее замечание о функциях загрузки и отключения DLL. В принципе, функция загрузки одной и той же DLL может вызываться в приложении несколько раз, что имеет смысл в многопоточном приложении. В этом случае при каждом новом вызове функции загрузки одной и той же DLL эта библиотека не загружается вновь, а используется счетчик ссылок на библиотеку, который при каждом вызове функции увеличивается на единицу. Соответственно, при каждом отключении этой DLL счетчик ссылок уменьшается на единицу. Динамически подключаемая библиотека выгружается только в том случае, если счетчик ссылок на эту библиотеку становится равным нулю.

Пример использования некоторых из рассмотренных функций будет приведен в *разд. 31.4*, в котором будет показано использование DLL, разработанной в предыдущем разделе.

31.4. Использование DLL

Если программа использует некоторые функции и переменные из DLL, то говорят, что она *импортирует* их из DLL. Для обеспечения доступа к импортируемым из DLL функциям и переменным используется функция `GetProcAddress`, которая имеет следующий прототип:

```
FARPROC  GetProcAddress(
  HMODULE  hModule,     // дескриптор DLL
  LPCSTR   lpProcName   // имя функции
);
```

В случае успешного завершения возвращает адрес экспортируемой из библиотеки функции, а в случае неудачи `NULL`. Теперь опишем параметры этой функции.

Параметр `hModule` должен содержать дескриптор DLL, который был предварительно получен функцией `LoadLibrary` или `LoadLibraryEx`.

Параметр `lpProcName` должен указывать на имя импортируемой функции или ее порядковый номер. В последнем случае для задания порядкового номера функции лучше всего использовать макрос `MAKEINTRESOURCE(n)`, где `n` задает порядковый номер функции.

Отметим, что функция `GetProcAddress` может также использоваться и для доступа к экспортируемым переменным из DLL. В листинге 31.2 приведена программа, которая импортирует функции из DLL, рассмотренной в *разд. 31.2*.

Листинг 31.2. Импорт функций и переменных из DLL с использованием их имен

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HMODULE  hDll;      // дескриптор DLL
  int (*Add)(int);    // указатель на функцию Add из DLL
  int (*Sub)(int);    // указатель на функцию Sub из DLL
  int *pCount;        // указатель на переменную count из DLL

  // загружаем динамически подключаемую библиотеку
  hDll = LoadLibrary("Count.dll");
  if (!hDll)
  {
    cerr << "Load library failed." << endl;
    return GetLastError();
  }
  // настраиваем адрес переменной count из DLL
  pCount = (int *)GetProcAddress(hDll, "count");
  if (!pCount)
  {
    cerr << "Get variable address failed." << endl;
    return GetLastError();
  }
  // выводим начальное значение переменной count
  cout << "Initial count = " << (*pCount) << endl;
  // настраиваем адреса функций
  Add = (int (*)(int))GetProcAddress(hDll, "Add");
  Sub = (int (*)(int))GetProcAddress(hDll, "Sub");
  if (!Add || !Sub)
  {
    cerr << "Get procedure address failed." << endl;
    return GetLastError();
```

```
  }
  // изменяем значение счетчика
  cout << "count = " << Add(20) << endl;
  cout << "count = " << Sub(15) << endl;
  // отключаем библиотеку
  if (!FreeLibrary(hDll))
  {
    cerr << "Free library failed." << endl;
    return GetLastError();
  }
  // выходим из программы
  cout << "Press any key to exit ";
  cin.get();
  return 0;
}
```

Теперь приведем, в листинге 31.3, эту же программу, но функции и переменные из DLL будут импортироваться, используя их порядковые номера.

Листинг 31.3. Импорт функций и переменных из DLL с использованием их порядковых номеров

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HMODULE  hDll;      // дескриптор DLL
  int (*Add)(int);    // указатель на функцию Add из DLL
  int (*Sub)(int);    // указатель на функцию Sub из DLL
  int *pCount;        // указатель на переменную count из DLL

  // загружаем динамически подключаемую библиотеку
  hDll = LoadLibrary("Count.dll");
  if (!hDll)
  {
    cerr << "Load library failed." << endl;
    return GetLastError();
  }
```

```
  // настраиваем адрес переменной count из DLL
  pCount = (int *)GetProcAddress(hDll, MAKEINTRESOURCE(3));
  if (!pCount)
  {
    cerr << "Get variable address failed." << endl;
    return GetLastError();
  }
  // выводим начальное значение переменной count
  cout << "Initial count = " << (*pCount) << endl;
  // настраиваем адреса функций
  Add = (int (*)(int))GetProcAddress(hDll, MAKEINTRESOURCE(1));
  Sub = (int (*)(int))GetProcAddress(hDll, MAKEINTRESOURCE(2));
  if (!Add || !Sub)
  {
    cerr << "Get procedure address failed." << endl;
    return GetLastError();
  }
  // изменяем значение счетчика
  cout << "count = " << Add(20) << endl;
  cout << "count = " << Sub(15) << endl;
  // отключаем библиотеку
  if (!FreeLibrary(hDll))
  {
    cerr << "Free library failed." << endl;
    return GetLastError();
  }
    // выходим из программы
  cout << "Press any key to exit";
  cin.get();

  return 0;
}
```

Отметим, что в этом случае порядковые номера функций начинаются с единицы, а только потом задаются переменные. В связи с этим сделаем следующее замечание. Хотя использование порядковых номеров функций и переменных в функции `GetProcAddress` и позволяет выполнять эту функцию быстрее, но использование символических имен более надежно, и, следовательно, более предпочтительно.

В заключение этого раздела сделаем важное замечание. Для того чтобы система смогла найти используемую DLL, ее нужно поместить в каталог, из которого запускается приложение.

31.5. Использование файла определений

Для определения экспортируемых из DLL функций и переменных можно использовать файл определения модуля, который должен иметь расширение def. Такие файлы включают инструкции, которые используются компоновщиком для создания исполняемого модуля или DLL. Однако эти файлы не обязательны для компоновки, т. к. большинство их параметров может быть задано при помощи определения опций компоновщика. При создании DLL def-файл содержит инструкции, которые определяют имена экспортируемых функций и переменных. Отметим, что если для описания экспорта из DLL используется файл определений, то квалификатор `__declspec(dllexport)` перед именами функций и переменных в самой DLL использовать не нужно. Для описания экспорта из DLL через файл определений достаточно использовать инструкции двух типов: `LIBRARY` и `EXPORTS`. Кратко опишем синтаксис этих инструкций.

Инструкция `LIBRARY` говорит компоновщику LINK о том, что создается динамическая библиотека. Эта инструкция имеет следующий синтаксис:

```
LIBRARY   [library]  [BASE = address]
```

Параметр `library` обозначает имя библиотеки. Это имя не используется как имя DLL модуля, а служит только для внутреннего именования библиотеки при сборке модуля.

Параметр `base` обозначает базовый адрес, начиная с которого операционная система будет загружать DLL. По умолчанию операционная система загружает DLL по адресу `0x10000000`.

Инструкция `EXPORTS` определяет экспортируемые имена. Эта инструкция имеет следующий синтаксис:

```
EXPORTS   entryname [ = internalname]  [@ordinal  [NONAME] ]
          [PRIVATE]  [DATA]
```

Параметр `entryname` в этой инструкции обозначает имя экспортируемой функции или переменной. Если это имя отличается от внутреннего имени функции или переменной в DLL, то это внутреннее имя задается параметром `internalname`, который не является обязательным.

Все остальные параметры также не являются обязательными.

Параметр `@ordinal` задает порядковый номер экспортируемого имени.

Атрибут `NONAME` указывает на то, что в таблице экспорта не будет храниться данное экспортируемое имя. В этом случае импорт адреса возможен только по его порядковому номеру.

Ключевое слово PRIVATE запрещает размещение экспортируемого имени в библиотеке импорта. О библиотеках импорта будет рассказано более подробно в следующем разделе.

Атрибут DATA указывает на то, что экспортируемое имя является именем переменной.

Сделаем следующее замечание относительно использования файлов определения модуля. Эти файлы удобно использовать тогда, когда DLL содержит большое число экспортируемых функций и переменных. В этом случае поиск экспортируемых имен занимает очень много времени. Поэтому для более быстрой работы приложения функции и переменные импортируются по их порядковым номерам. Чтобы сделать эти порядковые номера фиксированными, т. е. независимыми от расположения функции или переменной в DLL, используют файл определения модуля.

Теперь приведем пример использования файла определения модуля для создания DLL, которая была рассмотрена в начале главы. В этом случае текст самой DLL будет выглядеть так, как это показано в листинге 31.4. Файл определения модуля для этой DLL приведен в листинге 31.5.

Листинг 31.4. Создание DLL

```
#include <windows.h>

// главная функция
BOOL WINAPI DllMain(HINSTANCE hDll, DWORD dwReason, LPVOID lpReserved)
{
  switch (dwReason)
  {
  case DLL_PROCESS_ATTACH:   // загрузка DLL
    break;
  case DLL_THREAD_ATTACH:    // создание потока
    break;
  case DLL_THREAD_DETACH:    // завершение потока
    break;
  case DLL_PROCESS_DETACH:   // отключение DLL
    break;
  }

  return TRUE;
}
```

```
extern "C" int count = -5;

extern "C" int Add(int n)
{
  count += n;

  return count;
}

extern "C" int Sub(int n)
{
  count -= n;

  return count;
}
```

Листинг 31.5. Файл определения модуля для динамически подключаемой библиотеки Count

```
LIBRARY  "Count"

EXPORTS
    count  @1  DATA
    Add  @2
    Sub  @3
```

Сделаем несколько замечаний относительно работы с файлом определения модуля. Во-первых, такой файл может быть набран в любом текстовом редакторе. Во-вторых, этот файл должен иметь расширение def. И в-третьих, перед компоновкой модуля необходимо подключить этот файл к проекту в среде Visual Sudio.

В листинге 31.6 приведена программа, использующая созданную DLL. Эта программа отличается от программы 31.3 только порядковыми номерами переменной и функций, импортируемых из DLL.

Листинг 31.6. Использование DLL

```
#include <windows.h>
#include <iostream.h>

int main()
```

```
{
  HMODULE  hDll;      // дескриптор DLL
  int (*Add)(int);    // указатель на функцию Add из DLL
  int (*Sub)(int);    // указатель на функцию Sub из DLL
  int *pCount;        // указатель на переменную count из DLL

  // загружаем динамически подключаемую библиотеку
  hDll = LoadLibrary("Count.dll");
  if (!hDll)
  {
    cerr << "Load library failed." << endl;
    return GetLastError();
  }
  // настраиваем адрес переменной count из DLL
  pCount = (int *)GetProcAddress(hDll, MAKEINTRESOURCE(1));
  if (!pCount)
  {
    cerr << "Get variable address failed." << endl;
    return GetLastError();
  }
  // выводим начальное значение переменной count
  cout << "Initial count = " << (*pCount) << endl;
  // настраиваем адреса функций
  Add = (int (*)(int))GetProcAddress(hDll, MAKEINTRESOURCE(2));
  Sub = (int (*)(int))GetProcAddress(hDll, MAKEINTRESOURCE(3));
  if (!Add || !Sub)
  {
    cerr << "Get procedure address failed." << endl;
    return GetLastError();
  }
  // изменяем значение счетчика
  cout << "count = " << Add(20) << endl;
  cout << "count = " << Sub(15) << endl;
  // отключаем библиотеку
  if (!FreeLibrary(hDll))
  {
    cerr << "Free library failed." << endl;
    return GetLastError();
  }
```

```
  // выходим из программы
  cout << "Press any key to exit";
  cin.get();
  return 0;
}
```

31.6. Статическая загрузка DLL

Статическая загрузка DLL отличается от динамической тем, что выполняется на этапе компоновки исполняемого модуля, а не на этапе его исполнения. Для этого компоновщик использует *библиотеку импорта*. Библиотека импорта — это файл с расширением lib, который создается компоновщиком при создании динамической библиотеки. В библиотеке импорта содержатся ссылки на все экспортируемые из динамической библиотеки имена. Для статической загрузки DLL необходимо выполнить следующую последовательность действий.

1. Создать любым из вышеуказанных способов DLL.
2. Поместить саму библиотеку и файл импорта этой библиотеки в каталог, из которого запускается приложение.
3. В меню **Project** выбрать пункт **Settings**, на котором выбрать закладку **Link**. После этого в окне **Object/library modules** ввести имя используемой библиотеки импорта.
4. Описать импортируемые из DLL имена в приложении.

Отметим, что импортируемые из DLL имена должны иметь модификатор `extern` и квалификатор `__declspec(dllimport)`. Использование квалификатора `__declspec(dllimport)` с прототипами функций не обязательно, но его нужно обязательно использовать при импорте имен переменных. Однако использование этого квалификатора с именами функций делает описание функций более понятным и ускоряет работу компоновщика.

В листинге 31.7 приведена программа, которая использует разработанную ранее DLL, при условии, что эта DLL загружается статически.

Листинг 31.7. Использование DLL

```
#include <windows.h>
#include <iostream.h>

extern "C" __declspec(dllimport) int count;
```

```
extern "C" __declspec(dllimport) int Add(int n);
extern "C" __declspec(dllimport) int Sub(int n);

int main()
{
  // выводим начальное значение переменной count
  cout << "Initial count = " << count << endl;
  // изменяем значение счетчика
  cout << "count = " << Add(20) << endl;
  cout << "count = " << Sub(15) << endl;

  // выходим из программы
  cout << "Press any key to exit";
  cin.get();

  return 0;
}
```

Глава 32

Локальная память потока

32.1. Динамическая локальная память потока

Представим себе такую ситуацию: разрабатывается DLL, одна из функций которой динамически захватывает память для обслуживания приложения. Если эту DLL использует только один поток приложения, то для сохранения указателя на динамически распределенную память достаточно определить в DLL одну переменную. Теперь допустим, что эта DLL используется несколькими потоками одного приложения. В этом случае возникают следующие проблемы. Во-первых, нужно где-то хранить указатели на захваченную память, ведь заранее неизвестно, сколько потоков будут параллельно использовать DLL. Во-вторых, функция из DLL должна как-то узнать — какой поток ее вызывает, чтобы сопоставить каждому потоку свой указатель. Для решения подобных задач и предназначена динамическая локальная память потока.

Динамическая локальная память потока (thread local storage, TLS) представляет собой массив указателей, доступ к которым осуществляется через индексы при помощи специальных функций Win32 API. Этот массив указателей автоматически поддерживается операционной системой для каждого потока и содержит, по меньшей мере, `TLS_MINIMUM_AVAILABLE` указателей. Порядок работы с локальной памятью потока следующий:

- распределение указателя;
- работа с указателем;
- освобождение указателя.

Прежде чем использовать какой-либо указатель, его нужно распределить. Это делается при помощи функции `TlsAlloc`. После распределения указателя для каждого потока становится доступной своя копия этого указателя.

Для работы с указателями локальной памяти потока используются функции TlsSetValue и TlsGetValue, которые соответственно записывают некоторое значение и читают значение из локальной памяти потока. После окончания работы с указателем его нужно освободить, вызвав функцию TlsFree.

В следующих далее разделах будет описана подробно работа со всеми этими функциями.

32.2. Распределение и освобождение локальной памяти потока

Прежде чем использовать какой-либо указатель из локальной памяти потока, его нужно распределить. Для этой цели используется функция TlsAlloc, которая имеет следующий прототип:

```
DWORD  TlsAlloc(VOID);
```

Эта функция не имеет параметров и в случае успешного завершения возвращает индекс распределенного указателя из локальной памяти потока. При этом распределенный указатель инициализирован в ноль. Если функция закончилась неудачей, то она возвращает -1. Неудача при выполнении этой функции, скорее всего, означает, что локальная память потока исчерпана.

Для освобождения распределенного указателя из локальной памяти потока используется функция TlsFree, которая имеет следующий прототип:

```
BOOL  TlsFree(
  DWORD  dwTlsIndex    // TLS-индекс
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — FALSE. Единственным параметром этой функции является индекс освобождаемого указателя из локальной памяти потока.

Примеры вызова этих функций приведены в следующем разделе, который содержит программы, иллюстрирующие работу с динамической локальной памятью потока.

32.3. Запись и чтение из локальной памяти потока

Для записи значения в локальную память потока используется функция TlsSetValue, которая имеет следующий прототип:

```
BOOL  TlsSetValue(
  DWORD   dwTlsIndex,  // TLS-индекс
```

```
    LPVOID  lpTlsValue   // запоминаемое значение
  );
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — FALSE.

Параметр dwTlsIndex этой функции должен содержать индекс элемента в массиве указателей, а само значение, запоминаемое в этом элементе массива, передается через параметр lpTlsValue.

Для чтения значения из динамической локальной памяти потока используется функция TlsGetValue, которая имеет следующий прототип:

```
  LPVOID  TlsGetValue(
    DWORD  dwTlsIndex    // TLS-индекс
  );
```

В случае успешного завершения эта функция возвращает значение, хранимое в локальной памяти потока с индексом dwTlsIndex, а в случае неудачи — ноль.

В листинге 32.1 приведена программа, которая иллюстрирует работу с рассмотренными функциями. Эта программа распределяет все указатели из локальной памяти потока, а затем освобождает их. При этом для запоминания индексов распределенных указателей используются сами указатели.

Листинг 32.1. Работа с динамической локальной памятью потока

```
#include <windows.h>
#include <iostream.h>

DWORD thread(void)
{
  char  c;
  DWORD  dwFirstTlsIndex, dwPrevTlsIndex, dwNextTlsIndex;

  // распределяем первый TLS-индекс
  dwFirstTlsIndex = TlsAlloc();
  if (dwFirstTlsIndex == -1)
  {
    cerr << "Tls allocation failed." << endl;
    return GetLastError();
  }
  // устанавливаем предыдущий TLS-индекс равным первому индексу
  dwPrevTlsIndex = dwFirstTlsIndex;
```

```
// двигаемся дальше по TLS-индексам
for (;;)
{
  // выводим на экран предыдущий TLS-индекс
  cout << "Alloc tls index = " << dwPrevTlsIndex << endl;
  Sleep(50);
  // распределяем следующий TLS-индекс
  dwNextTlsIndex = TlsAlloc();
  // если больше индексов нет, то выходим из цикла
  if (dwNextTlsIndex == -1)
    break;
  // запомним следующий TLS-индекс в предыдущем индексе
  if (!TlsSetValue(dwPrevTlsIndex, (LPVOID)dwNextTlsIndex))
  {
    cerr << "Tls set value failed." << endl;
    return GetLastError();
  }
  // продвигаем индексы
  dwPrevTlsIndex = dwNextTlsIndex;
}

// ждем команду на освобождение индексов
cout << "Input any char to free tls indexes: ";
cin >> c;

// устанавливаем предыдущий TLS-индекс в первый
dwPrevTlsIndex = dwFirstTlsIndex;
  // двигаемся дальше по tls индексам
for (;;)
{
  // выводим на экран предыдущий TLS-индекс
  cout << "Free tls index = " << dwPrevTlsIndex << endl;
  Sleep(50);
  // получаем следующий TLS-индекс
  dwNextTlsIndex = (DWORD)TlsGetValue(dwPrevTlsIndex);
  if (!dwNextTlsIndex)
    break;
  // освобождаем предыдущий TLS-индекс
  if (!TlsFree(dwPrevTlsIndex))
```

```
    {
      cerr << "Tls free failed." << endl;
      return GetLastError();
    }
    // продвигаем индексы
    dwPrevTlsIndex = dwNextTlsIndex;
  }

  // ждем команду на выход из потока
  cout << "Input any char to exit: ";
  cin >> c;

  return 0;
}

int main()
{
  HANDLE  hThread;
  DWORD  IDThread;

  // запускаем поток
  hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread,
              NULL, 0, &IDThread);
  if (hThread == NULL)
  {
    cerr << "Create thread failed." << endl;
    return GetLastError();
  }
  // ждем, пока поток Add закончит работу
  WaitForSingleObject(hThread, INFINITE);
  // закрываем дескриптор потока
  CloseHandle(hThread);

  return 0;
}
```

Теперь приведем пример использования локальной памяти потока в DLL. Для этого создадим DLL, в которой всего одна функция `Add`. Эта функция увеличивает значение отдельного счетчика для каждого потока, при этом

счетчики создаются динамически. Так как в DLL заранее неизвестно, сколько потоков будут использовать эту функцию, то для хранения указателей на счетчики используется локальная память потоков. В листинге 32.2 приведен текст программы, в которой создается DLL.

Листинг 32.2. Пример DLL, использующей динамическую память потока

```
#include <windows.h>
#include <iostream.h>

DWORD  dwTlsIndex;   // индекс динамической локальной памяти потока

// главная функция
BOOL WINAPI DllMain(HINSTANCE hDll, DWORD dwReason, LPVOID lpReserved)
{
  BOOL  retVal = TRUE;   // возвращаемое значение
  int    *pCount;        // указатель на счетчик

  switch (dwReason)
  {
  case DLL_PROCESS_ATTACH:   // загрузка DLL
    // распределяем локальную память потоков
    dwTlsIndex = TlsAlloc();
    if (dwTlsIndex == -1)
    {
      cerr << "Tls allocation failed." << endl;
      retVal = FALSE;
    }
    break;
  case DLL_THREAD_ATTACH:    // создание потока
    // захватываем память под счетчик
    pCount = new int(0);
    // запоминаем указатель в индексе
    if (!TlsSetValue(dwTlsIndex, (LPVOID)pCount))
    {
      cerr << "Tls set value failed." << endl;
      retVal = FALSE;
    }
    break;
  case DLL_THREAD_DETACH:    // завершение потока
```

```
    // получаем указатель из индекса
    pCount = (int*)TlsGetValue(dwTlsIndex);
    // если ошибка, то выдаем сообщение
    if (!pCount)
    {
      cerr << "Tls get value failed." << endl;
      retVal = FALSE;
    }
    // иначе освобождаем память
    else
      delete pCount;
    break;
  case DLL_PROCESS_DETACH:  // отключение DLL
    // освобождаем локальную память потока
    if (!TlsFree(dwTlsIndex))
    {
      cerr << "Tls free failed." << endl;
      retVal = FALSE;
    }
    break;
  }

  return retVal;
}

extern "C" __declspec(dllexport) int Add(int n)
{
  int  *pCount;

  // получаем указатель на локальный счетчик
  pCount = (int*)TlsGetValue(dwTlsIndex);
  if (!pCount)
  {
    cerr << "Tls get value failed." << endl;
    return GetLastError();
  }
  // увеличиваем значение счетчика
  *pCount += n;

  return *pCount;
}
```

Теперь в листинге 32.3 приведем программу, которая использует DLL, созданную в предыдущем примере. В этой программе функцию `Add` вызывают два создаваемых потока.

Листинг 32.3. Использование DLL в многопоточном приложении

```
#include <windows.h>
#include <iostream.h>

int (*Add)(int);   // указатель на функцию Add из DLL

DWORD WINAPI thread(LPVOID iNum)
{
  for (int i = 0; i < (int)iNum; ++i)
  {
    cout << "count = " << Add((int)iNum) << endl;
    Sleep(15);
  }

  return 0;
}

int main()
{
  char  c;
  HMODULE  hDll;       // дескриптор DLL
  HANDLE  hThread[2];  // дескрипторы потоков
  DWORD  IDThread[2];  // идентификаторы потоков

  // загружаем динамически подключаемую библиотеку
  hDll = LoadLibrary("Count.dll");
  if (!hDll)
  {
    cerr << "Load library failed." << endl;
    return GetLastError();
  }
  // настраиваем адрес функци
  Add = (int (*)(int))GetProcAddress(hDll, "Add");
  if (!Add)
```

```
  {
    cerr << "Get procedure address failed." << endl;
    return GetLastError();
  }
  // запускаем потоки
  hThread[0] = CreateThread(NULL, 0, thread, (void*)3, 0, &IDThread[0]);
  hThread[1] = CreateThread(NULL, 0, thread, (void*)5, 0, &IDThread[1]);
  if (hThread[0] == NULL || hThread[1] == NULL)
  {
    cerr << "Create thread failed." << endl;
    return GetLastError();
  }
  // ждем завершения потоков
  WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
  // отключаем библиотеку
  if (!FreeLibrary(hDll))
  {
    cerr << "Free library failed." << endl;
    return GetLastError();
  }
  // выходим из программы
  cout << "Input any char to exit: ";
  cin >> c;

  return 0;
}
```

32.4. Статическая локальная память потока

Если DLL загружается статически, то динамический вызов функций является не лучшим подходом к работе с локальной памятью потока, т. к. замедляет работу приложения. В случае статической загрузки DLL информация о необходимой локальной памяти для каждого потока может быть получена на этапе компиляции DLL. Для этого в компиляторе Visual C++ фирмы Microsoft введен спецификатор памяти `declspec(thread)`, который может использоваться только с глобальными или статическими переменными. Естественно, что этот спецификатор может использоваться и в обычных программах, но там его использование не имеет смысла, т. к. в этом случае в каждом потоке можно определить локальные переменные как в обычных функциях. Если спецификатор `declspec(thread)` используется при объявле-

нии некоторой переменной, то система создает отдельный экземпляр этой переменной для каждого потока приложения, в котором используется эта переменная. Память, выделяемая под такие переменные, называется *статической локальной памятью потока*.

В листинге 32.4 приведен текст рассмотренной в предыдущем разделе DLL, в которой счетчик объявлен глобально, используя локальную статическую память потока.

Листинг 32.4. Пример DLL, использующей статическую память потока

```
#include <windows.h>

__declspec(thread) int count = 0;

// главная функция
BOOL WINAPI DllMain(HINSTANCE hDll, DWORD dwReason, LPVOID lpReserved)
{
  switch (dwReason)
  {
  case DLL_PROCESS_ATTACH:   // загрузка DLL
    break;
  case DLL_THREAD_ATTACH:    // создание потока
    break;
  case DLL_THREAD_DETACH:    // завершение потока
    break;
  case DLL_PROCESS_DETACH:   // отключение DLL
    break;
  }

  return TRUE;
}

extern "C" __declspec(dllexport) int Add(int n)
{
  count += n;

  return count;
}
```

Теперь в листинге 32.5 приведем текст многопоточного приложения, которое использует эту DLL. Отметим, что в этом приложении один экземпляр

счетчика используется главным потоком, а второй экземпляр — потоком, создаваемым динамически.

Листинг 32.5. Использование DLL в многопоточном приложении

```
#include <windows.h>
#include <iostream.h>

extern "C" __declspec(dllimport) int Add(int n);

DWORD WINAPI thread(LPVOID iNum)
{
  cout << "count from thread = " << Add((int)iNum) << endl;

  return 0;
}

int main()
{
  HANDLE  hThread;      // дескрипторы потоков
  DWORD   IDThread;     // идентификаторы потоков

  // изменяем значение счетчика
  cout << "count from main = " << Add(20) << endl;

  // запускаем поток
  hThread = CreateThread(NULL, 0, thread, (void*)10, 0, &IDThread);
  if (hThread == NULL)
  {
    cerr << "Create thread failed." << endl;
    return GetLastError();
  }
  // ждем завершения потока
  WaitForSingleObject(hThread, INFINITE);
  // выходим из программы
  cout << "Press any key to exit";
  cin.get();

  return 0;
}
```

В заключение этого раздела отметим, что локальная память потоков не работает в DLL, которые подключаются динамически.

Часть X

Разработка сервисов в Windows

Глава 33

Сервисы в Windows

33.1. Концепция сервиса

Сервис это процесс, который выполняет служебные функции. Сервисы являются аналогами резидентных программ, которые использовались в операционных системах, предшествующих операционной системе Windows NT. То есть сервис это такая программа, которая запускается при загрузке операционной системы или в процессе ее работы по специальной команде и заканчивает свою работу при завершении работы операционной системы или по специальной команде. Обычно сервисы выполняют определенные служебные функции, необходимые для работы приложений или какого-то конкретного приложения. Примером сервиса может служить фоновый процесс, который обеспечивает доступ к базе данных — такие сервисы также называются серверами. Другой тип сервисов — это программы, обеспечивающие доступ к внешним устройствам, такие сервисы называются драйверами. Как сервис также может быть реализован процесс, отслеживающий работу некоторого приложения, такие сервисы также называются мониторами.

Сервисы реализованы только на платформах Windows NT/2000/XP. Управляет работой сервисов специальная программа операционной системы, которая называется менеджер сервисов (Service Control Manager, SCM). Ниже перечислены функции, которые выполняет менеджер сервисов:

- поддержка базы данных установленных сервисов;
- запуск сервисов при загрузке операционной системы;
- поддержка информации о состоянии работающих сервисов;
- передача управляющих запросов работающим сервисам;
- блокировка и разблокирование базы данных сервисов.

Для того чтобы менеджер сервисов знал о существовании определенного сервиса — его нужно установить. Информация обо всех установленных сервисах

хранится в реестре операционной системы Windows под ключом HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services. Сервис может запускаться как операционной системой при загрузке, так и программно — из приложения. Если сервис больше не нужен, то его нужно удалить из базы данных операционной системы. Для работы с сервисами в операционных системах Windows предназначены специальные функции из Win32 API. Управлять работой сервисов можно также и через панель управления, а именно **Control Panel | Administrative Tools | Services** (Панель управления | Администрирование | Службы). В этой главе будут рассмотрены только программные средства для управления сервисами.

В заключение этого раздела скажем об отладке сервисов. Так как сервисы запускаются операционной системой, то невозможно выполнить отладку сервиса как обычного консольного приложения. В этом случае можно использовать отладчик так, как это описано в справочной системе MSDN (в разделе **Windows Development | Windows Base Services | DLL, Processes and Threads | SDK Documentation | DLL, Processes and Threads | Services | About Services | Debugging a Service**). Но можно использовать для отладки сервиса старые методы отладки программ. А именно, создать так называемый лог-файл и записывать в него протокол работы сервиса и его состояние, включающее контекст памяти, в контрольных точках.

33.2. Структура сервиса

Так как сервисы работают под управлением менеджера сервисов, то они должны удовлетворять определенным соглашениям, которые определяют интерфейс сервиса. Сам сервис может быть как консольным приложением, так и приложением с графическим интерфейсом. Это не имеет значения, т. к. сервисы могут взаимодействовать с пользователем только через рабочий стол или через окно сообщений. Поэтому обычно сервисы оформляются как консольные приложения. Кроме того, каждый сервис должен содержать две функции обратного вызова, которые вызываются операционной системой. Одна из этих функций определяет точку входа сервиса, т. е., собственно, и является сервисом, а вторая — должна реагировать на управляющие сигналы от операционной системы. Отсюда следует, что если консольное приложение определяет один сервис, то оно должно иметь следующую структуру:

```
// главная функция консольного приложения
int main(int  argc, char  *argv) { … }

// точка входа сервиса
VOID WINAPI ServiceMain(DWORD  dwArgc, LPTSTR  *lpszArgv) { … }

// обработчик запросов
VOID WINAPI ServiceCtrlHandler(DWORD  dwControl) { … }
```

Конечно, допускается присутствие и других функций, но эти три функции должны быть обязательно.

Главной задачей функции `main` является запуск *диспетчера сервиса*, который является потоком и управляет этим сервисом. Диспетчер сервиса получает управляющие сигналы от менеджера сервисов по именованному каналу и передает эти запросы функции `ServiceCtrlHandler`, которая обрабатывает эти управляющие запросы. Если в приложении несколько сервисов, то для каждого сервиса запускается свой диспетчер и для каждого диспетчера определяется своя функция обработки управляющих запросов, которая выполняется в контексте соответствующего диспетчера сервисов. Запуск диспетчеров сервисов выполняется при помощи вызова функции `StartServiceCtrlDispatcher`. Для подключения обработчика запросов к сервису используется функция `RegisterServiceCtrlHandler`.

Функция, определяющая точку входа сервиса, должна иметь следующий прототип:

```
VOID WINAPI имя_точки_входа(DWORD  dwArgc, LPTSTR  *lpszArgv);
```

Если определяется только один сервис, то эта функция обычно называется `ServiceMain`, хотя возможны и другие, более подходящие по смыслу имена точек входа сервисов. Если же в приложении определяется несколько сервисов, то естественно каждый из них должен иметь свое имя. Эта функция содержит два параметра, которые аналогичны параметрам функции `main` консольного приложения. Параметр `dwArgc` содержит количество аргументов в массиве `lpszArgv`, а сам этот массив содержит адреса строк. Причем первый из этих адресов указывает на строку, содержащую имя сервиса, а последующие аргументы передаются из функции `StartService`.

Функция, определяющая обработчик управляющих запросов, должна иметь следующий прототип:

```
VOID WINAPI имя_обработчика_запросов(DWORD  dwControl);
```

Если определяется только один сервис, то эта функция обычно называется `ServiceCtrlHandler`. Если же в приложении определяется несколько сервисов, то естественно, что обработчик запросов для каждого сервиса должен иметь свое имя. Эта функция содержит только один параметр, который содержит код управляющего сигнала. Более подробно возможные коды сигналов будут рассмотрены в *разд. 33.5*, посвященном обработчику запросов. Так как обработчик запросов вызывается диспетчером сервиса, то и коды управляющих сигналов он получает от своего диспетчера.

33.3. Организация функции *main*

Как уже было сказано, главной задачей функции `main` является запуск диспетчера сервиса для каждого из сервисов. Для запуска диспетчера используется функция `StartServiceCtrlDispatcher`, которая должна быть вызвана

в течение 30 секунд с момента запуска программы `main`. Если в течение этого промежутка времени функция `StartServiceCtrlDispatcher` вызвана не будет, то последующий вызов этой функции закончится неудачей. Поэтому всю необходимую инициализацию сервиса нужно делать в самом сервисе, т. е. в теле функции `ServiceMain`. Если же необходимо выполнить инициализацию глобальных переменных, то для этой цели лучше запустить из функции `main` отдельный поток.

Функция `StartServiceCtrlDispatcher` имеет следующий прототип:

```
BOOL  StartServiceCtrlDispatcher(
  CONST LPSERVICE_TABLE_ENTRY  lpServiceTable    // таблица сервисов
);
```

В случае успешного завершения функция `StartServiceCtrlDispatcher` возвращает ненулевое значение, а в случае неудачи — `FALSE`. Функция имеет единственный параметр, который указывает на таблицу сервисов, которая представляет собой массив. Каждый элемент этого массива имеет следующий тип:

```
typedef  struct  _SERVICE_TABLE_ENTRY {
  LPSTR  lpServiceName;                    // имя сервиса
  LPSERVICE_MAIN_FUNCTION  lpServiceProc;  // функция сервиса
} SERVICE_TABLE_ENTRY,  *LPSERVICE_TABLE_ENTRY;
```

Поле `lpServiceName` содержит указатель на строку, содержащую имя сервиса, а поле `lpServiceProc` указывает на функцию самого сервиса. При этом отметим, что последний элемент таблицы сервисов должен содержать пустую структуру, т. е. структуру с пустыми указателями.

Ниже в листинге 33.1 приведен пример программы `main`, которая запускает диспетчер для одного сервиса. Перед телом функции объявлены все глобальные имена, которые в дальнейшем используются в сервисе.

Листинг 33.1. Пример функции `main` сервиса

```
#include <windows.h>
#include <fstream.h>

char  service_name[] = "DemoService";    // имя сервиса

SERVICE_STATUS    service_status;
SERVICE_STATUS_HANDLE  hServiceStatus;

VOID WINAPI ServiceMain(DWORD dwArgc, LPTSTR *lpszArgv);
```

```
VOID WINAPI ServiceCtrlHandler(DWORD dwControl);

ofstream  out;   // выходной файл для протокола работы сервиса
int  nCount;     // счетчик

// главная функция приложения
int main()
{
  //  инициализируем структуру сервисов
  SERVICE_TABLE_ENTRY  service_table[] =
  {
    {service_name, ServiceMain},   // имя сервиса и функция сервиса
    { NULL, NULL }                 // больше сервисов нет
  };

  // запускаем диспетчер сервиса
  if (!StartServiceCtrlDispatcher(service_table))
  {
    out.open("C:\\ServiceFile.log");
    out << "Start service control dispatcher failed.";
    out.close();

    return 0;
  }

  return 0;
}
```

33.4. Организация функции *ServiceMain*

Как уже говорилось, функция, определяющая точку входа сервиса, должна иметь следующий прототип:

```
VOID WINAPI имя_точки_входа(DWORD  dwArgc, LPTSTR  *lpszArgv);
```

Если в приложении определен только один сервис, то эта функция обычно называется `ServiceMain`. В противном случае точка входа каждого сервиса должна иметь уникальное имя. Здесь параметр `dwArgc` содержит количество аргументов в массиве `lpszArgv`, а сам этот массив содержит адреса строк. Причем первый из этих адресов указывает на строку, содержащую имя сервиса, а последующие аргументы передаются из функции `StartService`.

Функция ServiceMain должна выполнить следующую последовательность действий:

1. Немедленно запустить обработчик управляющих команд от менеджера сервисов, вызвав функцию RegisterServiceCtrlHandler.
2. Установить стартующее состояние сервиса SERVICE_START_PENDING посредством вызова функции SetServiceStatus.
3. Провести локальную инициализацию сервиса.
4. Установить рабочее состояние сервиса SERVICE_RUNNING посредством вызова функции SetServiceStatus.
5. Выполнять работу сервиса, учитывая состояния сервиса, которые могут изменяться обработчиком управляющих команд от менеджера сервисов.
6. После перехода в состояние останова SERVICE_STOPPED выполнить освобождение захваченных ресурсов и закончить работу.

Опишем подробно функции, которые должны использоваться в сервисе.

Для запуска обработчика управляющих команд от менеджера сервисов сервис должен использовать функцию RegisterServiceCtrlHandler, которая имеет следующий прототип:

```
SERVICE_STATUS_HANDLE  RegisterServiceCtrlHandler(
  LPCTSTR  lpServiceName,            // имя сервиса
  LPHANDLER_FUNCTION  lpHandlerProc  // функция обработчика
);
```

В случае успешного завершения функция возвращает дескриптор состояния сервиса, а в случае неудачи — ноль. Параметр lpServiceName этой функции должен указывать на строку, которая содержит имя сервиса, а параметр lpHandlerProc — на процедуру обработки управляющих команд от менеджера сервисов.

Для запуска обработчика управляющих команд может также использоваться функция RegisterServiceCtrlHandlerEx, которая имеет более широкие возможности по сравнению с функцией RegisterServiceCtrlHandler.

Для изменения состояния сервиса используется функция SetServiceStatus, которая имеет следующий прототип:

```
BOOL  SetServiceStatus(
  SERVICE_STATUS_HANDLE hServiceStatus, // дескриптор состояния сервиса
  LPSERVICE_STATUS      lpServiceStatus // структура состояния
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — FALSE.

Параметр `hServiceStatus` должен содержать дескриптор состояния сервиса, который был получен функцией `RegisterServiceCtrlHandler`.

Параметр `lpServiceStatus` должен указывать на структуру типа `SERVICE_STATUS`, которая содержит информацию о состоянии сервиса. Эта структура имеет следующий формат:

```
typedef  struct  _SERVICE_STATUS {
  DWORD  dwServiceType;        // тип сервиса
  DWORD  dwCurrentState;       // текущее состояние сервиса
  DWORD  dwControlsAccepted;   // допускаемое управление
  DWORD  dwWin32ExitCode;      // код возврата для Win32
  DWORD  dwServiceSpecificExitCode;  // специфический код возврата
                                     // сервиса
  DWORD  dwCheckPoint;        // контрольная точка
  DWORD  dwWaitHint;          // период ожидания команды управления
}  SERVICE_STATUS,  *LPSERVICE_STATUS;
```

Поле `dwServiceType` содержит тип сервиса и может принимать следующие значения:

- `SERVICE_WIN32_OWN_PROCESS` — сервис является самостоятельным процессом;
- `SERVICE_WIN32_SHARE_PROCESS` — сервис разделяет процесс с другими сервисами;
- `SERVICE_KERNEL_DRIVER` — сервис является драйвером устройства;
- `SERVICE_FILE_SYSTEM_DRIVER` — сервис является драйвером файловой системы.

Кроме того, первые два флага могут быть установлены совместно с флагом `SERVICE_INTERACTIVE_PROCESS` — сервис может взаимодействовать с рабочим столом. То есть сервисы, которые не являются драйверами, могут взаимодействовать с рабочим столом.

Поле `dwCurrentState` содержит текущее состояние сервиса. Это поле может принимать одно из следующих значений:

- `SERVICE_STOPPED` — сервис остановлен;
- `SERVICE_START_PENDING` — сервис стартует;
- `SERVICE_STOP_PENDING` — сервис останавливается;
- `SERVICE_RUNNING` — сервис работает;
- `SERVICE_CONTINUE_PENDING` — сервис переходит в рабочее состояние;
- `SERVICE_PAUSE_PENDING` — сервис переходит в состояние ожидания;
- `SERVICE_PAUSED` — сервис находится в состоянии ожидания.

Поле dwControlsAccepted содержит коды управляющих команд, которые могут быть переданы обработчику этих команд, определенному в приложении. В этом поле может быть установлена любая комбинация следующих флагов:

- SERVICE_ACCEPT_STOP — можно остановить сервис;
- SERVICE_ACCEPT_PAUSE_CONTINUE — можно приостановить и возобновить сервис;
- SERVICE_ACCEPT_SHUTDOWN — сервис информируется о выключении системы.

Кроме того, в операционной системе Windows 2000 в этом поле может быть дополнительно установлена любая комбинация следующих флагов:

- SERVICE_ACCEPT_PARAMCHANGE — сервис может вновь прочитать свои параметры;
- SERVICE_ACCEPT_NETBINDCHANGE — сервис является сетевой компонентой;
- SERVICE_ACCEPT_HARDWAREPROFILECHANGE — сервер информируется об изменении аппаратного обеспечения компьютера;
- SERVICE_ACCEPT_POWEREVENT — сервис информируется об изменении состояния питания компьютера.

Если установлен флаг SERVICE_ACCEPT_NETBINDCHANGE, то обработчик управляющих команд сервиса может также получить следующие команды:

- SERVICE_CONTROL_NETBINDADD — к сети подключен новый компонент;
- SERVICE_CONTROL_NETBINDREMOVE — из сети удален компонент;
- SERVICE_CONTROL_NETBINDENABLE — одна из связей стала доступна;
- SERVICE_CONTROL_NETBINDDISABLE — одна из связей стала недоступна.

Обработчик управляющих команд может получить команду SERVICE_ACCEPT_POWEREVENT только в том случае, если он был установлен функцией RegisterServiceCtrlHandlerEx.

Кроме того, по умолчанию предполагается, что каждый обработчик может получить команду:

- SERVICE_CONTROL_INTERROGATE — немедленно обновить статус сервиса.

Поле dwWin32ExitCode может содержать одно из следующих двух значений:

- ERROR_SERVICE_SPECIFIC_ERROR — ошибка во время запуска или остановки сервиса;
- NO_ERROR — ошибки нет.

Если в этом поле установлено значение ERROR_SERVICE_SPECIFIC_ERROR, то код самой ошибки находится в следующем поле dwServiceSpecificExitCode. Этот код может быть получен функцией GetLastError.

Поле dwServiceSpecificExitCode содержит код возврата из сервиса, этот код действителен только в том случае, если в поле dwWin32ExitCode установлено значение ERROR_SERVICE_SPECIFIC_ERROR.

Поле dwCheckPoint содержит значение, которое сервис должен периодически увеличивать на единицу, сообщая о продвижении своей работы во время инициализации и длительных переходов из состояния в состояние. Это значение может использоваться программой пользователя, которая отслеживает работу сервиса. Если это значение не используется пользовательской программой и переход из состояния в состояние занимает менее 30 секунд, то оно может быть установлено в 0.

Поле dwWaitHint содержит интервал времени в миллисекундах, в течение которого сервис переходит из состояния в состояние перед вызовом функции установки состояния SetServiceStatus. Если в течение этого интервала не произошло изменение состояния сервиса в поле dwServiceState или не изменилось значение поля dwCheckPoint, то менеджер сервисов считает, что в сервисе произошла ошибка.

В листинге 33.2 приведен пример функции ServiceMain сервиса.

Листинг 33.2. Пример функции ServiceMain сервиса

```
VOID WINAPI ServiceMain(DWORD dwArgc, LPTSTR *lpszArgv)
{
  // регистрируем обработчик управляющих команд для сервиса
  hServiceStatus = RegisterServiceCtrlHandler(
    service_name,        // имя сервиса
    ServiceCtrlHandler   // обработчик управляющих команд
    );
  if (!hServiceStatus)
  {
    out.open("C:\\ServiceFile.log");
    out << "Register service control handler failed.";
    out.close();

    return;
  }

  // инициализируем структуру состояния сервиса
  service_status.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
  service_status.dwCurrentState = SERVICE_START_PENDING;
  service_status.dwControlsAccepted = SERVICE_ACCEPT_STOP |
```

```
                                        SERVICE_ACCEPT_SHUTDOWN;
service_status.dwWin32ExitCode = ERROR_SERVICE_SPECIFIC_ERROR;
service_status.dwServiceSpecificExitCode = 0;
service_status.dwCheckPoint = 0;
service_status.dwWaitHint = 5000;

// устанавливаем состояние сервиса
if (!SetServiceStatus(hServiceStatus, &service_status))
{
  out.open("C:\\ServiceFile.log");
  out << "Set service status 'SERVICE_START_PENDING' failed.";
  out.close();

  return;
}

// определяем сервис как работающий
service_status.dwCurrentState = SERVICE_RUNNING;
// нет ошибок
service_status.dwWin32ExitCode = NO_ERROR;
// устанавливаем новое состояние сервиса
if (!SetServiceStatus(hServiceStatus, &service_status))
{
  out.open("C:\\ServiceFile.log");
  out << "Set service status 'START_PENDING' failed.";
  out.close();
  return;
}

// открываем файл протокола работы сервиса
out.open("C:\\ServiceFile.log");
out << "The service is started." << endl << flush;
out << "My name is: " << lpszArgv[0] << endl << flush;
for (int i = 1; i < (int)dwArgc; ++i)
out << "My " << i << " parameter =  " << lpszArgv[i] << endl << flush;

// рабочий цикл сервиса
while (service_status.dwCurrentState == SERVICE_RUNNING)
```

```
  {
    ++nCount;
    Sleep(3000);
  }
}
```

33.5. Организация обработчика управляющих команд

Функция-обработчик управляющих запросов, должна иметь следующий прототип:

```
VOID WINAPI имя_обработчика_запросов(DWORD  dwControl);
```

Если приложение содержит только один сервис, то эта функция обычно называется `ServiceCtrlHandler`. В противном случае обработчик запросов для каждого сервиса должен иметь свое имя. Эта функция содержит только один параметр, который содержит код управляющего сигнала. Возможны следующие управляющие коды:

- `SERVICE_CONTROL_STOP` — остановить сервис;
- `SERVICE_CONTROL_PAUSE` — приостановить сервис;
- `SERVICE_CONTROL_CONTINUE` — возобновить сервис;
- `SERVICE_CONTROL_INTERROGATE` — обновить состояние сервиса;
- `SERVICE_CONTROL_SHUTDOWN` — закончить работу сервиса.

Дополнительно в операционной системе Windows 2000 допускаются следующие коды управляющих команд:

- `SERVICE_CONTROL_PARAMCHANGE` — изменились параметры сервиса;
- `SERVICE_CONTROL_NETBINDADD` — соединиться с новым компонентом сети;
- `SERVICE_CONTROL_NETBINDREMOVE` — отсоединиться от компонента сети;
- `SERVICE_CONTROL_NETBINDENABLE` — связь с компонентом сети восстановлена;
- `SERVICE_CONTROL_NETBINDDISABLE` — связь с компонентом сети разорвана.

Но обработчик должен обрабатывать только те команды, которые допускаются в поле `dwControlsAccepted` структуры типа `SERVICE_STATUS`, рассмотренной в предыдущем разделе. Напомним, что по соглашению обработчик всегда получает сигнал с кодом `SERVICE_CONTROL_INTERROGATE`, по которому он должен немедленно обновить состояние сервиса. Для обновления состояния сервиса используется функция `SetServiceStatus`, которая была рассмотрена в предыдущем разделе. Кроме того, обработчик может получать

коды, определенные пользователем. Для кодов пользователя зарезервирован диапазон от 128 до 255.

Вызывается обработчик управляющих запросов диспетчером сервиса и, следовательно, выполняется в его контексте. Обработчик должен изменить состояние сервиса в течение 30 секунд, в противном случае диспетчер сервисов считает, что произошла ошибка. Если для изменения состояния сервиса требуется более продолжительный интервал времени, то для этой цели нужно запустить отдельный поток. Для обработки кода SERVICE_CONTROL_SHUTDOWN сервису отводится 20 секунд, в течение которых он должен освободить захваченные им ресурсы.

В листинге 33.3 приведена функция обработки управляющих команд для сервиса из предыдущего раздела.

Листинг 33.3. Пример функции обработки управляющих команд

```
VOID WINAPI ServiceCtrlHandler(DWORD dwControl)
{
  switch(dwControl)
  {
  case SERVICE_CONTROL_STOP:     // остановить сервис
    // записываем конечное значение счетчика
    out << "Count = " << nCount << endl;
    out << "The service is finished." << endl << flush;
    // закрываем файл
    out.close();

    // устанавливаем состояние остановки
    service_status.dwCurrentState = SERVICE_STOPPED;
    // изменить состояние сервиса
    SetServiceStatus(hServiceStatus, &service_status);
    break;

  case SERVICE_CONTROL_SHUTDOWN:     // завершить сервис
    service_status.dwCurrentState = SERVICE_STOPPED;
    // изменить состояние сервиса
    SetServiceStatus(hServiceStatus, &service_status);
    break;

  default:
```

```
    // увеличиваем значение контрольной точки
    ++service_status.dwCheckPoint;
    // оставляем состояние сервиса без изменения
    SetServiceStatus(hServiceStatus, &service_status);
    break;
  }

  return;
}
```

В заключение этого раздела отметим, что если обработчик регистрируется функцией `RegisterServiceCtrlHandlerEx`, то он должен иметь другой прототип, который предоставляет более широкие возможности по обработке управляющих команд. В этом случае функция обработки обычно называется `HandlerEx`, а ее прототип можно найти в справочной документации.

Глава 34

Работа с сервисами в Windows

34.1. Открытие доступа к базе данных сервисов

Для установки связи с менеджером сервисов и открытия доступа к базе данных сервисов используется функция `OpenSCManager`, которая имеет следующий прототип:

```
SC_HANDLE  OpenSCManager(
  LPCTSTR  lpMachineName,      // имя компьютера
  LPCTSTR  lpDatabaseName,     // имя базы данных сервисов
  DWORD    dwDesiredAccess     // тип доступа
);
```

В случае успешного завершения функция возвращает дескриптор базы данных сервисов, а в случае неудачи — `NULL`. Теперь опишем назначение параметров функции `OpenSCManager`.

Параметр `lpMachineName` должен содержать адрес строки с именем компьютера, на котором находится база данных сервисов. Имени компьютера должны предшествовать символы `\\`. Если этот параметр содержит `NULL` или указывает на пустую строку, то предполагается, что база данных расположена на локальном компьютере.

Параметр `lpDatabaseName` должен содержать адрес строки с именем базы данных сервисов. Если этот параметр равен `NULL`, то открывается активная база данных сервисов.

Параметр `dwDesiredAccess` должен содержать тип доступа к базе данных. Этот параметр может принимать любую комбинацию следующих флагов:

- `SC_MANAGER_ALL_ACCESS` — любой из нижеперечисленных доступов;
- `SC_MANAGER_CONNECT` — связь с менеджером сервисов;
- `SC_MANAGER_CREATE_SERVICE` — установка сервиса;

- SC_MANAGER_ENUMERATE_SERVICE — просмотр сервисов в базе данных;
- SC_MANAGER_LOCK — запирание базы данных сервисов;
- SC_MANAGER_QUERY_LOCK_STATUS — определение состояния базы данных.

При этом отметим, что флаг SC_MANAGER_CONNECT считается установленным по умолчанию, а флаг SC_MANAGER_ALL_ACCESS включает в себя флаг STANDARD_RIGHTS_REQUIRED.

Для использования в этом параметре также определены флаги, обозначающие следующие родовые права доступа к базе данных сервисов:

- GENERIC_READ — включает следующие флаги: STANDARD_RIGHTS_READ, SC_MANAGER_ENUMERATE_SERVICE и SC_MANAGER_QUERY_LOCK_STATUS;
- GENERIC_WRITE — включает флаги STANDARD_RIGHTS_WRITE и SC_MANAGER_CREATE_SERVICE;
- GENERIC_EXECUTE — включает следующие флаги: STANDARD_RIGHTS_EXECUTE, SC_MANAGER_CONNECT и SC_MANAGER_LOCK.

Возможна установка любой комбинации этих флагов.

После окончания работы с менеджером сервисов или сервисом нужно закрыть его дескриптор. Для этой цели предназначена функция CloseServiceHandle, которая имеет следующий прототип:

```
BOOL CloseServiceHandle(
  SC_HANDLE  hSCObject    // дескриптор менеджера сервисов или сервиса
);
```

В случае успешного завершения эта функция возвращает ненулевое значение, а в случае неудачи — FALSE. Единственным параметром этой функции является дескриптор менеджера сервисов или дескриптор сервиса.

Пример использования функций OpenSCManager и CloseServiceHandle будет приведен в следующем разделе, в котором будет рассмотрена программа установки сервиса.

В заключение этого раздела отметим, что процесс, связывающийся с менеджером сервисов и открывающий базу данных сервисов, должен иметь права и привилегии администратора.

34.2. Установка сервиса

Для установки сервисов в базу данных используется функция CreateService, которая имеет следующий прототип:

```
SC_HANDLE  CreateService(
  SC_HANDLE  hSCManager      // дескриптор базы данных сервисов
  LPCTSTR  lpServiceName     // внутреннее имя сервиса
```

```
    LPCTSTR  lpDisplayName     // внешнее имя сервиса
    DWORD    dwDesiredAccess   // разрешаемый доступ к сервису
    DWORD    dwServiceType     // тип сервиса
    DWORD    dwStartType       // способ запуска сервиса
    DWORD    dwErrorControl    // режим обработки ошибок
    LPCTSTR  lpBinaryPathName  // имя бинарного файла сервиса
    LPCTSTR  lpLoadOrderGroup  // имя группы загрузки сервисов
    LPDWORD  lpdwTagId         // тег сервиса в группе
    LPCTSTR  lpDependencies    // массив имен сервисов
    LPCTSTR  lpServiceStartName  // имя пользователя
    LPCTSTR  lpPassword        // пароль пользователя
);
```

В случае успешного завершения функция возвращает дескриптор установленного сервиса, а в случае неудачи — `NULL`. Рассмотрим назначение параметров этой функции.

Параметр `hSCManager` должен содержать дескриптор менеджера сервисов, который был предварительно открыт функцией `OpenSCManager`.

Параметр `lpServiceName` должен указывать на строку с именем сервиса, которое используется менеджером сервисов. Имена сервисов не чувствительны к верхнему и нижнему регистрам. Длина имени не может превышать 256 символов. Имя сервиса не может содержать символы / и \.

Параметр `lpDisplayName` должен указывать на имя сервиса, которое используется в панели управления Windows для ссылок на сервис. В этом имени различаются верхний и нижний регистры, а его длина не может превышать 256 символов.

В параметре `dwDesiredAccess` должны быть установлены флаги, которые определяют доступ к сервису. Возможна любая комбинация следующих флагов:

- `DELETE` — разрешено удаление сервиса;
- `SERVICE_CHANGE_CONFIG` — разрешено изменять конфигурацию сервиса;
- `SERVICE_ENUMERATE_DEPENDENTS` — разрешено просматривать зависимые сервисы;
- `SERVICE_INTERROGATE` — разрешено запрашивать состояние сервиса;
- `SERVICE_PAUSE_CONTINUE` — разрешено приостанавливать и возобновлять сервис;
- `SERVICE_QUERY_CONFIG` — разрешено запрашивать конфигурацию сервиса;
- `SERVICE_QUERY_STATUS` — разрешено запрашивать состояние сервиса;
- `SERVICE_START` — разрешено запускать сервис;

- `SERVICE_STOP` — разрешено останавливать сервис;
- `SERVICE_USER_DEFINED_CONTROL` — разрешено использовать управляющие команды пользователя.

Кроме того, для обеспечения возможности работы с атрибутами защиты сервиса нужно установить комбинированный флаг `STANDARD_RIGHTS_REQUIRED` — стандартные права доступа, который также включает флаг `DELETE`.

Также можно использовать флаг `SERVICE_ALL_ACCESS` — полный доступ, который включает флаг `STANDARD_RIGHTS_REQUIRED` и все перечисленные выше флаги. Дополнительно существуют следующие родовые права доступа, которые можно установить в этом параметре:

- `GENERIC_READ` — включает флаги `STANDARD_RIGHTS_READ`, `SERVICE_QUERY_CONFIG`, `SERVICE_QUERY_STATUS`, `SERVICE_INTERROGATE` и `SERVICE_ENUMERATE_DEPENDENTS`;
- `GENERIC_WRITE` — включает флаги `STANDARD_RIGHTS_WRITE` и `SERVICE_CHANGE_CONFIG`;
- `GENERIC_EXECUTE` — включает флаги `STANDARD_RIGHTS_EXECUTE`, `SERVICE_START`, `SERVICE_STOP`, `SERVICE_PAUSE_CONTINUE` и `SERVICE_USER_DEFINED_CONTROL`.

В параметре `dwServiceType` задается тип сервиса. Этот параметр может принимать одно из следующих значений:

- `SERVICE_WIN32_OWN_PROCESS` — отдельный процесс;
- `SERVICE_WIN32_SHARE_PROCESS` — разделяет процесс с другими сервисами;
- `SERVICE_KERNEL_DRIVER` — драйвер ядра;
- `SERVICE_FILE_SYSTEM_DRIVER` — драйвер файловой системы.

В случае если сервис не является драйвером, то может быть также установлен флаг `SERVICE_INTERACTIVE_PROCESS` — интерактивный процесс, который разрешает сервису взаимодействовать с рабочим столом.

В параметре `dwStartType` должен быть установлен режим запуска сервиса. Этот параметр может принимать одно из следующих значений:

- `SERVICE_BOOT_START` — запуск драйвера при загрузке системы;
- `SERVICE_SYSTEM_START` — запуск драйвера функцией `IoInitSystem`;
- `SERVICE_AUTO_START` — запуск сервиса менеджером сервисов при загрузке системы;
- `SERVICE_DEMAND_START` — запуск сервиса функцией `StartService`;
- `SERVICE_DISABLED` — сервис не может быть запущен.

Заметим, что первые два флага могут быть установлены только для драйверов.

В параметре `dwErrorControl` устанавливается режим обработки ошибок, произошедших во время запуска сервиса, если этот сервис запускается во

время загрузки системы. В этом параметре должно быть установлено одно из следующих значений:

- SERVICE_ERROR_IGNORE — игнорировать ошибку;
- SERVICE_ERROR_NORMAL — игнорировать ошибку и сообщить об этом в окне ошибок;
- SERVICE_ERROR_SEVERE — продолжить или перезагрузиться в безопасной конфигурации;
- SERVICE_ERROR_CRITICAL — прервать загрузку и перезагрузиться в безопасной конфигурации.

В последнем случае если ошибка произошла при загрузке системы в безопасной конфигурации, то загрузка прекращается.

Параметр lpBinaryPathName должен указывать на имя исполняемого exe-файла, который содержит сервис. Рекомендуется, чтобы это имя заканчивалось символом \.

Параметр lpLoadOrderGroup должен указывать на имя группы, содержащей порядок загрузки сервисов и в которую входит сервис. Если сервис не зависит от других сервисов, то в этом параметре должно быть установлено значение NULL.

В параметре lpdwTagId должен быть задан адрес переменной типа DWORD, по которому функция помещает тег сервиса в группе загрузки. Этот тег может в дальнейшем использоваться для определения порядка запуска сервисов. Если тег не используется, то этот параметр нужно установить в NULL. При этом отметим, что тег может использоваться только для драйверов, для которых задан режим загрузки SERVICE_BOOT_START или SERVICE_SYSTEM_START.

Параметр lpDependencies должен указывать на массив строк, каждая из которых содержит имя сервиса или имя группы сервисов, которые система должна запустить перед запуском устанавливаемого сервиса. Если сервис не зависит от других сервисов, то в этом параметре должно быть установлено значение NULL. Отметим, что имя группы должно начинаться с префикса SC_GROUP_IDENTIFIER, чтобы отличить это имя от имени сервиса.

Параметр lpServiceStartName должен указывать на строку с именем сервиса. Если сервис является процессом, то этот параметр должен указывать на имя пользователя, под которым будет запущен сервис. Если сервис запускается под текущим локальным именем, то этот параметр должен быть установлен в NULL. Сервис, взаимодействующий с рабочим столом, должен быть запущен только под текущим именем пользователя.

Параметр lpPassword должен указывать на строку с паролем пользователя, под именем которого запускается сервис. Если сервис запускается под именем текущего пользователя, то в этом параметре должно быть установлено значение NULL.

В листинге 34.1 приведена программа, в которой устанавливается сервис, рассмотренный в *гл. 33*. При этом заметим, что исполняемый файл DemoService.exe содержит коды функций, представленных в листингах 33.1, 33.2 и 33.3.

Листинг 34.1. Установка сервиса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  SC_HANDLE  hServiceControlManager, hService;

  // связываемся с менеджером сервисов
  hServiceControlManager = OpenSCManager(
    NULL,      // локальная машина
    NULL,      // активная база данных сервисов
    SC_MANAGER_CREATE_SERVICE  // возможно создание сервиса
    );
  if (hServiceControlManager == NULL)
  {
    cout << "Open service control manager failed." << endl
      << "The last error code: " << GetLastError() << endl
      << "Press any key to exit." << endl;
    cin.get();

    return 0;
  }
  cout << "Service control manager is opened." << endl
    << "Press any key to continue." << endl;
  cin.get();

  // устанавливаем новый сервис
  hService = CreateService(
    hServiceControlManager,  // дескриптор менеджера сервисов
    "DemoService",           // внутреннее имя сервиса, используемое SCM
    "Demo_Service",          // внешнее имя сервиса в панели управления
    SERVICE_ALL_ACCESS,      // полный контроль над сервисом
```

```
    SERVICE_WIN32_OWN_PROCESS,   // сервис является процессом
    SERVICE_DEMAND_START,    // запускается сервис по требованию
    SERVICE_ERROR_NORMAL,    // обработка ошибок нормальная
    "C:\\DemoService.exe",   // путь к сервису
    NULL,                    // сервис не принадлежит к группе
    NULL,        // тег группы не изменяется
    NULL,        // сервис не зависит от других сервисов
    NULL,        // имя совпадает с текущим именем учетной записи
    NULL         // пароля нет
  );
  if (hService == NULL)
  {
    cout << "Create service failed." << endl
      << "The last error code: " << GetLastError() << endl
      << "Press any key to exit." << endl;
    cin.get();

    // закрываем дескриптор менеджера сервисов
    CloseServiceHandle(hServiceControlManager);

    return 0;
  }

  cout << "Service is installed." << endl
    << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескрипторы
  CloseServiceHandle(hService);
  CloseServiceHandle(hServiceControlManager);

  return 0;
}
```

Напомним, что после окончания работы с сервисом нужно закрыть его дескриптор, используя для этого функцию `CloseServiceHandle`, которая была рассмотрена в предыдущем разделе. Отметим, что эта функция не уничтожает сервис, а только разрывает связь прикладной программы с этим сервисом. Для удаления сервиса из базы данных нужно использовать функцию `DeleteService`.

34.3. Открытие доступа к сервису

Для открытия доступа к уже установленному сервису используется функция OpenService, которая имеет следующий прототип:

```
SC_HANDLE  OpenService(
  SC_HANDLE  hSCManager,       // дескриптор базы данных сервисов
  LPCTSTR    lpServiceName,    // имя сервиса
  DWORD      dwDesiredAccess   // требуемый доступ к сервису
);
```

В случае успешного завершения функция возвращает дескриптор сервиса, а в случае неудачи — NULL.

Параметр hSCManager должен содержать дескриптор базы данных сервисов, связь с которой была предварительно установлена вызовом функции OpenSCManager.

Параметр lpServiceName должен указывать на строку с именем сервиса.

В параметре dwDesiredAccess устанавливаются флаги доступа к сервису. Флаги доступа, которые можно установить в этом параметре, совпадают с флагами, которые можно установить в параметре dwDesiredAccess функции CreateService, рассмотренной в предыдущем разделе.

В следующем разделе приведен пример программы, в которой открывается доступ к уже установленному сервису для запуска этого сервиса из приложения.

34.4. Запуск сервиса

Для запуска сервиса используется функция StartService, которая имеет следующий прототип:

```
BOOL  StartService(
  SC_HANDLE  hService,             // дескриптор сервиса
  DWORD      dwNumServiceArgs,     // количество аргументов
  LPCTSTR    *lpServiceArgVectors  // массив аргументов
```

В случае удачного завершения функция возвращает ненулевое значение, а в случае неудачи — FALSE.

В параметре hService должен быть установлен дескриптор сервиса, который был предварительно получен функциями CreateService или OpenService.

Параметр dwNumServiceArgs должен содержать количество аргументов, которые передаются сервису. Каждый из аргументов является символьной строкой. Если аргументов нет, то в параметре должно быть установлено значение ноль.

Параметр lpServiceArgVectors должен содержать адрес массива, каждый элемент которого является указателем на строку. Эти строки и являются аргументами, которые передаются функции сервиса ServiceMain. Если аргументов нет, то этот параметр должен быть установлен в NULL.

Отметим, что во время работы функции StartService менеджер сервисов устанавливает следующие состояния сервиса в структуре SERVICE_STATUS:

- в поле dwCurrentState устанавливает значение SERVICE_START_PENDING;
- в поле dwControlsAccepted устанавливает значение ноль, что запрещает обработку управляющих команд;
- в поле dwCheckPoint устанавливает значение ноль;
- в поле dwWaitHint устанавливает значение, равное двум секундам.

Кроме того, во время работы функции StartService менеджер запросов блокирует базу данных сервисов.

В листинге 34.2 приведен пример программы, которая открывает доступ к сервису, а затем запускает его. Этот сервис должен быть предварительно установлен программой, которая приведена в листинге 34.1.

Листинг 34.2. Открытие и запуск сервиса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char  service_name[] = "DemoService";    // имя сервиса
  char  *param[] = { "p1", "p2", "p3" };   // список параметров
  SC_HANDLE  hServiceControlManager, hService;

  // связываемся с менеджером сервисов
  hServiceControlManager = OpenSCManager(
    NULL,    // локальная машина
    NULL,    // активная база данных сервисов
    SC_MANAGER_CONNECT   // соединение с менеджером сервисов
    );
  if (hServiceControlManager == NULL)
  {
    cout << "Open service control manager failed." << endl
      << "The last error code: " << GetLastError() << endl
```

```
    << "Press any key to continue." << endl;
  cin.get();

  return 0;
}

cout << "Service control manager is opened." << endl
  << "Press any key to continue." << endl;
cin.get();

// открываем сервис
hService = OpenService(
  hServiceControlManager,  // дескриптор менеджера сервисов
  service_name,            // имя сервиса
  SERVICE_ALL_ACCESS       // полный доступ к сервису
);
if (hService == NULL)
{
  cout << "Open service failed." << endl
    << "The last error code: " << GetLastError() << endl
    << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескриптор менеджера сервисов
  CloseServiceHandle(hServiceControlManager);

  return 0;
}

cout << "Service is opened." << endl
  << "Press any key to continue." << endl;
cin.get();

// стартуем сервис
if (!StartService(
  hService,    // дескриптор сервиса
  3,           // три параметра
  (const char**)param  // указатель на массив параметров
  ))
```

```
  {
    cout << "Start service failed." << endl
      << "The last error code: " << GetLastError() << endl
      << "Press any key to exit." << endl;
    cin.get();

    // закрываем дескрипторы
    CloseServiceHandle(hServiceControlManager);
    CloseServiceHandle(hService);

    return 0;
  }

  cout << "The service is started." << endl
    << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескрипторы
  CloseServiceHandle(hServiceControlManager);
  CloseServiceHandle(hService);

  return 0;
}
```

34.5. Определение и изменение состояния сервиса

Для определения состояния сервиса используется функция `QueryServiceStatus`, которая имеет следующий прототип:

```
BOOL  QueryServiceStatus(
  SC_HANDLE  hService,              // дескриптор сервиса
  LPSERVICE_STATUS  lpServiceStatus  // адрес структуры SERVICE_STATUS
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — `FALSE`.

В параметре `hService` должен быть установлен дескриптор сервиса, который предварительно получен функцией `CreateService` или `OpenService`.

Параметр `lpServiceStatus` должен указывать на структуру типа `SERVICE_STATUS`, которая содержит информацию о состоянии сервиса. Эта структура имеет следующий формат:

```
typedef  struct  _SERVICE_STATUS {
  DWORD  dwServiceType;          // тип сервиса
  DWORD  dwCurrentState;         // текущее состояние сервиса
  DWORD  dwControlsAccepted;     // допускаемое управление
  DWORD  dwWin32ExitCode;        // код возврата для Win32
  DWORD  dwServiceSpecificExitCode; // специфический код возврата
                                    // сервиса
  DWORD  dwCheckPoint;           // контрольная точка
  DWORD  dwWaitHint;             // период ожидания команды управления
}  SERVICE_STATUS,  *LPSERVICE_STATUS;
```

Подробнее назначение полей этой структуры было описано в *разд. 33.4*, посвященном организации функции `ServiceMain`.

В листинге 34.3 приведена программа, в которой определяется состояние сервиса посредством вызова функции `QueryServiceStatus`.

Листинг 34.3. Определение состояния сервиса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char  service_name[] = "DemoService";    // имя сервиса
  SC_HANDLE  hServiceControlManager, hService;
  SERVICE_STATUS  service_status;

  // связываемся с менеджером сервисов
  hServiceControlManager = OpenSCManager(
    NULL,       // локальная машина
    NULL,       // активная база данных сервисов
    SC_MANAGER_CONNECT   // соединение с менеджером сервисов
    );
  if (hServiceControlManager == NULL)
  {
    cout << "Open service control manager failed." << endl
      << "The last error code: " << GetLastError() << endl
```

```
    << "Press any key to continue." << endl;
  cin.get();

  return 0;
}

cout << "Service control manager is opened." << endl
  << "Press any key to continue." << endl;
cin.get();

// открываем сервис
hService = OpenService(
  hServiceControlManager,  // дескриптор менеджера сервисов
  service_name,            // имя сервиса
  SERVICE_ALL_ACCESS       // любой доступ к сервису
);
if (hService == NULL)
{
  cout << "Open service failed." << endl
    << "The last error code: " << GetLastError() << endl
    << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескриптор менеджера сервисов
  CloseServiceHandle(hServiceControlManager);

  return 0;
}

cout << "Service is opened." << endl
  << "Press any key to continue." << endl;
cin.get();

// определение состояния сервиса
if (!QueryServiceStatus(
  hService,        // дескриптор сервиса
  &service_status  // адрес структуры состояния сервиса
  ))
{
```

```
    cout << "Query service sataus failed." << endl
      << "The last error code: " << GetLastError() << endl
      << "Press any key to exit." << endl;
    cin.get();

    // закрываем дескрипторы
    CloseServiceHandle(hServiceControlManager);
    CloseServiceHandle(hService);

    return 0;
  }

  switch (service_status.dwCurrentState)
  {
  case SERVICE_STOPPED:
    cout << "The service is stopped." << endl;
    break;
  case SERVICE_RUNNING:
    cout << "The service is running." << endl;
    break;
  default:
    cout << "The service status: " << service_status.dwCurrentState
        << endl;
    break;
  }

  cout << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескрипторы
  CloseServiceHandle(hServiceControlManager);
  CloseServiceHandle(hService);

  return 0;
}
```

Для изменения состояния сервиса используется функция `SetServiceStatus`, которая была подробно рассмотрена в *разд. 33.4*, посвященном функции `ServiceMain`.

34.6. Определение и изменение конфигурации сервиса

Под конфигурацией сервиса понимается информация, которая описывает внутреннее и внешнее имена сервиса, режимы его запуска и работы.

Для определения конфигурации сервиса используется функция `QueryServiceConfig`, которая имеет следующий прототип:

```
BOOL  QueryServiceConfig(
  SC_HANDLE  hService,        // дескриптор сервиса
  LPQUERY_SERVICE_CONFIG  lpServiceConfig,   // указатель на буфер
  DWORD     dwBufferSize,     // размер буфера
  LPDWORD   lpdwBytesNeeded  // необходимая длина буфера
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. Параметры функции имеют следующее назначение.

Параметр `hService` должен содержать дескриптор сервиса, который был предварительно открыт при помощи функции `CreateService` или `OpenService`.

Параметр `lpServiceConfig` должен указывать на область памяти, в которую функция поместит информацию о конфигурации сервиса в случае своего успешного завершения. Эта информация будет храниться в формате, который определяется следующей структурой:

```
typedef  struct  _QUERY_SERVICE_CONFIG {
  DWORD  dwServiceType;       // тип сервиса
  DWORD  dwStartType;         // режим запуска сервиса
  DWORD  dwErrorControl;      // режим обработки ошибок при запуске
  LPSTR  lpBinaryPathName;    // имя бинарного файла сервиса
  LPSTR  lpLoadOrderGroup;    // информация о порядке загрузки сервисов
  DWORD  dwTagId;             // тег сервиса в группе загрузки
  LPSTR  lpDependencies;      // информация о порядке запуска сервисов
  LPSTR  lpServiceStartName; // имя пользователя, который запускает
                              // сервис
  LPSTR  lpDisplayName;       // имя сервиса в окне управления сервисами
} QUERY_SERVICE_CONFIG, *LPQUERY_SERVICE_CONFIG;
```

В поля этой структуры записываются те значения, которые были установлены в соответствующих параметрах при вызовах функций `CreateService` и `OpenService`.

В листинге 34.4 приведена программа, в которой при помощи функции `QueryServiceConfig` определяется конфигурация сервиса.

Листинг 34.4. Определение конфигурации сервиса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char  service_name[] = "DemoService";    // имя сервиса
  SC_HANDLE  hServiceControlManager, hService;

  // указатель на буфер для информации о конфигурации сервиса
  QUERY_SERVICE_CONFIG*  lpQueryServiceConfig;
  // переменная для размера структуры в случае неудачи функции
  DWORD  dwBytesNeeded;

  // связываемся с менеджером сервисов
  hServiceControlManager = OpenSCManager(
    NULL,      // локальная машина
    NULL,      // активная база данных сервисов
    SC_MANAGER_CONNECT   // соединение с менеджером сервисов
    );
  if (hServiceControlManager == NULL)
  {
    cout << "Open service control manager failed." << endl
      << "The last error code: " << GetLastError() << endl
      << "Press any key to continue." << endl;
    cin.get();

    return 0;
  }

  cout << "Service control manager is opened." << endl
    << "Press any key to continue." << endl;
  cin.get();

  // открываем сервис
```

```
hService = OpenService(
  hServiceControlManager,  // дескриптор менеджера сервисов
  service_name,            // имя сервиса
  SERVICE_ALL_ACCESS       // любой доступ к сервису
);
if (hService == NULL)
{
  cout << "Open service failed." << endl
    << "The last error code: " << GetLastError() << endl
    << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескриптор менеджера сервисов
  CloseServiceHandle(hServiceControlManager);

  return 0;
}

cout << "Service is opened." << endl
  << "Press any key to continue." << endl;
cin.get();

// захватываем память под буфер
lpQueryServiceConfig = (LPQUERY_SERVICE_CONFIG)new char[4096];

// определяем конфигурацию сервиса
if (!QueryServiceConfig(
  hService,              // дескриптор сервиса
  lpQueryServiceConfig,  // адрес структуры конфигурации сервиса
  4096,                  // размер этой структуры
  &dwBytesNeeded         // необходимое количество байтов
  ))
{
  cout << "Query service configuartion failed." << endl
    << "The last error code: " << GetLastError() << endl
    << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескрипторы
```

```
    CloseServiceHandle(hServiceControlManager);
    CloseServiceHandle(hService);

    return 0;
  }
  // распечатываем информацию о конфигурации
  cout << "Service type: " << lpQueryServiceConfig->dwServiceType << endl
    << "Start type: " << lpQueryServiceConfig->dwStartType << endl
    << "Error control: " << lpQueryServiceConfig->dwErrorControl << endl
    << "Binary path name: " << lpQueryServiceConfig->lpBinaryPathName
    << endl
    << "Service start name: " << lpQueryServiceConfig->lpServiceStartName
    << endl
    << "Display name: " << lpQueryServiceConfig->lpDisplayName << endl
    << endl;
  // освобождаем память
  delete[] lpQueryServiceConfig;

  cout << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескрипторы
  CloseServiceHandle(hServiceControlManager);
  CloseServiceHandle(hService);

  return 0;
}
```

Получить информацию о реакции диспетчера сервиса на завершение работы сервиса без установки состояния `SERVICE_STOPPED` можно при помощи вызова функции `QueryServiceConfig2`.

Для изменения конфигурации сервиса можно использовать функцию `ChangeServiceConfig`, которая имеет следующий прототип:

```
  BOOL  ChangeServiceConfig(
    SC_HANDLE  hService,       // дескриптор сервиса
    DWORD    dwServiceType,    // тип сервиса
    DWORD    dwStartType,      // режим запуска сервиса
    DWORD    dwErrorControl,   // режим обработки ошибок при запуске
    LPCSTR   lpBinaryPathName, // имя бинарного файла сервиса
```

```
    LPCSTR   lpLoadOrderGroup, // информация о порядке загрузки сервисов
    LPDWORD  lpdwTagId,        // тег сервиса в группе загрузки
    LPCSTR   lpDependencies,   // информация о порядке запуска сервисов
    LPCSTR   lpServiceStartName,   // имя пользователя
    LPCSTR   lpPassword,       // пароль пользователя
    LPCSTR   lpDisplayName     // имя сервиса в окне управления сервисами
  );
```

В случае удачного завершения функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. Параметры этой функции имеют тот же смысл, что и соответствующие параметры в функциях `CreateService` и `OpenService`.

В листинге 34.5 приведен пример, в котором функция `ChangeServiceConfig` используется для изменения режима запуска сервиса.

Листинг 34.5. Изменение конфигурации сервиса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char  service_name[] = "DemoService";    // имя сервиса
  SC_HANDLE  hServiceControlManager, hService;

  // указатель на буфер для информации о конфигурации сервиса
  QUERY_SERVICE_CONFIG*  lpQueryServiceConfig;
  // переменная для размера структуры в случае неудачи функции
  DWORD  dwBytesNeeded;

  // связываемся с менеджером сервисов
  hServiceControlManager = OpenSCManager(
    NULL,      // локальная машина
    NULL,      // активная база данных сервисов
    SC_MANAGER_CONNECT   // соединение с менеджером сервисов
    );
  if (hServiceControlManager == NULL)
  {
    cout << "Open service control manager failed." << endl
      << "The last error code: " << GetLastError() << endl
      << "Press any key to continue." << endl;
```

```
  cin.get();

  return 0;
}

cout << "Service control manager is opened." << endl
  << "Press any key to continue." << endl;
cin.get();

// открываем сервис
hService = OpenService(
  hServiceControlManager,  // дескриптор менеджера сервисов
  service_name,            // имя сервиса
  SERVICE_ALL_ACCESS       // любой доступ к сервису
);
if (hService == NULL)
{
  cout << "Open service failed." << endl
    << "The last error code: " << GetLastError() << endl
    << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескриптор менеджера сервисов
  CloseServiceHandle(hServiceControlManager);

  return 0;
}

cout << "Service is opened." << endl
  << "Press any key to continue." << endl;
cin.get();

// захватываем память под буфер
lpQueryServiceConfig = (LPQUERY_SERVICE_CONFIG)new char[4096];

// определям конфигурацию сервиса
QueryServiceConfig( hService, lpQueryServiceConfig, 4096,
                    &dwBytesNeeded);
// распечатываем информацию о режиме запуска сервиса
```

```
cout << "Old start type: " << lpQueryServiceConfig->dwStartType
     << endl;

// изменяем режим запуска сервиса
if (!ChangeServiceConfig(
  hService,              // дескриптор сервиса
  SERVICE_NO_CHANGE,     // тип сервиса не изменяем
  SERVICE_AUTO_START,    // запуск во время загрузки системы
  SERVICE_NO_CHANGE,     // режим обработки ошибок не изменяем
  NULL, NULL, NULL, NULL, NULL, NULL, NULL  // все остальные параметры
                                            // конфигурации не изменяем
  ))
{
  cout << "Change service configuration failed." << endl
    << "The last error code: " << GetLastError() << endl
    << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескрипторы
  CloseServiceHandle(hServiceControlManager);
  CloseServiceHandle(hService);

  return 0;
}

// определям конфигурацию сервиса
QueryServiceConfig( hService, lpQueryServiceConfig, 4096,
                    &dwBytesNeeded);
// распечатываем информацию о режиме запуска сервиса
cout << "New start type: " << lpQueryServiceConfig->dwStartType
     << endl;

// восстанавливаем режим запуска сервиса
ChangeServiceConfig(
  hService,              // дескриптор сервиса
  SERVICE_NO_CHANGE,     // тип сервиса не изменяем
  SERVICE_DEMAND_START,  // запуск по требованию
  SERVICE_NO_CHANGE,     // режим обработки ошибок не изменяем
  NULL, NULL, NULL, NULL, NULL, NULL, NULL  // все остальные параметры
```

```
                                                    // конфигурации не изменяем
    );
  // определям конфигурацию сервиса
  QueryServiceConfig( hService, lpQueryServiceConfig, 4096,
                      &dwBytesNeeded);
  // распечатываем информацию о режиме запуска сервиса
  cout << "Old start type: " << lpQueryServiceConfig->dwStartType
       << endl;

  // освобождаем память
  delete[] lpQueryServiceConfig;

  cout << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескрипторы
  CloseServiceHandle(hServiceControlManager);
  CloseServiceHandle(hService);

  return 0;
}
```

Изменить информацию о реакции диспетчера сервиса на завершение работы сервиса без установки состояния `SERVICE_STOPPED` можно при помощи вызова функции `ChangeServiceConfig2`.

34.7. Определение имени сервиса

Назовем внутренним именем сервиса то имя, под которым сервис хранится в базе данных сервисов, а также используется менеджером сервисов для ссылок на этот сервис. Узнать внутреннее имя сервиса можно при помощи функции `GetServiceKeyName`, которая имеет следующий прототип:

```
BOOL  GetServiceKeyName(
  SC_HANDLE  hSCManager,     // дескриптор базы данных сервисов
  LPCSTR     lpDisplayName,  // указатель на внешнее имя сервиса
  LPSTR      lpServiceName,  // указатель на внутреннее имя сервиса
  LPDWORD    lpdwBuffer      // размер памяти для внутреннего имени
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — FALSE. Параметры функции имеют следующее назначение.

В параметре hSCManager должен быть установлен дескриптор базы данных сервисов, которая была предварительно открыта функцией OpenSCManager.

Параметр lpDisplayName должен указывать на строку с именем сервиса, которое используется в окне управления сервисами, вызываемом из **Start | Settings | Control Panel | Administrative Tools | Services** (Пуск | Настройка | Панель управление | Администрирование | Службы).

Параметр lpServiceName должен указывать на область памяти, куда функция запишет внутреннее имя сервиса, т. е. имя, которое используется для ссылок менеджером сервисов. В случае удачного завершения функция записывает в эту память внутреннее имя сервиса.

Параметр lpdwBuffer должен указывать на двойное слово, в котором установлена длина области памяти, на которую указывает параметр lpServiceName. Функция записывает по этому адресу фактическую длину внутреннего имени сервиса, исключая завершающий пустой символ.

В листинге 34.6 приведена программа, в которой показано, как определить внутреннее имя сервиса по его внешнему имени при помощи функции GetServiceKeyName.

Листинг 34.6. Определение внутреннего имени сервиса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char  service_name[] = "DemoService";  // имя сервиса
  SC_HANDLE  hServiceControlManager;   // дескриптор базы данных сервисов

  // указатель на буфер для внутреннего имени сервиса
  char*  lpszServiceName;
  // переменная для длины буфера в случае неудачи функции
  DWORD  dwBufferSize;

  // связываемся с менеджером сервисов
  hServiceControlManager = OpenSCManager(
    NULL,       // локальная машина
    NULL,       // активная база данных сервисов
```

```
    SC_MANAGER_CONNECT   // соединение с менеджером сервисов
    );
  if (hServiceControlManager == NULL)
  {
    cout << "Open service control manager failed." << endl
      << "The last error code: " << GetLastError() << endl
      << "Press any key to continue." << endl;
    cin.get();

    return 0;
  }

  cout << "Service control manager is opened." << endl
    << "Press any key to continue." << endl;
  cin.get();

  // захватываем память под буфер
  dwBufferSize = 256;
  lpszServiceName = (char*)new char[dwBufferSize];

  // определение внутреннего имени сервиса
  if (!GetServiceKeyName(
    hServiceControlManager,    // дескриптор базы данных сервисов
    "Demo_Service",    // внешнее имя сервиса
    lpszServiceName,   // буфер для имени сервиса
    &dwBufferSize      // количество необходимых байтов
    ))
  {
    cout << "Get service key name failed." << endl
      << "The last error code: " << GetLastError() << endl
      << "Press any key to exit." << endl;
    cin.get();

    // закрываем дескриптор базы данных сервисов
    CloseServiceHandle(hServiceControlManager);

    return 0;
  }

  // выводим на консоль внутреннее имя сервиса
```

```
  cout << "Service key name: " << lpszServiceName << endl;

  cout << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескриптор базы данных сервисов
  CloseServiceHandle(hServiceControlManager);

  return 0;
}
```

Назовем внешним именем сервиса то имя, которое используется для ссылок на сервис в окне управления сервисами. Узнать внешнее имя сервиса можно с помощью функции `GetServiceDisplayName`, которая имеет следующий прототип:

```
BOOL  GetServiceDisplayName(
  SC_HANDLE  hSCManager,       // дескриптор базы данных сервисов
  LPCSTR     lpServiceName,    // указатель на внутреннее имя сервиса
  LPSTR      lpDisplayName,    // указатель на внешнее имя сервиса
  LPDWORD    lpdwBuffer        // размер памяти для внешнего имени
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. Параметры функции имеют следующее назначение.

В параметре `hSCManager` должен быть установлен дескриптор базы данных сервисов, которая была предварительно открыта функцией `OpenSCManager`.

Параметр `lpServiceName` должен указывать на строку с внутренним именем сервиса, под которым сервис хранится в базе данных.

Параметр `lpDisplayName` должен указывать на область памяти, куда функция запишет внешнее имя сервиса, т. е. имя, которое используется для ссылок на сервис в окне управления сервисами.

Параметр `lpdwBuffer` должен указывать на переменную типа `DWORD`, в котором установлена длина области памяти, на которую указывает параметр `lpDisplayName`. Функция записывает по этому адресу фактическую длину внешнего имени сервиса, исключая завершающий пустой символ.

В листинге 34.7 приведена программа, в которой показано, как определить внутренне имя сервиса по его внешнему имени при помощи функции `GetServiceDisplayName`.

Листинг 34.7. Определение внешнего имени сервиса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char  service_name[] = "DemoService";  // имя сервиса
  SC_HANDLE  hServiceControlManager;   // дескриптор базы данных сервисов

  // указатель на буфер для внутренненего имени сервиса
  char*  lpszServiceName;
  // переменная для длины буфера в случае неудачи функции
  DWORD  dwBufferSize;

  // связываемся с менеджером сервисов
  hServiceControlManager = OpenSCManager(
    NULL,      // локальная машина
    NULL,      // активная база данных сервисов
    SC_MANAGER_CONNECT   // соединение с менеджером сервисов
    );
  if (hServiceControlManager == NULL)
  {
    cout << "Open service control manager failed." << endl
      << "The last error code: " << GetLastError() << endl
      << "Press any key to continue." << endl;
    cin.get();

    return 0;
  }

  cout << "Service control manager is opened." << endl
    << "Press any key to continue." << endl;
  cin.get();

  // захватываем память под буфер
  dwBufferSize = 256;
  lpszServiceName = (char*)new char[dwBufferSize];
```

```
  // определение внешнего имени сервиса
  if (!GetServiceDisplayName(
    hServiceControlManager,  // дескриптор базы данных сервисов
    "DemoService",           // внутреннее имя сервиса
    lpszServiceName,         // буфер для имени сервиса
    &dwBufferSize            // количество необходимых байтов
    ))
  {
    cout << "Get service display name failed." << endl
      << "The last error code: " << GetLastError() << endl
      << "Press any key to exit." << endl;
    cin.get();

    // закрываем дескриптор базы данных сервисов
    CloseServiceHandle(hServiceControlManager);

    return 0;
  }

  // выводим на консоль внутреннее имя сервиса
  cout << "Service display name: " << lpszServiceName << endl;

  cout << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескриптор базы данных сервисов
  CloseServiceHandle(hServiceControlManager);

  return 0;
}
```

34.8. Управление сервисом

Приложение может послать сервису управляющую команду, которая будет передана обработчику управляющих команд сервиса. Для этой цели используется функция `ControlService`, которая имеет следующий прототип:

```
  BOOL  ControlService(
    SC_HANDLE  hService,     // дескриптор сервиса
```

```
    DWORD       dwControl,     // управляющий код
    LPSERVICE_STATUS  lpServiceStatus  // адрес структуры SERVICE_STATUS
  );
```

В случае успешного завершения функция `ControlService` возвращает ненулевое значение, а в случае неудачи — `FALSE`. Параметры функции имеют следующее назначение.

Параметр `hService` должен содержать дескриптор сервиса, который был предварительно получен вызовом функции `CreateService` или `OpenService`.

Параметр `dwControl` должен содержать код управляющей команды, который может принимать одно из следующих значений:

- `SERVICE_CONTROL_STOP` — остановить сервис;
- `SERVICE_CONTROL_PAUSE` — приостановить сервис;
- `SERVICE_CONTROL_CONTINUE` — возобновить сервис;
- `SERVICE_CONTROL_INTERROGATE` — обновить состояние сервиса.

Дополнительно, в операционной системе Windows 2000 возможны следующие коды управляющих команд:

- `SERVICE_CONTROL_PARAMCHANGE` — параметры запуска изменились;
- `SERVICE_CONTROL_NETBINDCHANGE` — параметры сети изменились.

Кроме того, сервису можно послать управляющие команды, коды которых заданы пользователем. Для задания таких команд зарезервирован диапазон от 128 до 255. Сервис может обрабатывать такие команды только в том случае, если при создании сервиса в параметре `dwDesiredAccess` установлен флаг `SERVICE_USER_DEFINED_CONTROL`.

В листинге 34.8 приведена программа, в которой функция `ControlService` используется для остановки сервиса.

Листинг 34.8. Управление сервисом

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char  service_name[] = "DemoService";    // имя сервиса
  SC_HANDLE  hServiceControlManager, hService;
  SERVICE_STATUS  service_status;

  // связываемся с менеджером сервисов
```

```
hServiceControlManager = OpenSCManager(
  NULL,      // локальная машина
  NULL,      // активная база данных сервисов
  SC_MANAGER_CONNECT   // соединение с менеджером сервисов
  );
if (hServiceControlManager == NULL)
{
  cout << "Open service control manager failed." << endl
    << "The last error code: " << GetLastError() << endl
    << "Press any key to continue." << endl;
  cin.get();

  return 0;
}

cout << "Service control manager is opened." << endl
  << "Press any key to continue." << endl;
cin.get();

// открываем сервис
hService = OpenService(
  hServiceControlManager,  // дескриптор менеджера сервисов
  service_name,            // имя сервиса
  SERVICE_ALL_ACCESS       // любой доступ к сервису
);
if (hService == NULL)
{
  cout << "Open service failed." << endl
    << "The last error code: " << GetLastError() << endl
    << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескриптор менеджера сервисов
  CloseServiceHandle(hServiceControlManager);

  return 0;
}

cout << "Service is opened." << endl
```

```
    << "Press any key to continue." << endl;
  cin.get();

  // останавливаем сервис
  if (!ControlService(
    hService,              // дескриптор сервиса
    SERVICE_CONTROL_STOP,  // управляющая команда
    &service_status        // адрес структуры состояния сервиса
    ))
  {
    cout << "Control service failed." << endl
      << "The last error code: " << GetLastError() << endl
      << "Press any key to exit." << endl;
    cin.get();

    // закрываем дескрипторы
    CloseServiceHandle(hServiceControlManager);
    CloseServiceHandle(hService);

    return 0;
  }
  cout << "The service is stopped." << endl
    << "Press any key to exit." << endl;
  cin.get();
  // закрываем дескрипторы
  CloseServiceHandle(hServiceControlManager);
  CloseServiceHandle(hService);

  return 0;
}
```

34.9. Удаление сервисов

Для удаления сервиса из базы данных сервисов служит функция `DeleteService`, которая имеет следующий прототип:

```
BOOL  DeleteService(
  SC_HANDLE  hService    // дескриптор сервиса
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — FALSE. Единственный параметр hService этой функции должен содержать дескриптор сервиса, который был предварительно получен вызовом функции CreateService или OpenService. Если сервис работает, то он не удаляется из базы данных до тех пор, пока не будет остановлен.

В листинге 34.9 приведена программа, в которой показано, как удалить сервис, используя функцию DeleteService.

Листинг 34.9. Удаление сервиса

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char  service_name[] = "DemoService";  // имя сервиса
  SERVICE_STATUS  service_status;        // состояние сервиса
  SC_HANDLE  hServiceControlManager, hService;

  // связываемся с менеджером сервисов
  hServiceControlManager = OpenSCManager(
    NULL,      // локальная машина
    NULL,      // активная база данных сервисов
    SC_MANAGER_CONNECT   // соединение с менеджером сервисов
    );
  if (hServiceControlManager == NULL)
  {
    cout << "Open service control manager failed." << endl
      << "The last error code: " << GetLastError() << endl
      << "Press any key to continue." << endl;
    cin.get();

    return 0;
  }

  cout << "Service control manager is opened." << endl
    << "Press any key to continue." << endl;
```

```
cin.get();

// открываем сервис
hService = OpenService(
  hServiceControlManager,  // дескриптор менеджера сервисов
  service_name,            // имя сервиса
  SERVICE_ALL_ACCESS | DELETE  // любой доступ к сервису
                               // и удаление из базы данных
);
if (hService == NULL)
{
  cout << "Open service failed." << endl
    << "The last error code: " << GetLastError() << endl
    << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескриптор менеджера сервисов
  CloseServiceHandle(hServiceControlManager);

  return 0;
}

cout << "Service is opened." << endl
  << "Press any key to continue." << endl;
cin.get();

// получаем состояние сервиса
if (!QueryServiceStatus(hService, &service_status))
{
  cout << "Query service status failed." << endl
    << "The last error code: " << GetLastError() << endl
    << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескрипторы
  CloseServiceHandle(hServiceControlManager);
  CloseServiceHandle(hService);

  return 0;
```

```
}

// если сервис работает, то останавливаем его
if (service_status.dwCurrentState != SERVICE_STOPPED)
{
  cout << "Service is working. It will be stoped" << endl;
  if (!ControlService(hService, SERVICE_CONTROL_STOP, &service_status))
  {
    cout << "Control service failed." << endl
    << "The last error code: " << GetLastError() << endl
    << "Press any key to exit." << endl;
    cin.get();

    // закрываем дескрипторы
    CloseServiceHandle(hServiceControlManager);
    CloseServiceHandle(hService);

    return 0;
  }

  // ждем, пока сервис остановится
  Sleep(500);
}

// удаляем сервис
if (!DeleteService(hService))
{
  cout << "Delete service failed." << endl
    << "The last error code: " << GetLastError() << endl
    << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескрипторы
  CloseServiceHandle(hServiceControlManager);
  CloseServiceHandle(hService);

  return 0;
}

cout << "The service is deleted." << endl
```

```
    << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескрипторы
  CloseServiceHandle(hServiceControlManager);
  CloseServiceHandle(hService);

  return 0;
}
```

34.10. Блокирование базы данных сервисов

Для получения монопольного доступа к базе данных сервисов процесс может заблокировать доступ других процессов к этой базе данных, используя функцию `LockServiceDatabase`, которая имеет следующий прототип:

```
SC_LOCK  LockServiceDatabase(
  SC_HANDLE  hSCManager    // дескриптор базы данных сервисов
);
```

В случае успешного завершения функция возвращает дескриптор блокирования базы данных сервисов, а в случае неудачи — `NULL`. Единственный параметр `hSCManager` этой функции должен содержать дескриптор базы данных сервисов, доступ к которой блокируется функцией. Эта база данных сервисов должна быть предварительно открыта для доступа вызовом функции `OpenSCManager`. Причем база данных сервисов должна быть открыта с ключом `SC_MANAGER_LOCK`.

Для разблокирования доступа к базе данных сервисов используется функция `UnockServiceDatabase`, которая имеет следующий прототип:

```
BOOL  UnlockServiceDatabase(
  SC_LOCK  hSCLock     // дескриптор блокирования базы данных сервисов
);
```

В случае успешного завершения функция возвратит ненулевое значение, а в случае неудачи — `FALSE`. Единственный параметр этой функции должен содержать дескриптор блокирования базы данных сервисов, который был предварительно получен вызовом функции `LockServiceDatabase`.

Для того чтобы узнать заблокирована база данных сервисов или нет, используется функция `QueryServiceLockStatus`, которая имеет следующий прототип:

```
BOOL  QueryServiceLockStatus(
  SC_HANDLE  hSCManager,       // дескриптор базы данных сервисов
```

```
  LPQUERY_SERVICE_LOCK_STATUS  lpLockStatus, // информация о блокировке
  DWORD      dwBufSize,        // размер буфера для информации
  LPDWORD    lpdwBytesNeeded   // количество необходимых байтов
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — FALSE. Параметры этой функции имеют следующее назначение.

Параметр hSCManager должен содержать дескриптор базы данных сервисов, который был предварительно получен вызовом функции OpenSCManager. Причем база данных сервисов должна быть открыта с ключом SC_MANAGER_QUERY_LOCK_STATUS.

Параметр lpLockStatus должен указывать на область памяти, куда система поместит информацию о блокировании базы данных сервисов. Эта информация будет записана в структуру типа QUERY_SERVICE_LOCK_STATUS, которая имеет следующий формат:

```
typedef  struct  _QUERY_SERVICE_LOCK_STATUSA {
  DWORD  fIsLocked;       // состояние блокировки
  LPSTR  lpLockOwner;     // владелец блокировки
  DWORD  dwLockDuration;  // время блокировки
} QUERY_SERVICE_LOCK_STATUS, *LPQUERY_SERVICE_LOCK_STATUS;
```

Поля этой структуры имеют следующее назначение.

В поле fIsLocked хранится состояние блокировки базы данных сервисов. Если значение этого поля равно нулю, то база данных не заблокирована. В противном случае база данных заблокирована.

Если база данных сервисов заблокирована, то в поле lpLockOwner записывается адрес строки, которая содержит имя пользователя, заблокировавшего базу данных.

Если база данных сервисов заблокирована, то в поле dwLockDuration хранится целое число без знака, которое определяет продолжительность блокирования базы данных в секундах.

В параметре dwBufSize должен быть установлен размер памяти, на которую указывает параметр lpLockStatus.

Параметр lpdwBytesNeeded должен указывать на двойное слово, в которое функция в случае своего неудачного завершения поместит необходимый размер буфера для информации о блокировании базы данных сервисов.

В листинге 34.10 приведена программа, в которой показано, как блокировать и разблокировать базу данных сервисов, а также определить заблокирована база данных сервисов или нет.

Листинг 34.10. Блокирование и разблокирование базы данных сервисов

```
#include <windows.h>
#include <iostream.h>

int main()
{
  char  service_name[] = "DemoService";  // имя сервиса
  SC_HANDLE  hServiceControlManager;     // дескриптор базы данных
                                         // сервисов
  SC_LOCK    hScLock;  // дескриптор блокировки базы данных сервисов

  // указатель на буфер для состояния о блокировке
  LPQUERY_SERVICE_LOCK_STATUS    lpLockStatus;
  // длина буфера
  DWORD  dwBufferSize;
  // переменная для длины буфера в случае неудачи функции
  DWORD  dwBytesNeeded;

  // связываемся с менеджером сервисов
  hServiceControlManager = OpenSCManager(
    NULL,      // локальная машина
    NULL,      // активная база данных сервисов
    SC_MANAGER_LOCK |              // разрешена блокировка базы данных
    SC_MANAGER_QUERY_LOCK_STATUS   // и определения состояния блокировки
    );
  if (hServiceControlManager == NULL)
  {
    cout << "Open service control manager failed." << endl
      << "The last error code: " << GetLastError() << endl
      << "Press any key to continue." << endl;
    cin.get();

    return 0;
  }

  cout << "Service control manager is opened." << endl
    << "Press any key to continue." << endl;
```

```
cin.get();

// блокируем базу данных сервисов
hScLock = LockServiceDatabase(hServiceControlManager);
if (hScLock == NULL)
{
  cout << "Lock service database failed." << endl
    << "The last error code: " << GetLastError() << endl
    << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескриптор базы данных сервисов
  CloseServiceHandle(hServiceControlManager);

  return 0;
}

cout << "\tThe service database is locked." << endl;

// захватываем память под буфер
dwBufferSize = 4096;
lpLockStatus = (LPQUERY_SERVICE_LOCK_STATUS)new char[dwBufferSize];
// определяем состояние блокировки
QueryServiceLockStatus(hServiceControlManager, lpLockStatus,
                       dwBufferSize, &dwBytesNeeded);
// выводим на консоль состояние блокировки
cout << "Lock state: " << lpLockStatus->fIsLocked << endl
  << "Lock owner: " << lpLockStatus->lpLockOwner << endl
  << "Lock duration: " << lpLockStatus->dwLockDuration << endl;

// разблокируем базу данных сервисов
UnlockServiceDatabase(hScLock);

cout << "\tThe service database is unlocked." << endl;

// определяем состояние блокировки
QueryServiceLockStatus(hServiceControlManager, lpLockStatus,
                       dwBufferSize, &dwBytesNeeded);
```

```
  // выводим на консоль состояние блокировки
  cout << "Lock state: " << lpLockStatus->fIsLocked << endl
    << "Lock owner: " << lpLockStatus->lpLockOwner << endl
    << "Lock duration: " << lpLockStatus->dwLockDuration << endl;

  cout << "Press any key to exit." << endl;
  cin.get();

  // закрываем дескрипторы
  CloseServiceHandle(hServiceControlManager);
  CloseServiceHandle(hScLock);

  return 0;
}
```

Часть XI

Управление безопасностью в Windows

Глава 35

Система информационной безопасности

35.1. Контроль доступа к ресурсам

Системы информационной безопасности обеспечивают защиту информационных ресурсов от несанкционированного доступа со стороны пользователей. Здесь под пользователями понимаются процессы, которые выполняются от имени пользователей компьютерной системы. Безопасность ресурсов обеспечивается посредством контроля доступа к этим ресурсам. *Контроль доступа к ресурсам* это некоторый механизм, который разрешает доступ к ресурсу только авторизованным пользователям этого ресурса, т. е. только тем пользователям, которым разрешен доступ к этому ресурсу. Часто механизм авторизации пользователя включает также и его аутентификацию, т. е. установление подлинности пользователя или, другими словами, определение того, не выдает ли пользователь системы себя за кого-то другого.

Более формально в системах информационной безопасности все ресурсы разбиваются на две категории: объекты и субъекты. *Объекты* представляют собой пассивные ресурсы, которые требуется защитить. К таким ресурсам относятся, например, файлы, каналы передачи данных, оперативная память, принтеры и другие внешние устройства для хранения и отображения информации. Для каждого объекта определяется множество операций, которые можно выполнить над этим объектом. С другой стороны, *субъекты* представляют собой активные ресурсы, которые выполняют операции над объектами и представляются процессами, исполняемыми от имени пользователей информационной системы. Выполнение субъектами операций над объектами называется *доступом субъектов к объектам*. Чтобы контролировать доступ субъектов к объектам для каждой пары (субъект, объект) определяется множество операций, которые данный субъект может выполнять над данным объектом. Очевидно, что это множество является подмножеством множества всех операций, заданных над объектом.

В системе информационной безопасности существует субъект, который контролирует доступ других субъектов к объектам, руководствуясь при этом некоторыми правилами. Совокупность этих правил называется *политикой безопасности*, а сам субъект, реализующий эту политику безопасности, называется *менеджером* или *монитором безопасности*. Предполагается, что менеджер безопасности всегда находится в активном состоянии, а для каждого пользователя создается, по крайней мере, один субъект, при его входе в систему. Кроме того, субъекты могут создаваться из объектов другими субъектами, работающими от имени пользователей, путем выполнения операции создания субъекта. Например, исполняемый файл, хранящийся на диске, может запускаться на выполнение, превращаясь при этом из объекта в субъект.

Следует отличать менеджера безопасности, который является программой, от администратора системы информационной безопасности, который является человеком. Вообще, *администратор системы информационной безопасности* занимается регистрацией пользователей информационной системы, наделением их определенными правами и привилегиями, а также отслеживанием работы системы безопасности. Здесь уместно сделать различие между правами и привилегиями. *Правом* называется возможность субъекта выполнять некоторые операции над объектами. Например, пользователь имеет право или, другими словами, ему разрешается читать некоторый файл. *Привилегия* это более общее понятие, которое позволяет пользователю выполнять действия в отношении других объектов и субъектов системы информационной безопасности. Как правило, привилегия выделяет пользователя из числа обычных пользователей системы. Например, пользователь имеет привилегию блокировать доступ к системе других пользователей.

35.2. Политика безопасности

Политикой безопасности называется набор требований, выполнение которых обеспечивает безопасную работу информационной системы. Следует отличать политику безопасности от способа реализации этой политики, который обычно называется *механизмом реализации политики безопасности* и представляет собой комплекс программных и аппаратных средств. Вообще понятие политики безопасности довольно общее и используется не только при разработке систем информационной безопасности, но и при разработке систем безопасности любых организаций.

Обычно разработка политики безопасности для системы разбивается на два этапа. На первом этапе проводится анализ угроз для системы информационной безопасности. При этом подробно изучаются следующие два вопроса:

- определяются информационные ресурсы, которые должны быть защищены;
- определяются возможные угрозы этим информационным ресурсам.

Причем угрозы рассматриваются с двух точек зрения:

- внутренние угрозы, т. е. угрозы от программного обеспечения;
- внешние угрозы, т. е. угрозы от пользователей.

Первый этап разработки политики безопасности также называют *анализом рисков*.

После определения защищаемых ресурсов и возможных угроз этим ресурсам переходят ко второму этапу, на котором разрабатываются средства защиты от возможных угроз безопасности информационной системы. При этом средства защиты должны предусматривать работу в трех режимах:

- обработка рутинных задач информационной безопасности системы;
- обработка исключительных ситуаций, как, например, обнаружение атаки вируса;
- обработка аварийных ситуаций, таких как, например, обнаружение вируса в системе.

Второй этап разработки политики безопасности также называют *управлением рисками*.

35.3. Модель безопасности

После разработки политики безопасности строят *модель безопасности*, которая является формальным описанием разработанной политики безопасности, хотя часто понятия политики и модели безопасности не различают. При этом модель безопасности рассматривается как система, которая включает следующие компоненты:

- пассивные ресурсы, которые называются объектами;
- наборы операций, которые можно выполнять над каждым объектом;
- активные ресурсы, которые выполняют операции над объектами;
- атрибуты защиты объектов, которые описывают права доступа субъектов к объектам.

Контроль доступа субъектов к объектам выполняет менеджер безопасности, используя для этого атрибуты безопасности объектов и соблюдая при этом некоторые правила, которые называются *правилами управления доступом*. Формально правила управления доступом могут быть выражены некоторыми предикатами (условиями) над компонентами системы, которые должны выполняться при доступе субъекта к объекту.

Состояние объектов, наборов операций, субъектов и атрибутов защиты объектов называется *состоянием системы безопасности*. Состояние системы безопасности называется *безопасным*, если в этом состоянии нет несанкционированного доступа субъекта к объекту. При доступе субъекта к объекту

система безопасности переходит из одного состояния в другое. Задача системы безопасности состоит в том, чтобы обеспечить переход из безопасного состояния в безопасное же состояние. Однако обеспечение такого перехода не обязательно гарантирует то, что все возможные дальнейшие состояния системы также будут безопасными. Поэтому общая задача безопасности системы может быть сформулирована следующим образом:

- начальное состояние системы безопасное;
- правила управления доступом обеспечивают переход из безопасного состояния системы в безопасное состояние системы;
- любое состояние системы, достижимое из ее начального состояния, является безопасным состоянием.

Для решения этой задачи используются методы математической логики. В общем случае доказательство безопасности системы является трудно решаемой задачей.

35.4. Дискреционная политика безопасности

Дискреционная политика безопасности основывается на следующих принципах:

- для каждого объекта определяется набор операций, которые можно выполнять над этим объектом;
- субъект может выполнить операцию над объектом при условии, если он имеет право на выполнение этой операции;
- субъект, который имеет права на выполнение некоторых операций над объектом, может передать эти права другому субъекту.

Суть дискреционной политики безопасности заключается в том, что права доступа субъекта к объекту определяются другим субъектом, который имеет эти права. Другими словами, доступ субъектов к объектам разрешается другими субъектами или можно сказать, что оставлен на усмотрение другим субъектам, которые обладают этими правами.

Менеджер безопасности контролирует доступ субъекта к объекту следующим образом: он проверяет, есть ли у субъекта разрешение на выполнение затребованной операции над объектом. Если такое разрешение есть, то выполнение затребованной операции разрешается, в противном случае выполнение затребованной операции запрещается.

В приведенном выше варианте дискреционной политики безопасности существует очевидная проблема. А именно, субъект может передать права на доступ к объекту другому субъекту, который не имеет таких прав, без контроля менеджера безопасности системы. Поэтому такая дискреционная политика безопасности называется *либеральной*. Чтобы разрешить указанную проблему, последний принцип в дискреционной политике безопасности

обычно изменяют следующим образом: субъект может передать права на выполнение некоторых операций над объектом другому субъекту только при условии, если он наделен такими полномочиями.

Полученную политику безопасности называют *строгой* дискреционной политикой безопасности. Очевидно, что такая политика повышает безопасность системы в целом.

35.5. Дискреционная модель безопасности

Для построения дискреционной модели безопасности поступают следующим образом. Идентифицируются все объекты и все субъекты системы. Для каждого объекта определяются операции, которые субъекты могут выполнять над этим объектом. После этого строится матрица, каждая строка которой соответствует одному субъекту и каждый столбец которой соответствует одному объекту системы. Тогда каждая клетка матрицы однозначно идентифицируется субъектом и объектом системы. В клетки полученной матрицы записывают права, которые субъект, соответствующий данной строке матрицы, имеет по отношению к объекту, соответствующему данному столбцу матрицы. Полученная матрица называется *матрицей управления доступами*. Например, матрица управления доступами может быть такой, как это показано в табл. 35.1, где обозначения для прав субъектов по отношению к объектам приведены в табл. 35.2, 35.3 соответственно.

Таблица 35.1. *Пример матрицы управления доступами*

	Kim_file	**Don_file**	**Payroll_1**	**Payroll_2**	**Martin_file**
Kim	R, W	R	R, W	R	
Joe		R			
Don		R, W	R		
Jones			R	R	
Martin					R, W
Manager	CP	CP	C	C	
Maria			R, W	R, W	

Права доступа субъекта к объекту, как правило, делятся на две категории:

- операции, которые субъект может выполнять над объектом;
- права управления объектом.

Операции, которые разрешается выполнять над объектами, также называются *режимами доступа к объекту*. Права управления объектом также называются *режимами управления объектом*.

Рассмотрим подробнее возможные операции над объектами. В системах информационной безопасности под объектами обычно понимаются файлы (хотя это могут быть и другие объекты с очень похожими операциями доступа). Поэтому остановимся на объектах подобных файлам, над которыми разрешается выполнять операции, или, другими словами, для которых определены режимы доступа, представленные в табл. 35.2.

Как правило, файловые системы операционных систем имеют древовидную структуру, в которой листья дерева соответствуют файлам, а некорневые вершины — каталогам (директориям, папкам). В этом случае возможны три варианта управления доступом к каталогам и файлам:

- запрашивается доступ к файлу, но не к каталогу;
- запрашивается доступ к каталогу, но не к файлу;
- запрашивается доступ к каталогу и файлу.

В первом случае субъект указывает полное имя файла, к которому он хочет получить доступ. В этом случае доступ к другим файлам из этого каталога данному субъекту запрещен.

Во втором случае субъект указывает только имя каталога, к которому он хочет получить доступ. Он получает доступ ко всем файлам, которые находятся в этом каталоге.

В третьем случае субъект для доступа к файлу должен указать как имя файла, к которому он хочет получить доступ, так и имя каталога, в котором находится этот файл.

Таблица 35.2. *Режимы доступа к объектам*

Название режима доступа	Краткое обозначение	Описание режима доступа
READ	R	Разрешается чтения содержимого объекта, как правило, в этом режиме также разрешено и копирование файла
WRITE	W	Разрешается любая модификация объекта
WRITE_APPEND	WA	Разрешается только добавление данных в объект
WRITE_CHANGE	WC	Разрешается только изменять или удалять данные из объекта, не разрешается добавлять данные в объект

Таблица 35.2 *(окончание)*

Название режима доступа	Краткое обозначение	Описание режима доступа
WRITE_UPDATE	WU	Разрешается только изменять данные из объекта, не разрешается удалять или добавлять данные в объект
DELETE	D	Разрешается удаление объекта
EXECUTE	E	Разрешается исполнение объекта
NULL	N	Нет доступа к объекту

Теперь перейдем к режимам управления доступом к объектам. Режимы управления объектом определяют субъектов, которые могут изменять права доступа других субъектов к объекту или передавать права управления этим объектом другим субъектам. Как правило, существует два режима управления, которые представлены в табл. 35.3.

Таблица 35.3. *Режимы управления объектами*

Название режима доступа	Краткое обозначение	Описание режима доступа
CONTROL	C	Разрешается устанавливать режимы доступа к объекту для субъектов, не разрешается передавать режим управления другому субъекту
CONTROL WITH PASSING ABILITY	CP	Разрешается устанавливать режимы доступа к объекту для субъектов и передавать режимы управления объектом другим субъектом

Так как субъекты могут передавать права управления объектами другим субъектам, то должны быть определены правила, которым подчиняются субъекты при передаче таких прав. Набор таких правил определяет *модель управления* в дискреционной модели безопасности. Существуют четыре модели управления, которые могут использоваться в дискреционной модели безопасности:

- иерархическое управление (hierarchical control);
- управление правами доступа владельцем объекта (concept of ownership).
- либеральное управление (laissez-fair);
- централизованное управление (centralized control).

Кратко опишем каждую из этих моделей управления.

В иерархической модели управления все субъекты, которые могут управлять правами доступа к объектам, упорядочиваются (иерархически), организуя при этом древовидную структуру. В вершине этой структуры находится администратор системы безопасности, который может управлять правами доступа ко всем объектам и наделять такими правами любого субъекта. Для остальных субъектов предполагается, что они могут управлять правами доступа только субъектов, которые находятся ниже их в иерархии.

В модели управления, использующей понятие владельца объекта, управление правами доступа к объекту для всех субъектов выполняется именно владельцем этого объекта. Владельцем объекта, как правило, считается тот субъект, который создает этот объект. В такой модели управления также присутствует администратор системы безопасности, который может ограничить права владельца объекта на передачу прав доступа к объекту другим субъектам. Фактически в этом случае модель управления представляет собой двухуровневую, иерархическую модель управления.

В либеральной модели управления передача прав доступа к объекту от субъекта, который обладает этими правами, к другому субъекту никак не контролируется. Как уже говорилось, такая модель управления является самой ненадежной. Здесь же можно сказать, что laissez-fair является сокращением французского выражения "Laissez fair, laissez passer", которое может быть переведено, как "Пусть все идет своим чередом".

В централизованной модели управления правами управления доступом к объектам обладает только один субъект, как правило, это администратор системы. Таким образом, все вопросы по разрешению доступа субъектов к объектам системы разрешаются только администратором. Централизованная модель управления является наиболее надежной из рассмотренных моделей управления доступами к объектам.

В заключение этого раздела отметим, что для каждой из рассмотренных моделей управления существует формальное описание, которое позволяет анализировать надежность дискреционной модели безопасности.

35.6. Реализация дискреционной модели безопасности

В дискреционной модели безопасности при обработке запроса субъекта на доступ к объекту менеджер безопасности должен проверить содержимое соответствующей клетки матрицы управления доступами, чтобы выяснить, имеет ли этот субъект право доступа к объекту, которое он запрашивает, или нет. Так как на практике матрица управления доступами довольно разреженная и имеет большой размер, то она редко хранится непосредственно в памяти компьютера. Как правило, для хранения матрицы управления доступами используют один из следующих двух подходов.

При первом подходе матрицу управления доступами рассматривают по строкам. Так как каждая строка этой матрицы соответствует одному субъекту, то можно сказать, что вся строка матрицы доступов описывает все объекты, к которым этот субъект имеет доступ, и режимы доступа к этим объектам. Эта информация называется *возможностями* (capabilities) *субъекта*. Для каждого пользователя информационной системы администратором информационной системы задаются его возможности. При этом менеджер системы безопасности работает так, что каждому субъекту разрешается доступ к ресурсам только в соответствии с его возможностями. Поэтому для процесса, работающего от имени некоторого пользователя, доступ к ресурсам также ограничен только его возможностями. Такой подход повышает безопасность системы безопасности, т. к. в случае, если от имени пользователя работает несанкционированная программа (например, вирус), то эта программа будет иметь доступ только к ограниченному множеству объектов.

Для реализации такого подхода, как правило, поступают следующим образом. Возможности каждого пользователя хранят в виде списка, каждый элемент которого содержит имена объектов и режимы доступа к этим объектам. Такой список называется *профилем* (profile) *пользователя*. Тогда субъект может открыть заданный режим доступа к объекту только в том случае, если этот объект и заданный режим присутствует в его профиле. При работе с профилями возникают следующие проблемы:

- при удалении объекта нужно корректировать профили всех пользователей, которые имели доступ к этому объекту;
- при изменении режимов доступа к объекту также нужно корректировать профили всех пользователей, которые имели доступ к этому объекту.

При втором подходе матрицу управления доступами рассматривают по столбцам. Так как каждый столбец матрицы соответствует одному объекту, то можно сказать, что весь столбец матрицы доступов описывает всех субъектов, которые имеют доступ к этому объекту, и режимы доступа каждого субъекта. В этом случае при реализации столбец матрицы управления доступами хранится в виде списка, который называется *список управления доступами* (access control list). На практике такие списки обычно сокращенно называют ACL, а элементы этих списков — ACE (access control elements). Каждый элемент списка управления доступами содержит имя пользователя и разрешенные этому пользователю режимы доступа к объекту. Список управления доступами объекта обычно создается владельцем объекта или администратором информационной системы. При этом менеджер системы безопасности открывает субъекту требуемый доступ к объекту только в том случае, если его имя и требуемый режим доступа находятся в списке управления доступами для этого объекта.

Очевидно, что при такой реализации матрицы управления доступами легко решаются вопросы корректировки доступа к объектам и удаления объектов,

однако возникают сложности с корректировкой информации о пользователях системы. Так, например, если пользователю запрещается доступ к ресурсам информационной системы, то его имя и режимы доступа нужно удалить из всех списков управления доступами для всех объектов, к которым этот пользователь имел доступ. Для упрощения решения этой задачи всех пользователей информационной системы разбивают на группы, предполагая, что каждая из этих групп может работать только с определенными объектами в заданных режимах доступа. Тогда в списки управления доступами вносят имена групп пользователей с соответствующими режимами доступа. Это значительно сокращает объем списков и упрощает их корректировку. При запросе субъекта на доступ к объекту менеджер системы безопасности проверяет принадлежность этого субъекта к какой-либо группе, а затем сверяется со списком управления доступами, чтобы разрешить или запретить доступ этого субъекта к объекту.

Глава 36

Управление безопасностью в Windows

36.1. Модель безопасности в Windows

В операционных системах Windows NT/2000/XP реализована дискреционная модель безопасности. В качестве активных субъектов этой модели безопасности рассматриваются процессы и потоки, каждый из которых работает от имени некоторого пользователя. Когда пользователь регистрируется и входит в систему, то для него создается *маркер доступа* (access token), который идентифицирует этого пользователя и содержит его привилегии. Каждый процесс, исполняемый от имени пользователя, имеет маркер доступа этого пользователя. Маркер доступа используется для контроля доступа процесса к объектам, которые называются в Windows *охраняемыми объектами* (securable objects). К охраняемым объектам относятся все объекты Windows, которые могут иметь имя. Кроме того, к охраняемым объектам относятся также потоки и процессы. Каждый охраняемый объект имеет *дескриптор безопасности* (security descriptor), который создается вместе с охраняемым объектом и содержит информацию, необходимую для защиты объекта от несанкционированного доступа. В дескрипторе безопасности идентифицируется владелец объекта, определяются пользователи и группы пользователей, которым разрешен или запрещен доступ к охраняемому объекту, а также информация для аудита доступа к объекту. Изменять информацию, заданную в дескрипторе безопасности, может только владелец объекта, которым по умолчанию является создатель этого объекта. При доступе к охраняемому объекту система сверяет информацию о пользователе, заданную в маркере доступа, с информацией, заданной в дескрипторе безопасности. Если в дескрипторе безопасности указано, что пользователю разрешен доступ к объекту, то процесс получает запрашиваемый доступ, в противном случае в доступе отказывается.

Для хранения информации о пользователях, которым разрешен или запрещен доступ к охраняемым объектам, каждый дескриптор безопасности

содержит *список управления дискреционным доступом* (Discretionary Access-Control List, DACL). Для управления аудитом доступа к объекту в дескрипторе безопасности хранится *список управления системным доступом* (System Access-Control List, SACL). Общее название для этих списков — *списки управления доступом* (Access-Control Lists) или сокращенно ACL.

Таким образом, можно сказать, что в операционных системах Windows NT/2000/XP реализована дискреционная модель безопасности, в которой управление правами доступа к объекту выполняет владелец этого объекта. В следующих разделах более подробно рассмотрена структура компонент системы управления безопасностью в операционных системах Windows и работа с функциями, которые выполняют настройку этих компонент.

В заключение этого раздела скажем, что реализованная в Windows модель безопасности удовлетворяет уровню безопасности C-2, который определен министерством обороны США. Перечислим наиболее важные требования, которым должна удовлетворять система, поддерживающая безопасность на этом уровне:

- ❒ владелец ресурса должен управлять доступом к ресурсу;
- ❒ операционная система должна защищать ресурсы от случайного повторного использования, например, содержимое памяти завершившегося процесса должно быть недоступно другому процессу;
- ❒ каждый пользователь системы должен при входе в систему указывать свое имя и пароль, система должна иметь средства для отслеживания работы пользователя;
- ❒ администратор системы должен иметь средства для проведения аудита событий, связанных с безопасностью системы, доступ к полученным при аудите данным должен иметь только авторизованный администратор системы;
- ❒ система должна защищать себя от постороннего вмешательства во время своей работы, например, система должна защищать себя от модификации извне во время своей работы.

36.2. Учетные записи

Прежде чем пользователь сможет работать в среде операционных систем Windows NT/2000/XP, он должен быть зарегистрирован администратором системы. При регистрации пользователя администратором создается *учетная запись пользователя* (user account), которая хранится в базе данных менеджера учетных записей (Security Account Manager, SAM). Эта база данных является частью реестра Windows и доступ к ней имеет только администратор системы. Регистрация пользователя выполняется на уровне домена локальной сети, что упрощает управление доступом к ресурсам этой сети. Вообще

доменом называется группа компьютеров в локальной сети, которые разделяют общую базу данных учетных записей пользователей. Более подробно о доменах будет сказано в следующем разделе.

В учетной записи хранится следующая информация:

- ❒ имя пользователя для входа в систему (username);
- ❒ пароль (password);
- ❒ полное имя пользователя (full name);
- ❒ допустимое время работы в системе (logon hours);
- ❒ допустимые компьютеры для входа в систему (logon workstations);
- ❒ дата окончания срока действия учетной записи (expiration date);
- ❒ рабочий каталог пользователя (home directory);
- ❒ действия, выполняемые при загрузке (logon script);
- ❒ профиль пользователя, который содержит информацию для настройки рабочего стола пользователя (profile);
- ❒ тип учетной записи (account type).

Существуют четыре типа учетных записей, а именно:

- ❒ учетная запись пользователя;
- ❒ учетная запись группы пользователей;
- ❒ учетная запись компьютера;
- ❒ учетная запись домена.

Учетная запись компьютера используется для регистрации компьютеров в домене и создается при подключении компьютера к домену. Также как пользователь должен войти в систему на компьютере, так и компьютер должен при включении войти в домен. *Учетная запись домена* используется для регистрации домена в других доменах. Учетные записи пользователя, компьютера и домена имеют пароли. Доступ к паролю пользователя имеют администратор и сам пользователь. Паролем домена управляет администратор, а паролем компьютера управляет контроллер доменов (Primary Domain Controller, PDC).

После того как для пользователя заведена учетная запись, он может работать на любом компьютере, входящем в домен. При входе пользователя в операционную систему он должен ввести свое имя (user name) и пароль (password). После этого система безопасности операционной системы Windows проверяет базу данных учетных записей на наличие учетной записи данного пользователя. Если пользователь зарегистрирован в системе, то ему разрешается доступ в систему, в противном случае доступ в систему запрещается.

По умолчанию операционная система Windows NT создает три учетных записи:

- Administrator — администратор;
- Guest — гость;
- System — система.

Учетная запись администратора предназначена для управления локальной системой, установленной на одном компьютере. По умолчанию эта учетная запись создается без пароля. Желательно, чтобы в дальнейшем администратор системы изменил имя пользователя и установил пароль для этой учетной записи. Учетную запись администратора системы нельзя заблокировать. Администратор системы может создавать новые учетные записи.

Учетная запись гостя предназначена для обычного пользователя системы, который не имеет административных полномочий. Эта учетная запись также создается без пароля. Она не может быть удалена, но может быть переименована.

Учетная запись системы используется самой операционной системой для выполнения различных задач, которые требуют аутентификации. Эта учетная запись также имеет административные привилегии.

В операционной системе Windows XP дополнительно создаются учетные записи:

- HelpAssistant — используется удаленным экспертом для регистрации на локальном компьютере, чтобы оказать помощь пользователю этого компьютера;
- SUPPORT_xxxxxxxx — предназначен для сервисной поддержки и обслуживания аппаратного и программного обеспечения его производителями в режиме удаленного доступа.

В заключение этого раздела еще раз отметим, что создание, удаление и модификацию учетных записей могут производить только пользователи, которые входят в группу администраторов системы или группу операторов учетных записей.

36.3. Домены

Домен это группа компьютеров в локальной сети, которые поддерживают одну политику безопасности и разделяют общую базу данных учетных записей пользователей. Эта политика безопасности и база данных учетных записей пользователей хранятся в базе данных домена (Domain Directory Database). Домен включает серверы и рабочие станции. Один из серверов выбирается контроллером домена. На этом сервере хранится база данных учетных записей домена. Основная функция контроллера домена — выполнять

аутентификацию пользователей при входе в домен на одном из компьютеров домена. Администратор управляет доменом через контроллер домена. В домене можно определить один первичный контроллер домена (Primary Domain Controller, PDC) и несколько вторичных контроллеров (Backup Domain Controller, BDC). На вторичных контроллерах домена хранится копия базы данных учетных записей. Поэтому вторичные контроллеры домена также могут выполнять аутентификацию пользователей.

Так как пользователь регистрируется в домене, то будет уместно пояснить организацию доменов на платформе Windows. Понятие домена локальной сети было введено для того, чтобы упростить управление безопасностью объектов в локальной сети. То есть зарегистрированный в домене пользователь может работать на любом компьютере, входящем в домен и отмеченном в учетной записи пользователя, а также имеет доступ к ресурсам, расположенным на компьютерах домена. Конечно, доступ к этим ресурсам может быть выполнен только в том случае, если у пользователя есть соответствующие полномочия. В локальной сети может быть несколько доменов. Для того чтобы пользователь, зарегистрированный в одном домене, имел доступ к ресурсам другого домена, между этими доменами должно быть установлено *отношение доверия* (trust relationship). Отношение доверия характеризуется двумя свойствами: направлением и транзитивностью.

С точки зрения свойства направление отношения доверия может быть односторонним и двусторонним. В одностороннем отношении доверия между двумя доменами один из них разрешает пользователям другого домена доступ к своим ресурсам, но второй домен не разрешает этого первому. В двустороннем отношении доверия оба домена разрешают друг другу доступ к своим ресурсам.

Свойство *транзитивности* определяет возможность передачи прав на доступ к ресурсам одного домена транзитивно через другой домен. Если отношение доверия транзитивно, то права на доступ к ресурсам домена, который передает эти права, могут быть переданы другому домену через домен, который является получателем этих прав. В противном случае такая передача права на доступ к ресурсам другому домену запрещается.

Можно сказать, что отношение доверия связывает домены в древовидные структуры. А множество всех доменов локальной сети образует *лес*.

Операционная система Windows NT поддерживает односторонние и двусторонние нетранзитивные отношения доверия. Операционная система Windows 2000 поддерживает следующие типы отношений доверия:

- транзитивное двухстороннее отношение доверия;
- нетранзитивное одностороннее отношение доверия.

Двустороннее, нетранзитивное отношение доверия должно моделироваться, используя два односторонних нетранзитивных отношения доверия.

36.4. Группы

Теперь рассмотрим группы пользователей. Это понятие также было введено для того, чтобы упростить управление безопасностью объектов. *Группа* (group) это просто набор учетных записей пользователей, которые объединены по какому-либо признаку, например, пользователи одной группы могут работать в одном отделе. При этом отметим, что одна учетная запись пользователя может входить более чем в одну группу. Каждая группа имеет свою учетную запись и наделена своими правами и полномочиями. Эти права и полномочия передаются каждому члену группы. Однако администратор системы может изъять некоторые права и полномочия у некоторых членов группы. Максимальное количество групп, в которые может входить пользователь, равно 1000.

На платформе Windows NT различают три типа групп:

- глобальные группы;
- локальные группы;
- специальные группы.

Локальные и глобальные группы создаются администратором системы. Специальные группы создаются системой по умолчанию. Рассмотрим каждый из этих типов подробнее.

Глобальная группа используется для организации пользователей с целью упорядочения их доступа к ресурсам, находящимся в локальной сети вне домена, в котором создана эта группа. Глобальная группа имеет следующие характеристики:

- может содержать учетные записи только пользователей из домена, в котором она создается;
- может быть создана только на первичном контроллере домена;
- по умолчанию не имеет прав и привилегий.

По умолчанию система создает следующие глобальные группы:

- Domain Admins — эта группа включает учетные записи всех администраторов внутри домена;
- Domain Guests — эта группа включает учетные записи всех гостей домена;
- Domain Users — эта группа учетных записей всех пользователей домена.

Локальная группа используется для организации доступа пользователей к ограниченному множеству ресурсов внутри домена. Локальная группа имеет следующие характеристики:

- может содержать учетные записи пользователей и глобальных групп из разных доменов;
- может быть создана на любом контроллере домена;

- обычно получает права и привилегии на доступ к некоторым ресурсам домена.

По умолчанию система создает следующие локальные группы:

- Account Operators (операторы учетных записей) — члены этой группы могут создавать новые учетные записи пользователей;
- Administrators (администраторы) — члены этой группы наделяются неограниченными полномочиями администратора системы;
- Backup Operators (операторы резервирования) — членам этой группы разрешается создавать резервные копии файлов системы;
- Guests (гости) — эта группа предназначена для ограничения полномочий разовых пользователей системы, члены этой группы имеют такие же права, как и члены группы Users;
- Print Operators (операторы принтеров) — члены этой группы ответственны за управление принтерами системы: они могут добавлять, удалять и изменять принтеры (на сервере), используемые пользователями сети;
- Power Users (полномочные пользователи) — члены этой группы обладают широкими правами, которые можно сравнить с административными правами, но не имеют власти над администраторами системы;
- Replicator (репликаторы) — членам этой группы разрешается выполнять копирование каталогов, содержащих управляющую информацию, между компьютерами локальной сети;
- Server Operator (операторы сервера) — члены этой группы могут управлять сервером, однако они не имеют полномочий, сравнимых с администраторами системы;
- Users (пользователи) — члены этой группы являются зарегистрированными пользователями системы, они могут создавать другие локальные группы пользователей и управлять ими.

В операционной системе Windows XP дополнительно создаются следующие группы:

- Network Configuration Operators (операторы сети) — могут устанавливать компоненты и изменять конфигурацию локальной сети;
- Remote Desktop Users (удаленные пользователи) — обеспечивают доступ к компьютеру через Remote Desktop Connection;
- Help Services Group (группа обслуживающего персонала) — разрешают обслуживающему персоналу подключиться к компьютеру.

Специальные группы создаются системой для управления доступом к ресурсам. Членство в этих группах предопределено и не может быть изменено. Существуют следующие специальные группы:

- Creator/Owner (создатели и владельцы) — членами этой группы являются владельцы объектов;

- Everyone (любые учетные записи) — членами этой группы являются все учетные записи, зарегистрированные в системе, включая анонимную и пустую учетные записи;
- Interactive (интерактивные пользователи) — членами этой группы являются учетные записи, которые соответствуют пользователям, интерактивно работающим в системе;
- Network (сетевые пользователи) — членами этой группы являются учетные записи пользователей, которые работают с системой через локальную сеть;
- System (системные процессы) — эта специальная группа используется самой системой.

36.5. Идентификаторы безопасности

Для каждой учетной записи операционная система создает *идентификатор безопасности* (Security Identifier, SID), который хранится в базе данных менеджера учетных записей SAM. Идентификатор безопасности является бинарным представлением учетной записи и используется системой безопасности при своей работе для идентификации учетных записей. Фактически идентификатор безопасности идентифицирует пользователя на уровне системы безопасности. Использование идентификатора безопасности ускоряет работу системы безопасности, т. к. в этом случае система при идентификации пользователей работает с числовыми, а не символьными данными.

Символически структура идентификатора безопасности может быть описана следующим образом:

```
S - R - I - SA0 - SA1 - SA2 -SA3 - SA4 ...
```

Здесь каждый символ обозначает группу бит, имеющих определенное значение, а именно:

- `S` — представляет символ S, который обозначает, что дальнейшее числовое значение является идентификатором безопасности;
- `R` — представляет версию (Revision Level) формата идентификатора безопасности, начиная с операционной системы Windows NT версии 3.1, формат идентификатора безопасности не изменялся и поэтому значение R всегда равно 1;
- `I` — представляет 48-битное число, которое обозначает уровень авторизации учетной записи (Top-level Authority или Identifier Authority), которая связана с данным идентификатором безопасности. Это значение также называется *идентификатором авторизации учетной записи*;
- `SA` — представляет 32-битное число, которое уточняет уровень авторизации учетной записи (Subauthority), связанной с данным идентификато-

ром безопасности. Это число также называется *относительным идентификатором учетной записи* (Relative Identifier, RID).

Здесь под уровнем авторизации понимается уровень, на котором была создана учетная запись, или, другими словами, множество учетных записей, которому принадлежит учетная запись. Предопределенные значения уровней авторизации учетных записей приведены в табл. 36.1.

В свою очередь, относительные идентификаторы учетной записи предназначены для конкретизации или, другими словами, однозначной идентификации учетных записей.

В общем случае количество битовых полей типа SA в идентификаторе безопасности может быть произвольным. В нашем конкретном случае рассмотрим структуру идентификатора безопасности, который имеет только пять битовых полей типа SA:

$$S - R - I - SA_0 - SA_1 - SA_2 - SA_3 - SA_4$$

Такие идентификаторы безопасности создаются для пользователей и групп. В этом случае битовые поля типа SA имеют следующий смысл:

- поле SA_0 уточняет авторизацию учетной записи, которая связана с данным идентификатором безопасности;
- поля SA_1, SA_2 и SA_3 представляют уникальный 96-битовый идентификатор компьютера, на котором установлена система;
- поле SA_4 нумерует идентификаторы безопасности, создаваемые внутри системы. Номера от 0 до 999 зарезервированы для использования системой. А номера, начиная с 1000, присваиваются новым идентификаторам безопасности, при этом значение SA_4 увеличивается на 1 при создании каждого нового идентификатора безопасности для пользователя или группы внутри системы.

В табл. 36.1—36.4 приведены символьные представления некоторых предопределенных констант и идентификаторов безопасности, используемых в операционных системах Windows. При этом отметим, что идентификаторы безопасности предопределенных групп обычно имеют следующую структуру:

$$S - R - I - SA_0$$

В табл. 36.1 представлены предопределенные универсальные идентификаторы безопасности. Здесь слово "универсальный" означает, что эти идентификаторы безопасности имеют смысл в любой системе, которая использует дискреционную модель безопасности.

В табл. 36.2 представлены предопределенные универсальные идентификаторы учетной записи (Identifier Authority), которые определены в системах с дискреционной политикой безопасности.

Таблица 36.1. *Предопределенные универсальные SID*

Предопределенные универсальные SID	Символьное значение	Уровень авторизации учетной записи
Null SID	S-1-0-0	Группа без членов
World	S-1-1-0	Группа, которая включает всех пользователей
Local	S-1-2-0	Локальные пользователи системы
Creator Owner ID	S-1-3-0	Идентификатор безопасности владельца объекта
Creator Group ID	S-1-3-1	Идентификатор безопасности группы, которой принадлежит владелец объекта

Таблица 36.2. *Предопределенные значения типа учетной записи*

Идентификаторы учетной записи	Значение	Префикс SID
SECURITY_NULL_SID_AUTHORITY	0	S-1-0
SECURITY_WORLD_SID_AUTHORITY	1	S-1-1
SECURITY_LOCAL_SID_AUTHORITY	2	S-1-2
SECURITY_CREATOR_SID_AUTHORITY	3	S-1-3
SECURITY_NT_AUTHORITY	5	S-1-5

Первые четыре идентификатора учетной записи из табл. 36.2 используются в предопределенных универсальных дескрипторах безопасности. Последний идентификатор используется в дескрипторах безопасности, известных на платформах Windows. Эти дескрипторы безопасности приведены в табл. 36.3.

Таблица 36.3 представляет символьные обозначения для относительных идентификаторов (RID), используемых с предопределенным идентификатором безопасности SECURITY_NT_AUTHORITY (S-1-5) для создания SID, которые не являются универсальными, но известны на платформе Windows.

Таблица 36.3. *Дескрипторы безопасности, известные на платформах Windows*

Предопределенные RID	Известные SID	Уровень авторизации учетной записи
SECURITY_DIALUP_RID	S-1-5-1	Группа, члены которой вошли в систему с терминалов, используя dial-up модем

Таблица 36.3 *(окончание)*

Предопределенные RID	Известные SID	Уровень авторизации учетной записи
SECURITY_NETWORK_RID	S-1-5-2	Группа, члены которой вошли в систему через сеть
SECURITY_BATCH_RID	S-1-5-3	Группа, члены которой могут исполнять пакетные задания
SECURITY_INTERACTIVE_RID	S-1-5-4	Группа, членам которой разрешается выполнять интерактивные операции
SECURITY_LOGON_IDS_RID	S-1-5-5-X-Y	Учетная запись для идентификации сессии пользователя, взаимодействующего с системой (logon session). Значения X и Y отличаются для каждой сессии
SECURITY_SERVICE_RID	S-1-5-6	Учетная запись сервиса
SECURITY_ANONYMOUS_LOGON_RID	S-1-5-7	Анонимная или пустая учетная запись
SECURITY_PROXY_RID	S-1-5-8	Прокси
SECURITY_ENTERPRISE_ CONTROLLERS_RID	S-1-5-9	Контроллер предприятия
SECURITY_PRINCIPAL_SELF_RID	S-1-5-10	Используется для замещения идентификатора безопасности владельца объекта при его наследовании
SECURITY_AUTHENTICATED_ USER_RID	S-1-5-11	Учетная запись аутентифицированного пользователя
SECURITY_RESTRICTED_CODE_RID	S-1-5-12	Ограниченный код
SECURITY_TERMINAL_SERVER_RID	S-1-5-13	Учетная запись пользователя, который вошел на терминальный сервер
SECURITY_LOCAL_SYSTEM_RID	S-1-5-18	Специальная учетная запись, используемая операционной системой
SECURITY_NT_NON_UNIQUE	S-1-5-21	
SECURITY_BUILTIN_DOMAIN_RID	S-1-5-32	Учетная запись встроенного системного домена

В табл. 36.4 приведены предопределенные относительные идентификаторы учетной записи, которые используются в универсальных дескрипторах безопасности SID.

Таблица 36.4. *Предопределенные относительные идентификаторы учетной записи*

Относительные идентификаторы учетной записи	Значение	Идентификаторы учетной записи
SECURITY_NULL_RID	0	S-1-0
SECURITY_WORLD_RID	0	S-1-1
SECURITY_LOCAL_RID	0	S-1-2
SECURITY_CREATOR_OWNER_RID	0	S-1-3
SECURITY_CREATOR_GROUP_RID	1	S-1-3

В заключение этого раздела скажем, что текстовые представления установленных идентификаторов безопасности можно просмотреть в ключе реестра HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\ProfileList. Щелкнув кнопкой мыши на нужном идентификаторе безопасности, можно увидеть учетную запись, связанную с этим идентификатором безопасности.

36.6. Дескрипторы безопасности

Как уже говорилось в начале главы, каждый охраняемый объект имеет *дескриптор безопасности* (security descriptor) или сокращенно SD, который создается вместе с охраняемым объектом и содержит информацию, необходимую для защиты объекта от несанкционированного доступа. Дескриптор безопасности включает заголовок (header), который содержит управляющие флаги и указатели на следующие компоненты:

- SID владельца объекта;
- SID первичной группы владельца объекта;
- указатель на список управления дискреционным доступом — DACL;
- указатель на список управления системным доступом — SACL.

Идентификаторы безопасности владельца объекта и первичной группы, идентифицируют пользователя, который создает объект. Структура идентификаторов безопасности была рассмотрена в предыдущем разделе. Здесь же только поясним, что такое первичная группа.

Каждый пользователь, т. е. его учетная запись, может принадлежать нескольким группам, но одна из них должна быть выбрана в качестве *первичной группы* пользователя (primary group). Это сделано для того, чтобы ресурсы, создаваемые пользователем, были доступны другим членам группы, к

которой принадлежит этот пользователь. Такая политика доступа к ресурсам определяется в приложениях, совместимых со стандартом POSIX, или работающих на базе компьютеров Macintosh, которые подключены к локальной сети, работающей под управлением сервера Windows. Поэтому дескриптор безопасности субъекта и содержит идентификатор безопасности первичной группы. В операционных системах Windows в качестве первичной группы может быть выбрана любая глобальная группа домена или просто пользователь этого домена. По умолчанию для каждого пользователя, система в качестве первичной группы определяет глобальную группу Domain Users. Поэтому чтобы удалить пользователя из этой группы, нужно сначала включить его в другую глобальную группу, а затем сделать эту группу его первичной группой.

Отметим, что в операционных системах Windows для работы с дескрипторами безопасности охраняемых объектов используются функции `SetNamedSecurityInfo`, `SetSecurityInfo`, `GetNamedSecurityInfo` и `GetSecurityInfo`. Небольшое ограничение существует только для объектов ядра, а именно, функции `GetNamedSecurityInfo` и `SetNamedSecurityInfo` работают только со следующими объектами ядра: семафорами, событиями, мьютексами, ожидающими таймерами и отображениями файлов в память.

В заключение этого раздела скажем, что списки управления дискреционным и системным доступом будут более подробно рассмотрены в следующем разделе.

36.7. Списки управления доступом ACL

Список управления доступом (ACL) содержит элементы, которые называются *входами управления доступом* (Access-Control Entries, ACE). Каждый элемент ACE содержит следующую информацию:

- идентификатор безопасности субъекта, которому разрешен или запрещен доступ к охраняемому объекту;
- маску доступа, которая специфицирует права доступа субъекта к охраняемому объекту;
- флаг, который определяет тип элемента ACE;
- флаги, которые определяют свойства наследования данного элемента ACE;
- флаги, которые управляют аудитом доступа к охраняемому объекту.

Идентификаторы безопасности определяют учетную запись пользователя, от имени которого выступает субъект. Структура идентификаторов безопасности была рассмотрена в предыдущем разделе. Теперь рассмотрим подробно остальные компоненты элемента ACE из списка управления дискреционным доступом.

Маска доступа представляет собой 32-битовое поле, в котором каждый бит специфицирует определенное право доступа субъекта к охраняемому объекту, за исключением 24—27 битов, которые зарезервированы для дальнейшего использования. Каждое право доступа представляет собой набор операций, которые субъекту разрешается или запрещается выполнять над охраняемым объектом. В Windows права доступа также называются *разрешениями* (permissions). В операционных системах Windows NT определены следующие четыре категории прав доступа:

- Specific Access Rigths — специфические права доступа;
- Standard Access Rights — стандартные права доступа;
- Generic Access Rights — родовые права доступа;
- SACL Access Right — право доступа к списку управления системным доступом (SACL);

Рассмотрим каждую из этих категорий прав доступа к охраняемому объекту более подробно.

Специфические права доступа определяют операции, которые могут выполняться только над объектом заданного типа. Поэтому для каждого объекта определяется свой набор специфических прав доступа. Специфические права доступа устанавливаются в 0—15 битах маски доступа.

Стандартные права доступа содержат такие операции, которые можно выполнять над большинством охраняемых объектов. Права этого типа включают следующие права:

- `READ_CONTROL` — право читать информацию из дескриптора безопасности объекта, за исключением информации из списка SACL;
- `WRITE_DAC` — право модифицировать список DACL в дескрипторе безопасности охраняемого объекта;
- `WRITE_OWNER` — право изменить владельца объекта в дескрипторе безопасности охраняемого объекта;
- `DELETE` — право удалять охраняемый объект;
- `SYNCHRONIZE` — право использовать объект для синхронизации.

Стандартные права доступа устанавливаются в 16—22 битах маски доступа.

Родовые права включают следующие четыре права:

- `GENERIC_READ` — право читать содержимое объекта;
- `GENERIC_READ` — право записи в объект;
- `GENERIC_EXECUTE` — право исполнения объекта;
- `GENERIC_ALL` — объединяет три предыдущих права.

Каждое из этих прав отображается в подходящие стандартные и специфические права доступа для каждого конкретного охраняемого объекта. Родовые права доступа устанавливаются в 28—31 битах маски доступа.

Для списка управления системным доступом (SACL) определено единственное право:

- `ACCESS_SYSTEM_SECURITY` — право доступа к списку SACL дескриптора безопасности.

Оно дает субъекту возможность просматривать и изменять содержимое списка SACL. Этот флаг устанавливается в 23-ем бите маски доступа.

В операционной системе Windows 2000 также определены права для доступа к объектам активного каталога (Active Directory), которые здесь рассматриваться не будут.

Теперь перейдем к флагу, который определяет тип элемента списка управления доступом. Элементы списка DACL могут быть двух типов:

- `ACCESS_ALLOWED_ACE` — элемент, разрешающий доступ субъекта к охраняемому объекту;
- `ACCESS_DENIED_ACE` — элемент, запрещающий доступ субъекта к охраняемому объекту.

Причем субъект, доступ которого к охраняемому объекту контролирует элемент списка DACL, задается идентификатором безопасности, который хранится в этом же элементе списка ACE. А операции, которые субъекту разрешается или запрещается выполнять над объектом, определяются флагами, отмечающими права доступа.

Список SACL может содержать только элементы типа:

- `SYSTEM_AUDIT_ACE` — элемент списка SACL.

Каждый из этих элементов указывает системе, что нужно создать аудиторскую запись при попытке выполнения над объектом операции, разрешенной специфицированными в этом же элементе списка правами доступа к охраняемому объекту.

Последняя компонента элемента списка управления доступом содержит флаги, которые задают свойства наследования данного элемента ACE. То есть эти флаги определяют, наследуется ли элемент ACE списка управления доступом для вновь создаваемого охраняемого объекта, который является дочерним по отношению к текущему охраняемому объекту. Например, пусть в качестве охраняемого объекта выступает каталог, и элемент ACE списка управления доступов в дескрипторе безопасности этого объекта является наследуемым. Тогда копия этого элемента будет помещена в соответствующий список управления доступом вновь создаваемого подкаталога в рассматриваемом каталоге. Если же элемент ACE ненаследуемый, то этого

не произойдет. Для управления наследованием элементов списка управления доступом используются следующие флаги:

- `OBJECT_INHERIT_ACE` — элемент наследуется неконтейнерным дочерним объектом. Если не установлен флаг `NO_PROPAGATE_INHERIT_ACE`, то элемент также наследуется и контейнерным дочерним объектом, но при этом в дочернем объекте устанавливается флаг `INHERIT_ONLY_ACE`;
- `CONTAINER_INHERIT_ACE` — элемент наследуется только контейнерным дочерним объектом;
- `NO_PROPAGATE_INHERIT_ACE` — элемент был унаследован от родительского объекта, но флаги `OBJECT_INHERIT_ACE` и `CONTAINER_INHERIT_ACE` в элементе сбрасываются, что отменяет дальнейшее наследование этого элемента;
- `INHERIT_ONLY_ACE` — отмечает, что элемент был унаследован от родительского объекта; этот элемент не участвует в контроле доступа к объекту;
- `INHERITED_ACE` — отмечает, что элемент был унаследован от родительского объекта.

При этом сделаем следующее замечание — операционная система Windows 2000 автоматически выполняет распространение наследуемых элементов списка управления доступом в списки управления доступом дочерних объектов, если эти элементы включаются в список управления доступом при помощи функций `SetNamedSecurityInfo` и `SetSecurityInfo`. То есть если элемент с установленным флагом наследования `CONTAINER_INHERIT_ACE` включается в список управления доступом к каталогу, то этот элемент будет включен также в списки управления доступом всех подкаталогов этого каталога.

Теперь приведем правила, следуя которым выполняется автоматическое наследование элементов списка управления DACL:

- если дочерний объект имеет список DACL, то в этот список включаются наследуемые элементы списка DACL родительского объекта;
- если дочерний объект не имеет списка DACL, то этот список создается и в него включаются наследуемые элементы списка DACL родительского объекта;
- если наследуемый элемент списка DACL удаляется из родительского объекта, то он удаляется также из списков DACL дочерних объектов;
- если список DACL дочернего объекта содержит только наследуемые элементы и все эти элементы удаляются из списка, то сам список не удаляется, а становится пустым.

Особенно стоит обратить внимание на последнее правило, т. к. при доступе субъекта к объекту система различает случаи отсутствия списка DACL и наличие пустого списка DACL, т. е. списка DACL, который не содержит элементов. В первом случае доступ к объекту открыт для всех субъектов, а во втором — доступ к объекту закрыт для всех субъектов. Более подроб-

но о контроле доступа субъектов к охраняемому объекту будет сказано в *разд. 36.9*.

В заключение этого раздела рассмотрим флаги, которые управляют аудитом доступа к охраняемому объекту. Определены два флага, управляющие аудитом, а именно:

- `SUCCESSFUL_ACCESS_ACE_FLAG` — отмечает, что нужно генерировать аудиторское сообщение при успешном открытии доступа к объекту;
- `FAILED_ACCESS_ACE_FLAG` — отмечает, что нужно генерировать аудиторское сообщение при неудачной попытке открытия доступа к объекту.

Отметим, что эти флаги используются только в элементах, принадлежащих спискам управления доступом SACL, и могут быть установлены как по отдельности, так и совместно.

36.8. Маркеры доступа

Каждый субъект, которым в Windows может быть поток или процесс, имеет маркер доступа, который идентифицирует субъекта при попытке его доступа к объекту. Поэтому также говорят, что маркер доступа описывает *контекст безопасности субъекта*, т. е. ограничивает доступ субъекта к объектам. В маркере доступа хранится информация, идентифицирующая пользователя, от имени которого исполняется поток или процесс, и привилегии, которыми обладает субъект. Вообще *привилегией* называется право пользователя выполнить некоторое действие по отношению к некоторым объектам или субъектам системы. Главное отличие привилегий от прав доступа заключается в том, что, во-первых, привилегии касаются субъектов, а не охраняемых объектов системы; а во-вторых, привилегии назначаются субъектам администратором системы, а правами доступа к объекту управляет владелец этого объекта.

В табл. 36.5 перечислены привилегии и встроенные учетные записи, которым эти привилегии назначаются по умолчанию в операционных системах Windows.

Таблица 36.5. *Привилегии, назначаемые учетным записям по умолчанию*

Привилегия	Строковая константа, именующая привилегию	Учетные записи, которым привилегия назначается по умолчанию	Описание привилегии
Replace a process-level token	`SeAssignPrimaryTokenPrivilege`	Empty	Разрешает пользователю заменять маркер доступа процесса

Таблица 36.5 *(продолжение)*

Привилегия	Строковая константа, именующая привилегию	Учетные записи, которым привилегия назначается по умолчанию	Описание привилегии
Generate Security Audits	SeAuditPrivilege	Empty	Разрешает пользователю создавать элементы в списке SACL для аудита доступа к объекту
Back up Files and Directories	SeBackupPrivilege	Administrators Backup Operators Server Operators	Разрешает пользователю создавать резервные копии файлов и каталогов
Log on as a Batch Job	SeBatchLogonRight	Empty	Разрешает пользователю запустить пакетное задание из очереди заданий
Bypass Traverse Checking	SeChangeNotifyPrivilege	Administrators Power Users Backup Operators Users Everyone	Разрешает пользователю получать извещения об изменениях в каталогах или файлах. Разрешает пользователю продвигаться по каталогам, чтобы достичь каталога или файла, к которому он имеет доступ. Содержимое каталогов во время продвижения просматривать нельзя
Create a Pagefile	SeCreatePagefile Privilege	Administrators	Разрешает пользователю создавать и изменять размеры файла подкачки страниц
Create Permanent Shared Objects	SeCreatePermanent Privilege	Empty	Разрешает пользователю создать постоянный объект, который может совместно использоваться другими пользователями
Create a Token Object	SeCreateTokenPrivilege	Local System	Разрешает процессу, работающему от имени учетной записи пользователя, создавать маркер доступа, который разрешает доступ к любым ресурсам на локальном компьютере

Таблица 36.5 *(продолжение)*

Привилегия	Строковая константа, именующая привилегию	Учетные записи, которым привилегия назначается по умолчанию	Описание привилегии
Debug Programs	SeDebugPrivilege	Administrators	Разрешает пользователю присоединять отладчик к любому процессу
Increase Scheduling Priority	SeIncreaseBasePriority Privilege	Administrators	Разрешает процессу увеличить приоритет другого процесса
Increase Quotas	SeIncreaseQuotaPrivilege	Administrators	Разрешает процессу увеличивать квоту процессорного времени, выделяемого другому процессу
Log on Locally	SeInteractiveLogonRight	Account Operators Administrators Backup Operators Print Operators Server Operators Everyone Guests Power Users Users	Разрешает пользователю интерактивный вход в систему
Load and Unload Device Drivers	SeLoadDriverPrivilege	Administrators	Разрешает пользователю устанавливать и удалять драйверы устройств
Lock Pages in Memory	SeLockMemoryPrivilege	Empty	Разрешает процессу запирать виртуальные страницы в реальной памяти. Эта привилегия считается устаревшей
Add Workstations to the Domain	SeMachineAccount Privilege	Empty	Разрешает пользователю добавлять компьютер к домену
Access this Computer from the Network	SeNetworkLogonRight	Administrators Power Users Backup Operators Users Everyone	Разрешает доступ к компьютеру из локальной сети

Таблица 36.5 *(продолжение)*

Привилегия	Строковая константа, именующая привилегию	Учетные записи, которым привилегия назначается по умолчанию	Описание привилегии
Profile Single Process	SeProfSingleProcess	Administrators	Разрешает пользователю настраивать производительность процесса
Force Shutdown from a Remote System	SeRemoteShutdown Privilege	Administrators Server Operators Power Users	Разрешает пользователю выключать компьютер с другого компьютера в сети
Restore files and directories	SeRestorePrivilege	Administrators Backup Operators Server Operators	Разрешает пользователю восстанавливать содержимое сохраненных файлов и каталогов
Manage Auditing and Security Log	SeSecurityPrivilege	Administrators	Разрешает пользователю проводить аудит доступа к объектам
Log on as a Service	SeServiceLogonRight	Empty	Разрешает сервису работать от имени пользователя
Shut down the system	SeShutdownPrivilege	Account Operators Administrators Backup Operators Print Operators Server Operators Everyone Guests Poser Users Users	Разрешает пользователю выключать систему
Modify Firmware Environment Variables	SeSystemEnvironment Privilege	Administrators	Разрешает пользователю изменять значения переменных системного окружения
Profile System Performance	SeSystemProfile Privilege	Administrators	Разрешает настраивать производительность системы

Таблица 36.5 *(окончание)*

Привилегия	Строковая константа, именующая привилегию	Учетные записи, которым привилегия назначается по умолчанию	Описание привилегии
Change the System Time	SeSystemTimePrivilege	Administrators Power Users Server Operators	Разрешает пользователю устанавливать системное время и дату
Take ownership of files or other objects	SeTakeOwnership Privilege	Administrators	Разрешает пользователю становиться владельцем объекта без разрешения текущего владельца этого объекта
Act as Part of the Operating System	SeTcbPrivilege	Local System	Разрешает процессу работать от имени любого пользователя и иметь доступ к его ресурсам

Теперь рассмотрим структуру маркера доступа. В маркере доступа хранится следующая информация:

- SID учетной записи пользователя;
- список SID учетных записей групп, которым принадлежит пользователь;
- идентификатор безопасности текущей сессии (logon session);
- список привилегий, которыми обладает пользователь и группы, в которые он входит, на локальном компьютере;
- идентификатор безопасности пользователя или группы, который используется по умолчанию для задания владельца вновь создаваемого или существующего охраняемого объекта;
- идентификатор безопасности первичной группы пользователя;
- список DACL, которую система использует по умолчанию при создании охраняемого объекта;
- идентификатор процесса, который вызвал создание маркера доступа;
- значение, определяющее тип маркера доступа: первичный маркер или маркер, замещающий первичный маркер маркером другого пользователя (impersonation token);
- список SID, которые ограничивают доступ потока к охраняемым объектам, так называемые ограничивающие SID (restricting SID);

- значение, которое отмечает уровни замещения сервером маркера доступа клиента (impersonation levels);
- статистическая информация о маркере доступа.

Поясним два понятия, которые встречаются в описании маркера доступа, а именно, работа одного потока в контексте безопасности другого потока и ограничение контекста безопасности процесса. О первичной группе пользователя было сказано в параграфе, посвященном дескрипторам безопасности.

Прежде всего скажем о персонификации маркеров доступа. Каждый поток имеет свой маркер доступа, который, как уже было сказано, определяет контекст безопасности этого потока или, другими словами, охраняемые объекты, к которым поток имеет доступ. Теперь представим себе, что поток-клиент посылает по сети потоку-серверу запрос на выполнение некоторой операции. Для того чтобы сервер не вышел за пределы контекста безопасности потока-клиента, он должен работать от имени клиента, а не от своего имени. Такая замена контекста безопасности потока называется *подменой контекста безопасности.* Подмена контекста безопасности потока может выполняться только при помощи подмены маркера доступа потока-сервера на маркер доступа потока-клиента. Поэтому маркер доступа и содержит информацию о том, является ли он первичным или замещается маркером другого пользователя. Подмену маркера доступа потока-сервера на маркер доступа потока-клиента будем также назвать *замещением маркера доступа* (impersonation) Исходя из этих рассуждений, можно также сказать, что замещение маркера доступа это способность одного процесса использовать атрибуты защиты другого процесса.

Поток-сервер может обрабатывать запросы потока-клиента на четырех уровнях безопасности, каждый их которых указывает серверу, до какой степени он может исполнять роль клиента. Далее перечислены эти уровни:

- Security Anonymous level — анонимный уровень, т. е. сервер не может идентифицировать клиента, который остается анонимным;
- Security Identification level — идентифицирующий уровень, на этом уровне клиент может идентифицировать клиента, т. е. может получить любую информацию из идентификатора безопасности (SID) клиента, но не может выполнять от имени клиента никаких действий;
- Security Impersonation level — уровень подмены контекста безопасности, в этом случае поток-сервер может работать от имени потока-клиента. Если поток-сервер работает на удаленном компьютере, то он имеет доступ только к локальным ресурсам на этом компьютере. В противном случае поток-сервер имеет доступ как к локальным, так и к сетевым ресурсам, входящим в контекст безопасности потока-клиента;
- Security Delegation level — уровень делегирования, на этом уровне поток-клиент полностью делегирует свои полномочия потоку-серверу, который может распоряжаться ими как на локальном, так и на удаленном компь-

ютере. Этот уровень передачи полномочий реализован только в операционных системах Windows 2000/XP.

Теперь рассмотрим *ограничивающие SID* (restricting SID), само название которых говорит о том, что они используются для ограничения доступа субъекта к объектам. Это ограничение выполняется следующим образом. Система управления безопасностью при контроле доступа субъекта к охраняемому объекту проверяет не только идентификатор безопасности этого субъекта, но и его ограничивающие идентификаторы безопасности.

В заключение этого раздела скажем, что маркеры доступа сами являются охраняемыми объектами и поэтому для них также нужно задавать атрибуты безопасности. Доступ к дескриптору безопасности маркера доступа осуществляется так же, как и доступ к объекту ядра при помощи функций `SetKernelObjectSecurity` и `GetKernelObjectSecurity`.

36.9. Создание новых объектов

При создании нового объекта поток, создающий этот объект, может предоставить атрибуты безопасности этого объекта, а может и не предоставить. Атрибуты безопасности объекта задаются структурой `SECURITY_ATTRIBUTES`, которая имеет следующий формат:

```
typedef struct _SECURITY_ATTRIBUTES {
  DWORD nLength;                // длина структуры в байтах
  LPVOID lpSecurityDescriptor; // указатель на дескриптор безопасности
  BOOL bInheritHandle;          // признак наследования
} SECURITY_ATTRIBUTES, *PSECURITY_ATTRIBUTES, *LPSECURITY_ATTRIBUTES;
```

Адрес этой структуры передается как параметр функции создания нового охраняемого объекта. Как видно из определения, структура `SECURITY_ATTRIBUTES` содержит только адрес дескриптора безопасности объекта и флаг наследования этого объекта дочерними процессами. О наследовании объектов было рассказано в *разд. 4.4*. Теперь же поговорим о том, как система устанавливает дескриптор безопасности объекта.

Если структура типа `SECURITY_ATTRIBUTES` содержит указатель на дескриптор безопасности, то этот дескриптор безопасности и устанавливается для нового объекта. Если же дескриптор безопасности не задан, то функция создания объекта все равно строит дескриптор безопасности этого объекта, но его содержимое определяется системой. Для этого и предназначены идентификаторы безопасности владельца объекта, первичной группы владельца объекта и список DACL из маркера доступа, которые используются по умолчанию при создании новых объектов. Кратко рассмотрим, как назначаются владелец объекта и список доступа DACL объекта, если они не заданы явно в дескрипторе доступа при создании объекта.

Если владелец объекта не задан, то владельцем объекта становится пользователь, заданный как владелец объекта по умолчанию в маркере доступа, при условии, что учетная запись этого пользователя связана с маркером доступа и для этого пользователя установлена привилегия `SeTakeOwnershipPrivilege`. В противном случае владельцем объекта становится учетная запись, от имени которой работает поток, создающий объект.

Если объект имеет имя, то, начиная с операционной системы Windows 2000, список управления DACL этого объекта всегда содержит наследуемые элементы из списка управления доступом родительского контейнерного объекта. Если наследуемых элементов нет или объект не имеет имени, то в дескриптор безопасности объекта включается список DACL по умолчанию, который задан в маркере доступа потока, создающего объект. Если такого списка нет, то дескриптор безопасности объекта также не содержит список DACL, что открывает всем субъектам неограниченный доступ к этому объекту.

36.10. Контроль доступа к охраняемому объекту

Опишем общую схему контроля доступа субъектов к объектам. Для этого напомним, что в операционных системах Windows в качестве субъекта выступает процесс или поток, который исполняется от имени некоторого пользователя. С каждым процессом ассоциируется маркер доступа, который идентифицирует учетную запись пользователя, от имени которого выполняется этот процесс. Маркер доступа связывается с потоком во время его запуска при входе пользователя в систему. Но фактически доступ к объектам осуществляет поток, который выполняется в контексте этого процесса. Каждый поток также имеет свой маркер доступа, который может быть первичным, т. е. совпадать с маркером доступа процесса, в контексте которого выполняется этот поток, или заимствованным у другого процесса, в случае если произошла подмена контекста безопасности (impersonation). Доступ к охраняемому объекту выполняется из потока. Поэтому под субъектом часто и понимается исполняемый поток, который при контроле доступа представляется маркером доступа этого потока. В свою очередь охраняемый объект, доступ к которому контролируется системой управления безопасностью, представляется при контроле доступа дескриптором безопасности этого объекта.

Теперь можно перейти к общей схеме контроля доступа субъектов к объектам. Контроль доступа субъекта к охраняемому объекту выполняется следующим образом. При открытии субъектом доступа к охраняемому объекту, что обычно выполняется посредством функций типа `Create` или `Open`, система управления безопасностью просматривает список DACL этого охраняемого объекта для поиска элемента, в котором хранится идентификатор безопасности субъекта. Если такой элемент в списке DACL не найден,

то поток получает отказ в доступе к объекту. Если же такой элемент найден, то система проверяет тип этого элемента. Если найденный элемент имеет тип `ACCESS_ALLOWED_ACE`, то система безопасности проверяет, установлены ли в этом элементе флаги прав доступа субъекта, которые соответствуют запрашиваемому потоком доступу. Если такие флаги установлены, то поток получает доступ к охраняемому объекту, в противном случае поток получает отказ в доступе к охраняемому объекту. Если же найденный элемент имеет тип `ACCESS_DENIED_ACE`, то система безопасности проверяет, установлены ли флаги прав доступа субъекта, которые соответствуют запрашиваемому потоком доступу. Если хотя бы один из таких флагов установлен, то поток получает отказ в доступе к охраняемому объекту, в противном случае система продолжает просмотр списка управления доступом DACL.

Как видно из изложенного, доступ потока к охраняемому объекту зависит от порядка расположения элементов в списке управления доступом DACL этого объекта. Система управления безопасностью включает элементы типа `ACCESS_DENIED_ACE` в начало списка, если это включение выполняется при помощи функции `SetEntriesInAcl`. Однако низкоуровневые функции для работы со списками доступа, такие как `AddAccessAllowedAce`, `AddAccessDeniedAce` и `AddAce`, не обеспечивают такую последовательность элементов в списках. При работе с такими функциями программа сама должна контролировать порядок элементов в списках управления доступом.

Кроме того, доступ к охраняемому объекту также зависит от того, в каком порядке в этот список DACL включаются наследуемые элементы из родительских объектов. В операционной системе Windows 2000 принят следующий порядок включения элементов в список DACL при автоматическом наследовании элементов. Наследуемые элементы типа `ACCESS_ALLOWED_ACE` включаются в самый конец списка DACL, после ненаследованных элементов такого же типа. Наследуемые элементы типа `ACCESS_DENIED_ACE` включаются после всех ненаследуемых элементов такого типа, но перед всеми элементами типа `ACCESS_ALLOWED_ACE`.

Заметим, что при определении права доступа потока к охраняемому объекту следует различать две ситуации:

- отсутствие списка DACL у объекта (Null DACL) — в этом случае доступ к объекту не ограничен и разрешен для любого потока;
- пустой список DACL у объекта (Empty DACL) — в этом случае доступ к объекту запрещен для всех потоков.

Такая интерпретация ситуации, в которой отсутствует список DACL, позволяет избежать затрат, связанных с работой системы безопасности, если пользователь не использует возможности, предоставляемые операционной системой для обеспечения безопасности объекта от несанкционированного доступа.

36.11. Аудит доступа к охраняемому объекту

Теперь перейдем к аудиту доступа субъектов к охраняемым объектам. Аудит доступа субъекта к охраняемому объекту выполняется следующим образом. При открытии субъектом доступа к охраняемому объекту, что обычно выполняется посредством функций типа `Create`, система управления безопасностью просматривает список управления доступом SACL этого объекта. Если в списке SACL найден элемент, который содержит идентификатор безопасности, совпадающий с идентификатором безопасности субъекта, и в этом элементе установлены флаги `SUCCESSFUL_ACCESS_ACE_FLAG` или `FAILED_ACCESS_ACE_FLAG` (или оба флага вместе), то система проверяет флаги управления доступом, установленные в этом элементе. Если запрашиваемый субъектом доступ соответствует правам пользователя, т. е. субъект получает доступ к охраняемому объекту, и в элементе списка SACL для затребованных субъектом прав доступа установлен флаг `SUCCESSFUL_ACCESS_ACE_FLAG`, то система генерирует аудиторское сообщение об успешном доступе субъекта к объекту. Или если субъекту отказано в доступе к охраняемому объекту и в элементе списка SACL для затребованных субъектом прав доступа установлен флаг `FAILED_ACCESS_ACE_FLAG`, то система генерирует аудиторское сообщение о неудачном доступе субъекта к объекту. В других случаях аудиторские сообщения не генерируются. При доступе субъекта к охраняемым объектам возможны следующие аудиторские сообщения:

- Object Open — объект открыт;
- Object Open for Delete — объект открыт для удаления;
- Object Deleted — объект удален.

Каждое сообщение также содержит информацию о субъекте, который выполнил действие, отмеченное этим сообщением. Аудиторские сообщения записываются в журнал сообщений (Event log). Доступ к этому журналу имеют администраторы системы посредством просмотра соответствующих ключей реестра.

36.12. Структура системы безопасности

В заключительном разделе этой главы кратко рассмотрим структуру системы безопасности операционных систем Windows. На рис. 36.1 приведена общая схема взаимодействия основных компонент системы безопасности операционных систем Windows. Дадим краткое описание и функциональное назначение каждой из компонент системы безопасности.

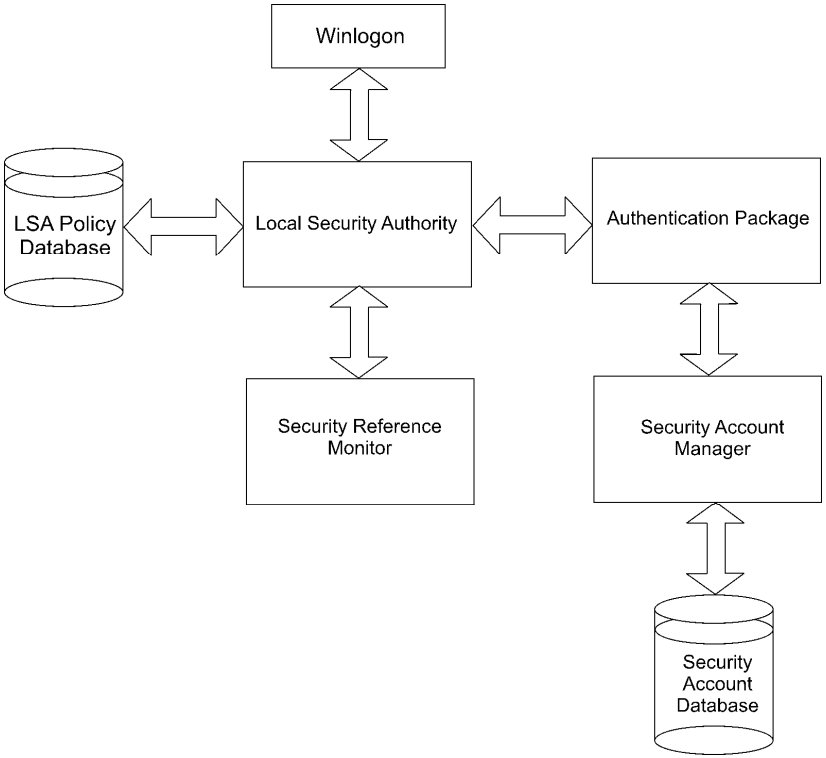


Рис. 36.1. Схема взаимодействия компонент системы безопасности

Сначала перечислим ключи реестра, в которых хранятся базы данных системы безопасности:

- Local Security Authority Policy Database (HKEY_LOCAL_MACHINE\SECURITY) — защищенная база данных, в которой хранятся данные, определяющие политику безопасности на локальной машине;
- Security Accounts Database (HKEY_LOCAL_MACHINE\SAM) — защищенная база данных, в которой хранятся учетные записи пользователей и групп, а также связанная с ними информация, такая как, например, пароли и идентификаторы безопасности.

Теперь перечислим основные программные компоненты системы безопасности и кратко опишем их функциональность:

- Security Reference Monitor (компонент NTOSKRNL.EXE) — контролирует доступ субъектов к охраняемым объектам, поддерживает аудит, является компонентом ядра операционной системы и работает в защищенном режиме (режиме ядра);

- Local Security Authority (компонент LSASS.EXE) — определяет, может ли пользователь войти в систему, и отвечает за исполнение политики безопасности на локальном компьютере;
- Security Accounts Manager (компонент LSASS.EXE) — является компонентом Local Security Authority и поддерживает базу данных учетных записей.

Кроме того, укажем вспомогательные процессы, которые участвуют в работе системы безопасности:

- Winlogon (WINLOGON.EXE) — запускается во время загрузки системы и управляет процессом входа пользователей в систему;
- Authentication Package (MSV1_0.DLL) — используется для верификации полномочий пользователя при доступе субъектов к объектам;
- Netlogon (SERVICES.EXE) — контролирует вход в сеть, обеспечивает безопасную передачу данных о пользователе контроллеру домена.

Глава 37

Управление пользователями

37.1. Создание учетной записи пользователя

Для создания учетной записи пользователя используется функция `NetUserAdd`, которая имеет следующий прототип:

```
NET_API_STATUS  NetUserAdd(
  LPCWSTR  servername,     // имя сервера
  DWORD    level,          // уровень информации
  LPBYTE   buf,            // указатель на информацию
  LPDWORD  parm_err        // индексация структуры с информацией
);
```

Отметим, что успешно эту функцию могут выполнить только те пользователи, которые являются администраторами или операторами учетных записей.

В случае успешного завершения функция `NetUserAdd` возвращает значение `NERR_Success`, а в случае неудачи возможны следующие коды завершения:

- `ERROR_ACCESS_DENIED` — пользователю отказано в доступе;
- `NERR_InvalidComputer` — неправильное имя компьютера;
- `NERR_NotPrimary` — операция может выполняться только на первичном контроллере домена;
- `NERR_GroupExists` — группа уже существует;
- `NERR_UserExists` — учетная запись уже существует;
- `NERR_PasswordTooShort` — пароль короче, чем требуется.

Параметры функции `NetUserAdd` имеют следующее назначение.

Параметр `servername` должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку

Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение NULL.

Параметр level указывает тип структуры, которая содержит информацию о пользователе. Этот параметр может принимать одно из следующих значений:

- 1 — используется структура типа USER_INFO_1;
- 2 — используется структура типа USER_INFO_2;
- 3 — используется структура типа USER_INFO_3;
- 4 — используется структура типа USER_INFO_4.

В примере будет использована структура типа USER_INFO_1, поэтому после описания функции приведем тип и опишем назначение полей этой структуры. Описание остальных структур может быть найдено в MSDN.

Параметр buf должен указывать на буфер с информацией о пользователе. Информация должна храниться в структуре, тип которой задан параметром level.

В параметр parm_err функция NetUserAdd помещает индекс первого поля в структуре с информацией о пользователе, которое содержит неправильные данные и, как следствие, вызвало завершение функции NetUserAdd с ошибкой. В этом параметре может быть установлено значение NULL. В этом случае индекс поля с неправильными данными не возвращается.

Теперь опишем структуру USER_INFO_1, которая будет использоваться в примере. Эта структура имеет следующий тип:

```
typedef  struct  _USER_INFO_1  {
  LPWSTR  usri1_name;            // имя пользователя
  LPWSTR  usri1_password;        // пароль пользователя
  DWORD   usri1_password_age;    // возраст пароля
  DWORD   usri1_priv;            // уровень привилегий
  LPWSTR  usri1_home_dir;        // домашний каталог
  LPWSTR  usri1_comment;         // комментарий
  DWORD   usri1_flags;           // флаги
  LPWSTR  usri1_script_path;     // путь к скрипту
}USER_INFO_1, *PUSER_INFO_1, *LPUSER_INFO_1;
```

Поля этой структуры имеют следующее назначение.

В поле usri1_name хранится указатель на строку с именем пользователя. Длина строки не должна превышать UNLEN байтов, а сама строка должна иметь кодировку Unicode.

В поле usri1_password хранится указатель на строку с паролем. Длина строки не должна превышать PWLEN байтов, а сама строка должна иметь кодировку Unicode.

Поле usri1_password_age предназначено для хранения количества секунд, которые истекли с момента последнего изменения пароля.

Поле usri1_priv предназначено для хранения уровня привилегий, которые получает пользователь. В этом поле может храниться одно из следующих значений:

- USER_PRIV_GUEST — гость;
- USER_PRIV_USER — пользователь;
- USER_PRIV_ADMIN — администратор.

При вызове функции NetUserAdd это поле может содержать только значение USER_PRIV_USER.

В поле usri1_home_dir хранится указатель на строку с именем домашнего каталога пользователя. Строка должна иметь кодировку Unicode. Это поле может содержать значение NULL.

В поле usri1_comment хранится указатель на строку с комментариями о пользователе. Строка должна иметь кодировку Unicode. Это поле может содержать значение NULL.

Поле usri1_flags предназначено для хранения флагов, определяющих тип учетной записи и ее свойства. В этом поле может быть установлен один из следующих флагов:

- UF_NORMAL_ACCOUNT — учетная запись обычного пользователя;
- UF_TEMP_DUPLICATE_ACCOUNT — временная учетная запись, действительная только в домене;
- UF_WORKSTATION_TRUST_ACCOUNT — учетная запись компьютера в домене как рабочей станции, так и сервера;
- UF_SERVER_TRUST_ACCOUNT — учетная запись вторичного контроллера домена;
- UF_INTERDOMAIN_TRUST_ACCOUNT — учетная запись в доверительном домене.

Кроме того, в этом поле может быть установлена любая комбинация следующих флагов:

- UF_SCRIPT — исполнять скрипт (сценарий) при входе в систему;
- UF_ACCOUNTDISABLE — учетная запись недействительна;
- UF_HOMEDIR_REQUIRED — требуется домашний каталог;
- UF_PASSWD_NOTREQD — пароль не требуется;
- UF_PASSWD_CANT_CHANGE — пароль нельзя изменять;
- UF_LOCKOUT — учетная запись заблокирована;
- UF_DONT_EXPIRE_PASSWD — пароль не нужно изменять;

- UF_ENCRYPTED_TEXT_PASSWORD_ALLOWED — пароль сохранен в зашифрованном виде в Active Directory;
- UF_NOT_DELEGATED — учетная запись не может делегироваться другим пользователям;
- UF_SMARTCARD_REQUIRED — для входа в систему требуется смарт-карта;
- UF_USE_DES_KEY_ONLY — использовать только DES-стандарт шифрования;
- UF_DONT_REQUIRE_PREAUTH — не требуется Kerberos-аутентификация при входе в систему;
- UF_TRUSTED_FOR_DELEGATION — учетная запись может делегироваться;
- UF_PASSWORD_EXPIRED — время действия пароля истекло.

Отметим, что, начиная с флага UF_DONT_EXPIRE_PASSWD, все последующие флаги действительны только в версиях Windows 2000 и выше.

В поле usri1_script_path хранится указатель на строку с именем файла, содержащего скрипт, который выполняется перед входом пользователя в систему. Строка должна иметь кодировку Unicode. Это поле может содержать значение NULL.

В листинге 37.1 приведена программа, которая создает учетную запись пользователя, используя для этого функцию NetUserAdd.

Листинг 37.1. Создание учетной записи пользователя

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" ) // подключаем сетевую библиотеку

int main()
{
  wchar_t server_name[256] = L"\\\\";     // имя сервера
  wchar_t user_name[UNLEN];               // имя пользователя
  wchar_t user_password[PWLEN];           // пароль пользователя
  wchar_t user_comment[256];              // комментарии о пользователе
  USER_INFO_1    ui;                      // информация о пользователе
  NET_API_STATUS  ret_status;             // код возврата из функции

  printf("Input server name: ");
// формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));
```

```
  printf("Input user name: ");    // читаем имя пользователя
  wscanf(L"%s", user_name);       // читаем имя пользователя
  ui.usri1_name = user_name;      // устанавливаем имя пользователя

  printf("Input user password: ");
  wscanf(L"%s", user_password);       // читаем пароль пользователя
  ui.usri1_password = user_password; // устанавливаем пароль пользователя

  ui.usri1_priv = USER_PRIV_USER;  // обычный пользователь
  ui.usri1_home_dir = NULL;        // домашнего каталога нет

  printf("Input user comment: ");
  getwchar();                      // очищаем поток
  _getws(user_comment);            // читаем комментарии о пользователе
  ui.usri1_comment = user_comment; // устанавливаем комментарии

  ui.usri1_flags = UF_SCRIPT;     // исполнять скрипт при входе
                                  // пользователя в систему
  ui.usri1_script_path = NULL;    // пока файл со скриптом не определяем

  // добавляем пользователя
  ret_status = NetUserAdd(
    server_name,      // имя сервера
    1,                // уровень информации 1
    (LPBYTE)&ui,      // адрес информации о пользователе
    NULL);            // индексирования в структуре данных нет

  // проверяем на успешное завершение
  if (ret_status != NERR_Success)
  {
    printf("Net user add failed.\n");
    printf("Error code: %u\n", ret_status);

    return ret_status;
  }
  printf("The user is added.\n");

  return 0;
}
```

37.2. Получение информации о пользователе

Для получения информации о пользователе из его учетной записи используется функция NetUserGetInfo, которая имеет следующий прототип:

```
NET_API_STATUS  NetUserGetInfo(
  LPCWSTR  servername,   // имя сервера
  LPCWSTR  username,     // имя пользователя
  DWORD    level,        // уровень информации
  LPBYTE   *bufptr       // указатель на информацию
);
```

В случае успешного завершения функция NetUserGetInfo возвращает значение NERR_Success, а в случае неудачи возможны следующие коды завершения:

- ERROR_ACCESS_DENIED — пользователю отказано в доступе;
- NERR_InvalidComputer — неправильное имя компьютера;
- NERR_UserNotFound — пользователь не найден.

Параметры функции NetUserGetInfo имеют следующее назначение.

Параметр servername должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение NULL.

Параметр username должен указывать на строку с именем пользователя, информацию о котором возвращает функция. Эта строка должна иметь кодировку Unicode.

Параметр level указывает тип структуры, в которую будет записана информация о пользователе. Этот параметр может принимать одно из следующих значений:

- 1 — используется структура типа USER_INFO_1;
- 2 — используется структура типа USER_INFO_2;
- 3 — используется структура типа USER_INFO_3;
- 4 — используется структура типа USER_INFO_4;
- 10 — используется структура типа USER_INFO_10;
- 11 — используется структура типа USER_INFO_11;
- 20 — используется структура типа USER_INFO_20;
- 23 — используется структура типа USER_INFO_23.

В примере будет использована структура типа `USER_INFO_1`, тип которой описан в предыдущем разделе. Описание типов остальных структур можно найти в MSDN.

Параметр `bufptr` должен содержать адрес указателя на массив байтов. В этот указатель функция запишет адрес буфера с информацией о пользователе. Память под буфер выделяет система. Поэтому после прочтения информации о пользователе из буфера эта память должна быть освобождена посредством вызова функции `NetApiBufferFree`, которая рассмотрена далее.

Для освобождения памяти, которая резервируется функциями из библиотеки netapi32.lib, используется функция `NetApiBufferFree`, которая имеет следующий прототип:

```
NET_API_STATUS  NetApiBufferFree(
  LPVOID  buffer       // указатель на буфер
);
```

В случае успешного завершения функция `NetApiBufferFree` возвращает значение `NERR_Success`, а в случае неудачи код ошибки. Единственный параметр этой функции должен указывать на буфер, память для которого была предварительно распределена одной из сетевых функций.

В листинге 37.2 приведена программа, которая получает информацию о пользователе при помощи функции `NetUserGetInfo`.

Листинг 37.2. Получение информации о пользователе

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )   // подключаем сетевую библиотеку

int main()
{
  wchar_t server_name[256] = L"\\\\";  // имя сервера
  wchar_t user_name[UNLEN];            // имя пользователя
  USER_INFO_1  *ui;                    // информация о пользователе
  NET_API_STATUS  ret_status;          // код возврата из функции

  printf("Input server name: ");       // вводим имя сервера
  // формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));
```

```
  printf("Input user name: ");
  wscanf(L"%s", user_name);            // читаем имя пользователя

  // получаем информацию о пользователе
  ret_status = NetUserGetInfo(
    server_name,      // имя сервера
    user_name,        // имя пользователя
    1,                // уровень информации 1
    (LPBYTE*)&ui);    // адрес информации о пользователе

  // проверяем на успешное завершение
  if (ret_status != NERR_Success)
  {
    printf("Net user get info failed.\n");

    return ret_status;
  }

  wprintf(L"User name: %s\n", ui->usri1_name);
  wprintf(L"User commemnt: %s\n", ui->usri1_comment);

  NetApiBufferFree(ui);     // освобождаем буфер

  return 0;
}
```

37.3. Перечисление пользователей

Для того чтобы перечислить учетные записи всех пользователей, зарегистрированных на сервере, используется функция `NetUserEnum`, которая имеет следующий прототип:

```
  NET_API_STATUS  NetUserEnum(
    LPCWSTR  servername,      // имя сервера
    DWORD    level,           // уровень информации
    DWORD    filter,          // фильтр на учетные записи
    LPBYTE   *bufptr,         // указатель на буфер с информацией
    DWORD    prefmaxlen,      // длина буфера с информацией
    LPDWORD  entriesread,     // количество прочитанных структур
```

```
    LPDWORD  totalentries,   // общее количество структур
    LPDWORD  resume_handle   // дескриптор для продолжения чтения
  );
```

В случае успешного завершения функция `NetUserEnum` возвращает значение `NERR_Success`, а в случае неудачи возможны следующие коды завершения:

- `ERROR_ACCESS_DENIED` — пользователю отказано в доступе;
- `NERR_InvalidComputer` — неправильное имя компьютера;
- `ERROR_MORE_DATA` — не все данные прочитаны.

Параметры функции `NetUserEnum` имеют следующее назначение.

Параметр `servername` должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение `NULL`.

Параметр `level` указывает тип структуры, в которую будет записана информация о пользователе. Этот параметр может принимать одно из следующих значений:

- 1 — используется структура типа `USER_INFO_1`;
- 2 — используется структура типа `USER_INFO_2`;
- 3 — используется структура типа `USER_INFO_3`;
- 4 — используется структура типа `USER_INFO_4`;
- 10 — используется структура типа `USER_INFO_10`;
- 11 — используется структура типа `USER_INFO_11`;
- 20 — используется структура типа `USER_INFO_20`;
- 23 — используется структура типа `USER_INFO_23`.

Отметим, что если установлен один из первых четырех уровней, то функция не возвращает пароль пользователя. В примере будет использована структура типа `USER_INFO_1`, тип которой описан в *разд. 37.1*. Описание типов остальных структур можно найти в MSDN.

Параметр `filter` задает типы учетных записей, которые должны быть перечислены. Этот параметр может принимать одно из следующих значений:

- `FILTER_TEMP_DUPLICATE_ACCOUNT` — перечисляются локальные учетные записи;
- `FILTER_NORMAL_ACCOUNT` — перечисляются глобальные учетные записи;
- `FILTER_INTERDOMAIN_TRUST_ACCOUNT` — перечисляются доверительные учетные записи домена;
- `FILTER_WORKSTATION_TRUST_ACCOUNT` — перечисляются учетные записи рабочих станций и серверов домена;

- FILTER_SERVER_TRUST_ACCOUNT — перечисляются учетные записи контроллеров домена.

Параметр bufptr должен содержать адрес указателя, в который система запишет адрес буфера с информацией об учетных записях. Память под буфер распределяется системой. Поэтому после прочтения информации из буфера эту память нужно освободить при помощи вызова функции NetApiBufferFree, которая была описана в предыдущем разделе. Эту память нужно освободить также и в том случае, если функция вернула код ошибки ERROR_MORE_DATA.

Параметр prefmaxlen должен содержать предпочтительную длину буфера в байтах. Если памяти для данных не хватит, то функция вернет код ошибки ERROR_MORE_DATA. В этом параметре можно установить значение MAX_PREFERRED_LENGTH. В этом случае система сама определит необходимую длину буфера для информации об учетных записях.

Параметр entriesread должен указывать на двойное слово, в которое функция запишет количество учетных записей, информация о которых записана в буфер данных.

Параметр totalentries должен указывать на переменную типа DWORD, в которую функция запишет общее количество учетных записей, удовлетворяющих условию отбора.

Параметр resume_handle должен указывать на переменную типа DWORD, в которую функция запишет дескриптор для продолжения перечисления учетных записей. При первом вызове функции это двойное слово должно содержать значение 0. Если продолжение чтения информации об учетных записях не предполагается, то в этом параметре нужно установить значение NULL.

В листинге 37.3 приведена программа, которая перечисляет учетные записи пользователей, используя функцию NetUserEnum.

Листинг 37.3. Перечисление учетных записей пользователей

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" ) // подключаем сетевую библиотеку

int main()
{
  wchar_t server_name[256] = L"\\\\";  // имя сервера
  USER_INFO_0  *ui;         // информация о пользователе
```

```
  DWORD  entries_read;     // количество прочитанных пользователей
  DWORD  total_entries;    // общее количество пользователей

  NET_API_STATUS  ret_status;  // код возврата из функции

  printf("Input server name: ");   // читаем имя сервера
  // формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));

  // получаем информацию о пользователях
  ret_status = NetUserEnum(
    server_name,        // имя сервера
    0,                  // узнаем только имена пользователей
    FILTER_NORMAL_ACCOUNT,   // перечисляем пользователей,
                             // зарегистрированных на компьютере
    (LPBYTE*)&ui,       // адрес информации о пользователях
    MAX_PREFERRED_LENGTH,    // перечисляем всех пользователей
    &entries_read,      // количество прочитанных пользователей
    &total_entries,     // общее количество пользователей
    NULL);              // индексации нет

  // проверяем на успешное завершение
  if (ret_status != NERR_Success)
  {
    printf("Net user get info failed.\n");
    printf("Net API Status: %d\n", ret_status);
    NetApiBufferFree(ui);    // освобождаем буфер

    return ret_status;
  }

  for (DWORD i = 0; i < entries_read; ++i)
    wprintf(L"User name: %s\n", ui[i].usri0_name);

  NetApiBufferFree(ui);      // освобождаем буфер

  return 0;
}
```

37.4. Перечисление групп, которым принадлежит пользователь

Для перечисления глобальных групп, которым принадлежит пользователь, используется функция NetUserGetGroups, которая имеет следующий прототип:

```
NET_API_STATUS  NetUserGetGroups(
  LPCWSTR  servername,   // имя сервера
  LPCWSTR  username,     // имя пользователя
  DWORD    level,        // уровень информации
  LPBYTE   *bufptr,      // адрес указателя на буфер с информацией
  DWORD    prefmaxlen,   // длина буфера с информацией
  LPDWORD  entriesread,  // количество прочитанных структур
  LPDWORD  totalentries  // общее количество структур
);
```

В случае успешного завершения функция NetUserGetGroups возвращает значение NERR_Success, а в случае неудачи возможны следующие коды завершения:

- ERROR_ACCESS_DENIED — пользователю отказано в доступе;
- ERROR_MORE_DATA — не все данные прочитаны;
- NERR_InvalidComputer — неправильное имя компьютера;
- NERR_UserNotFound — имя пользователя не найдено.

Параметры функции NetUserGetGroups имеют следующее назначение.

Параметр servername должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение NULL.

Параметр username должен указывать на строку с именем пользователя, которое проверяется на принадлежность глобальным группам для их перечисления.

Параметр level указывает тип структуры, в которую будет записана информация о глобальных группах. Этот параметр может принимать только значение 0, которое обозначает, что для записи информации о локальной группе будет использоваться структура GROUP_USERS_INFO_0, которая имеет следующий тип:

```
typedef struct  _GROUP_USERS_INFO_0 {
  LPWSTR  grui0_name;    // имя пользователя
}  GROUP_USERS_INFO_0, *PGROUP_USERS_INFO_0, *LPGROUP_USERS_INFO_0;
```

Параметр `bufptr` должен содержать адрес указателя, в который система запишет адрес буфера с информацией о глобальных группах. Память под буфер распределяется системой. Поэтому после прочтения информации из буфера эту память нужно освободить при помощи вызова функции `NetApiBufferFree`, которая была описана в *разд. 37.2*. Эту память нужно освободить также и в том случае, если функция вернула код ошибки `ERROR_MORE_DATA`.

Параметр `prefmaxlen` должен содержать предпочтительную длину буфера в байтах. Если памяти для данных не хватит, то функция вернет код ошибки `ERROR_MORE_DATA`. В этом параметре можно установить значение `MAX_PREFERRED_LENGTH`. В этом случае система сама определит необходимую длину буфера для информации об учетных записях.

Параметр `entriesread` должен указывать на переменную типа `DWORD`, в которую функция запишет количество глобальных групп, имена которых записаны в буфер данных.

Параметр `totalentries` должен указывать на переменную типа `DWORD`, в которую функция запишет общее количество глобальных групп, которым принадлежит пользователь.

В листинге 37.4 приведена программа, которая перечисляет глобальные группы, которым принадлежит пользователь, используя функцию `NetUserGetGroups`.

Листинг 37.4. Перечисление глобальных групп, которым принадлежит пользователь

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" ) // подключаем сетевую библиотеку

int main()
{
  wchar_t server_name[256] = L"\\\\";  // имя сервера
  wchar_t user_name[UNLEN];     // имя пользователя
  GROUP_USERS_INFO_0  *ui;      // информация о группах
  DWORD  entries_read;          // количество прочитанных групп
  DWORD  total_entries;         // общее количество групп

  NET_API_STATUS  ret_status;       // код возврата из функции
```

```
  printf("Input server name: ");
  // формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));

  printf("Input user name: ");
  wscanf(L"%s", user_name);        // читаем имя пользователя

  // получаем информацию о группах, в которые входит пользователь
  ret_status = NetUserGetGroups(
    server_name,       // имя сервера
    user_name,         // имя пользователя
    0,                 // узнаем имена групп
    (LPBYTE*)&ui,      // адрес информации о группах
    MAX_PREFERRED_LENGTH,    // перечисляем все группы
    &entries_read,     // количество прочитанных групп
    &total_entries);   // общеее количество групп

  // проверяем на успешное завершение
  if (ret_status != NERR_Success)
  {
    printf("Net user get groups failed.\n");
    printf("Net API Status: %d\n", ret_status);
    NetApiBufferFree(ui);      // освобождаем буфер

    return ret_status;
  }

  for (DWORD i = 0; i < entries_read; ++i)
    wprintf(L"Global group name: %s\n", ui[i].grui0_name);

  NetApiBufferFree(ui);      // освобождаем буфер

  return 0;
}
```

Для перечисления локальных групп, которым принадлежит пользователь, используется функция `NetUserGetLocalGroups`, которая имеет следующий прототип:

```
  NET_API_STATUS  NetUserGetLocalGroups (
    LPCWSTR  servername,   // имя сервера
```

```
    LPCWSTR  username,     // имя пользователя
    DWORD    level,        // уровень информации
    DWORD    flags,        // флаг косвенного поиска
    LPBYTE   *bufptr,      // адрес указателя на буфер с информацией
    DWORD    prefmaxlen,   // длина буфера с информацией
    LPDWORD  entriesread,  // количество прочитанных структур
    LPDWORD  totalentries  // общее количество структур
  );
```

В случае успешного завершения функция `NetUserGetLocalGroups` возвращает значение `NERR_Success`, а в случае неудачи возможны такие же коды завершения, как и в случае функции `NetUserGetGroups`. Параметры функции `NetUserGetLocalGroups` имеют такое же назначение, как и параметры функции `NetUserGetGroups`. Существуют только небольшие дополнения, о которых и сказано далее.

Если имя пользователя в строке, на которую указывает параметр `username`, задано в виде `имя_домена\имя_пользователя`, то имя пользователя ищется в домене. Если же имя пользователя задано только как `имя_пользователя`, то оно ищется на сервере, имя которого задано первым параметром.

Параметр `flags` может принимать значение `LG_INCLUDE_INDIRECT`. Если это значение установлено, то перечисляются как локальные группы, которым пользователь принадлежит непосредственно, так и локальные группы, которым пользователь принадлежит косвенно. То есть через членство в глобальных группах, которые принадлежат локальной группе. В противном случае перечисляются только локальные группы, которым пользователь принадлежит только непосредственно.

В листинге 37.5 приведена программа, которая перечисляет локальные группы, которым принадлежит пользователь, используя для этого функцию `NetUserGetLocalGroups`.

Листинг 37.5. Перечисление локальных групп, которым принадлежит пользователь

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )   // подключаем сетевую библиотеку

int main()
{
```

```
  wchar_t server_name[256] = L"\\\\";  // имя сервера
  wchar_t user_name[UNLEN];            // имя пользователя
  LOCALGROUP_USERS_INFO_0  *ui;        // информация о группах
  DWORD  entries_read;     // количество прочитанных групп
  DWORD  total_entries;    // общее количество групп

  NET_API_STATUS  ret_status;      // код возврата из функции

  printf("Input server name: ");
// формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));

  printf("Input user name: ");
  wscanf(L"%s", user_name);        // читаем имя пользователя

  // получаем локальные группы, в которые входит пользователь
  ret_status = NetUserGetLocalGroups(
    server_name,              // имя сервера
    user_name,                // имя пользователя
    0,                        // узнаем только имена групп
    LG_INCLUDE_INDIRECT,      // включая косвенную принадлежность
    (LPBYTE*)&ui,             // адрес информации о группе
    MAX_PREFERRED_LENGTH,     // перечисляем все группы
    &entries_read,            // количество прочитанных групп
    &total_entries);          // общее количество групп

  // проверяем на успешное завершение
  if (ret_status != NERR_Success)
  {
    printf("Net user get groups failed.\n");
    printf("Net API Status: %d\n", ret_status);
    NetApiBufferFree(ui);      // освобождаем буфер

    return ret_status;
  }

  for (DWORD i = 0; i < entries_read; ++i)
```

```
    wprintf(L"Local group name: %s\n", ui[i].lgrui0_name);

  NetApiBufferFree(ui);      // освобождаем буфер

  return 0;
}
```

37.5. Изменение учетной записи пользователя

Для изменения содержимого учетной записи пользователя используется функция NetUserSetInfo, которая имеет следующий прототип:

```
NET_API_STATUS  NetUserSetInfo(
  LPCWSTR  servername,    // имя сервера
  LPCWSTR  username,      // имя пользователя
  DWORD    level,         // уровень информации
  LPBYTE   buf,           // буфер с информацией об учетной записи
  LPDWORD  parm_err       // индекс поля с неправильными данными
);
```

Отметим, что успешно эту функцию могут выполнить только пользователи, которые являются администраторами или операторами учетных записей.

В случае успешного завершения функция NetUserSetInfo возвращает значение NERR_Success, а в случае неудачи возможны следующие коды завершения:

- ERROR_ACCESS_DENIED — пользователю отказано в доступе;
- ERROR_INVALID_PARAMETER — неправильный параметр;
- NERR_InvalidComputer — неправильное имя компьютера;
- NERR_NotPrimary — не первичный контроллер домена;
- NERR_SpeGroupOp — не разрешается выполнять над членами специальной группы;
- NERR_LastAdmin — не разрешается выполнять над учетной записью администратора;
- NERR_BadPassword — неправильный пароль;
- NERR_PasswordTooShort — пароль короче, чем требуется;
- NERR_UserNotFound — имя пользователя не найдено.

Параметры этой функции имеют следующее назначение.

Параметр `servername` должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение `NULL`.

Параметр `username` должен указывать на строку с именем учетной записи, содержимое которой модифицируется. Эта строка должна иметь кодировку Unicode.

Параметр `level` указывает тип структуры, которая содержит новую информацию об учетной записи пользователя. Этот параметр может принимать одно из следующих значений:

- 0 — используется структура типа `USER_INFO_0`;
- 1 — используется структура типа `USER_INFO_1`;
- 2 — используется структура типа `USER_INFO_2`;
- 3 — используется структура типа `USER_INFO_3`;
- 4 — используется структура типа `USER_INFO_4`;
- 21 — используется структура типа `USER_INFO_21`;
- 22 — используется структура типа `USER_INFO_22`;
- 1003 — используется структура типа `USER_INFO_1003`;
- 1005 — используется структура типа `USER_INFO_1005`;
- 1006 — используется структура типа `USER_INFO_1006`;
- 1007 — используется структура типа `USER_INFO_1007`;
- 1008 — используется структура типа `USER_INFO_1008`;
- 1009 — используется структура типа `USER_INFO_1009`;
- 1010 — используется структура типа `USER_INFO_1010`;
- 1011 — используется структура типа `USER_INFO_1011`;
- 1012 — используется структура типа `USER_INFO_1012`;
- 1014 — используется структура типа `USER_INFO_1014`;
- 1017 — используется структура типа `USER_INFO_1017`;
- 1020 — используется структура типа `USER_INFO_1020`;
- 1024 — используется структура типа `USER_INFO_1024`;
- 1051 — используется структура типа `USER_INFO_1051`;
- 1052 используется структура типа `USER_INFO_1052`;
- 1053 используется структура типа `USER_INFO_1053`.

В примере будет использована структура `USER_INFO_0`, которая имеет следующий тип:

```
typedef struct _USER_INFO_0 {
  LPWSTR  usri0_name;      // имя пользователя
}USER_INFO_0, *PUSER_INFO_0, *LPUSER_INFO_0;
```

Эта структура используется в том случае, если при помощи функции `NetUserSetInfo` изменяется имя пользователя. Остальные структуры имеют также конкретное назначение. Например, структура типа `USER_INFO_1008` используется для установки флагов, которые управляют свойствами учетной записи пользователя. Описание типов остальных структур можно найти в MSDN.

Параметр `buf` должен указывать на структуру, которая содержит информацию об учетной записи пользователя. Тип этой структуры задается параметром `level`.

Параметр `parm_err` должен указывать на переменную типа `DWORD`, в которую функция `NetUserSetInfo` поместит индекс поля в структуре с информацией о пользователе, содержимое которого вызвало неудачное завершение функции. В этом случае функция `NetUserSetInfo` вернет значение `ERROR_INVALID_PARAMETER`.

В листинге 37.6 приведена программа, которая изменяет имя пользователя, используя функцию `NetUserSetInfo`.

Листинг 37.6. Изменение имени пользователя

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )   // подключаем сетевую библиотеку

int main()
{
  wchar_t server_name[256] = L"\\\\";  // имя сервера
  wchar_t old_name[UNLEN];          // старое имя пользователя
  wchar_t new_name[UNLEN];          // новое имя пользователя
  USER_INFO_0 ui;                   // информация о пользователе
  NET_API_STATUS  ret_status;       // код возврата из функции

  printf("Input server name: ");
```

```
// формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));

  printf("Input old user name: ");
  wscanf(L"%s", old_name);       // читаем старое имя пользователя

  printf("Input new user name: ");
  wscanf(L"%s", new_name);       // читаем новое имя пользователя
  ui.usri0_name = new_name;      // устанавливаем новое имя пользователя

  // устанавливаем информацию о пользователе
  ret_status = NetUserSetInfo(
    server_name,     // имя сервера
    old_name,        // имя пользователя
    0,               // изменяем имя пользователя
    (LPBYTE)&ui,     // адрес информации о пользователе
    NULL);           // нет индексации

  // проверяем на успешное завершение
  if (ret_status != NERR_Success)
  {
    printf("Net user set info failed.\n");
    printf("Net API Status: %d\n", ret_status);

    return ret_status;
  }

  printf("User name is changed.\n");

  USER_INFO_1  *uix;         // информация о пользователе
  // получаем информацию о пользователе
  ret_status = NetUserGetInfo(
    server_name,     // имя сервера
    new_name,        // имя пользователя
    1,               // уровень информации 1
    (LPBYTE*)&uix);  // адрес информации о пользователе

  // проверяем на успешное завершение
  if (ret_status != NERR_Success)
```

```
  {
    printf("Net user get info failed.\n");

    return ret_status;
  }

  wprintf(L"User name: %s\n", uix->usri1_name);
  wprintf(L"User commemnt: %s\n", uix->usri1_comment);

  return 0;
}
```

37.6. Изменение пароля пользователя

Для изменения пароля пользователя используется функция `NetUserChangePassword`, которая имеет следующий прототип:

```
NET_API_STATUS  NetUserChangePassword(
  LPCWSTR  domainname,   // имя домена
  LPCWSTR  username,     // имя пользователя
  LPCWSTR  oldpassword,  // старый пароль
  LPCWSTR  newpassword   // новый пароль
);
```

Отметим, что сервер может быть сконфигурирован таким образом, что успешно эту функцию могут выполнить только пользователи, которые являются администраторами или операторами учетных записей, или пользователь, который изменяет свой пароль.

В случае успешного завершения функция `NetUserChangePassword` возвращает значение `NERR_Success`, а в случае неудачи возможны следующие коды завершения:

- `ERROR_ACCESS_DENIED` — пользователю отказано в доступе;
- `ERROR_INVALID_PASSWORD` — неправильный пароль;
- `NERR_InvalidComputer` — неправильное имя компьютера;
- `NERR_NotPrimary` — не первичный контроллер домена;
- `NERR_PasswordTooShort` — пароль короче, чем требуется;
- `NERR_UserNotFound` — имя пользователя не найдено.

Параметры этой функции имеют следующее назначение.

Параметр `domainname` должен указывать на строку с именем сервера или домена, в котором зарегистрирован пользователь. Эта строка должна иметь

кодировку Unicode и начинаться с символов \\. Если этот параметр равен NULL, то имя пользователя ищется в домене, в котором зарегистрирована учетная запись, от имени которой вызвана функция.

Параметр username должен указывать на строку с именем учетной записи, пароль которой изменяется. Эта строка должна иметь кодировку Unicode. Если значение этого параметра равно NULL, то используется имя пользователя, указанное при входе в систему.

Параметр oldpassword должен указывать на строку со старым паролем пользователя. Эта строка должна иметь кодировку Unicode.

Параметр newpassword должен указывать на строку с новым паролем пользователя. Эта строка должна иметь кодировку Unicode

В листинге 36.7 приведена программа, которая изменяет пароль пользователя, используя функцию NetUserChangePassword.

Листинг 37.7. Изменение пароля пользователя.

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )   // подключаем сетевую библиотеку

int main()
{
  wchar_t domain_name[256] = L"\\\\";  // имя сервера
  wchar_t user_name[UNLEN];            // имя пользователя
  wchar_t old_password[PWLEN];         // старый пароль пользователя
  wchar_t new_password[PWLEN];         // новый пароль пользователя

  NET_API_STATUS  ret_status;  // код возврата из функции

  printf("Input server or domain name: ");
  wscanf(L"%s", domain_name);    // читаем имя домена

  printf("Input user name: ");
  wscanf(L"%s", user_name);      // читаем имя пользователя

  printf("Input old password: ");
  wscanf(L"%s", old_password);   // читаем имя пользователя
```

```
  printf("Input new password: ");
  wscanf(L"%s", new_password);   // читаем имя пользователя

  // изменяем пароль пользователя
  ret_status = NetUserChangePassword(
    domain_name,     // имя домена
    user_name,       // имя пользователя
    old_password,    // старый пароль
    new_password);   // новый пароль

  // проверяем на успешное завершение
  if (ret_status != NERR_Success)
  {
    printf("Net user change password failed.\n");
    printf("Net API Status: %d\n", ret_status);

    return ret_status;
  }

  printf("The password was changed.\n");

  return 0;
}
```

37.7. Удаление учетной записи пользователя

Для удаления учетной записи пользователя из базы данных менеджера учетных записей используется функция `NetUserDel`, которая имеет следующий прототип:

```
  NET_API_STATUS  NetUserDel(
    LPCWSTR  servername,   // имя сервера
    LPCWSTR  username      // имя пользователя
  );
```

Отметим, что успешно эту функцию могут выполнить только пользователи, которые являются администраторами или операторами учетных записей. А учетную запись администратора может удалить только другой администратор.

В случае успешного завершения функция NetUserDel возвращает значение NERR_Success, а в случае неудачи возможны следующие коды завершения:

- ERROR_ACCESS_DENIED — пользователю отказано в доступе;
- NERR_InvalidComputer — неправильное имя компьютера;
- NERR_NotPrimary — не первичный контроллер домена;
- NERR_UserNotFound — имя пользователя не найдено.

Параметры этой функции имеют такое же назначение, как в других функциях управления пользователями.

Параметр servername должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение NULL.

Параметр username должен указывать на строку с именем учетной записи, которая удаляется из базы данных менеджера учетных записей. Эта строка должна иметь кодировку Unicode.

В листинге 37.8 приведена программа, которая удаляет учетную запись пользователя, используя функцию NetUserDel.

Листинг 37.8. Удаление учетной записи пользователя

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )   // подключаем сетевую библиотеку

int main()
{
  wchar_t server_name[256] = L"\\\\";  // имя сервера
  wchar_t user_name[UNLEN];            // имя пользователя

  NET_API_STATUS  ret_status;  // код возврата из функции

  printf("Input server name: ");
  wscanf(L"%s", server_name);    // читаем имя домена

  printf("Input user name: ");
```

```
  wscanf(L"%s", user_name);      // читаем имя пользователя

  // удалаем пользователя
  ret_status = NetUserDel(
    server_name,     // имя домена
    user_name);      // имя пользователя

  // проверяем на успешное завершение
  if (ret_status != NERR_Success)
  {
    printf("Net user del failed.\n");
    printf("Net API Status: %d\n", ret_status);

    return ret_status;
  }

  printf("The user is deleted.\n");

  return 0;
}
```

Глава 38

Управление группами

В этой главе будут рассмотрены только функции, предназначенные для работы с локальными группами. Функции, предназначенные для работы с глобальными группами, рассматриваться не будут. Для этого существуют две причины. Во-первых, сигнатура и правила использования функций для работы с глобальными группами очень похожи соответственно на сигнатуру и правила использования функций для работы с локальными группами. Поэтому после изучения этого раздела программист сможет самостоятельно разобраться с функциями, предназначенными для управления глобальными группами. Во-вторых, этот материал предназначен для начинающих программистов. Поэтому маловероятно, что такие программисты будут экспериментировать с глобальными группами локальной сети. А для этого достаточно научиться управлять только локальными группами.

38.1. Создание локальной группы

Для создания учетной записи локальной группы в базе данных менеджера учетных записей используется функция `NetLocalGroupAdd`, которая имеет следующий прототип:

```
NET_API_STATUS  NetLocalGroupAdd(
  LPCWSTR  servername,   // имя сервера
  DWORD    level,        // уровень информации
  LPBYTE   buf,          // буфер с информацией
  LPDWORD  parm_err      // индексирование ошибки
);
```

Отметим, что успешно эту функцию могут выполнить только пользователи, которые являются администраторами или операторами учетных записей.

В случае успешного завершения функция NetLocalGroupAdd возвращает значение NERR_Success, а в случае неудачи возможны следующие коды завершения:

- ERROR_ACCESS_DENIED — пользователю отказано в доступе;
- NERR_InvalidComputer — неправильное имя компьютера;
- NERR_NotPrimary — операция может выполняться только на первичном контроллере домена;
- NERR_GroupExists — группа уже существует;
- ERROR_ALIAS_EXISTS — группа уже существует.

Параметры функции NetLocalGroupAdd имеют следующее назначение.

Параметр servername должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение NULL.

Параметр level указывает тип структуры, которая содержит информацию о локальной группе. Этот параметр может принимать одно из следующих значений:

- 0 — используется структура типа LOCALGROUP_INFO_0;
- 1 — используется структура типа LOCALGROUP_INFO_1.

В примере будет использована структура типа LOCALGROUP_INFO_1, описание которой и будет приведено после описания функции. Описание типа структуры LOCALGROUP_INFO_0 может быть найдено в MSDN.

Параметр buf должен указывать на буфер с информацией о локальной группе. Информация должна храниться в структуре, тип которой задан параметром level.

В переменную, на которую указывает параметр parm_err, функция NetLocalGroupAdd помещает индекс первого поля в структуре с информацией о локальной группе, которое содержит неправильные данные и, как следствие, вызывает завершение функции NetLocalGroupAdd с ошибкой. В этом параметре может быть установлено значение NULL. В этом случае индекс поля с неправильными данными не возвращается.

Так как в примере будет использована структура типа LOCALGROUP_INFO_1, то приведем тип и опишем назначение полей этой структуры. Структура LOCALGROUP_INFO_1 имеет следующий тип:

```
typedef struct _LOCALGROUP_INFO_1 {
  LPWSTR  lgrpi1_name;       // имя группы
  LPWSTR  lgrpi1_comment;    // комментарии
}LOCALGROUP_INFO_1, *PLOCALGROUP_INFO_1, *LPLOCALGROUP_INFO_1;
```

Поля этой структуры имеют следующее назначение.

В поле `lgrpi1_name` хранится указатель на строку с именем локальной группы. Длина строки не должна превышать `GNLEN` байт, а сама строка должна иметь кодировку Unicode.

В поле `lgrpi1_comment` хранится указатель на строку с комментариями относительно локальной группы. Длина строки не должна превышать `MAXCOMMENTSZ` байтов, а сама строка должна иметь кодировку Unicode.

В листинге 38.1 приведена программа, которая создает локальную группу, используя для этого функцию `NetLocalGroupAdd`.

Листинг 38.1. Создание локальной группы

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )   // подключаем сетевую библиотеку

int main()
{
  wchar_t server_name[256] = L"\\\\";  // имя сервера
  wchar_t  group_name[GNLEN];          // имя локальной группы
  wchar_t comment[MAXCOMMENTSZ];       // комментарий о группе
  LOCALGROUP_INFO_1  group_info;       // информация о группе
  NET_API_STATUS ret_status;           // код возврата из функции

  printf("Input server name: ");
// формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));

  printf("Input a name for the local group: ");
  wscanf(L"%s", group_name);           // вводим имя группы
  group_info.lgrpi1_name = group_name;   // устанавливаем адрес имени
                                         // в структуру


  printf("Input group comments: ");
  getwchar();              // очищаем поток
  _getws(comment);         // читаем комментарий о локальной группе
```

```
  group_info.lgrpi1_comment = comment;   // устанавливаем комментарий

  // создаем локальную группу
  ret_status = NetLocalGroupAdd(
    server_name,           // имя сервера
    1,                     // уровень входных данных
    (LPBYTE)&group_info,   // имя группы и комментарии
    NULL);                 // индексацию данных не используем
  if (ret_status != NERR_Success)
  {
    printf("Net local group add failed.\n");

    return ret_status;
  }

  printf("The group is created.\n");

  return 0;
}
```

38.2. Получение информации о локальной группе

Для получения информации о локальной группе используется функция `NetLocalGroupGetInfo`, которая имеет следующий прототип:

```
  NET_API_STATUS  NetLocalGroupGetInfo(
    LPCWSTR  servername,   // имя сервера
    LPCWSTR  groupname,    // имя группы
    DWORD    level,        // уровень информации
    LPBYTE   *bufptr       // буфер для информации
  );
```

Отметим, что успешно эту функцию могут выполнить только пользователи, которые являются администраторами или операторами учетных записей.

В случае успешного завершения функция `NetLocalGroupGetInfo` возвращает значение `NERR_Success`, а в случае неудачи возможны следующие коды завершения:

- `ERROR_ACCESS_DENIED` — пользователю отказано в доступе;
- `NERR_InvalidComputer` — неправильное имя компьютера;
- `ERROR_NO_SUCH_ALIAS` — группа с таким именем не существует.

Параметры функции NetLocalGroupGetInfo имеют следующее назначение.

Параметр servername должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение NULL.

Параметр groupname должен указывать на строку с именем локальной группы, о которой получают информацию. Эта строка должна иметь кодировку Unicode.

Параметр level указывает тип структуры, которая содержит информацию о локальной группе. Этот параметр может принимать только одно значение — 1, указывающее на то, что используется структура типа LOCALGROUP_INFO_1;

В этом случае будет использована структура LOCALGROUP_INFO_1, тип которой был описан в предыдущем разделе.

Параметр bufptr должен содержать адрес указателя, в который система запишет адрес буфера с информацией о локальных группах. Память под буфер распределяется системой. Поэтому после прочтения информации из буфера эту память нужно освободить при помощи вызова функции NetApiBufferFree, которая была описана в *разд. 37.2*.

В листинге 38.2 приведена программа, которая получает информацию о локальной группе, используя для этого функцию NetLocalGroupGetInfo.

Листинг 38.2. Получение информации о локальной группе

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )   // подключаем сетевую библиотеку

int main()
{
  wchar_t server_name[256] = L"\\\\";    // имя сервера
  wchar_t  group_name[GNLEN];            // имя локальной группы
  LOCALGROUP_INFO_1  *buf_ptr = NULL;    // адрес буфера для данных
  NET_API_STATUS    ret_status;          // код возврата из функции

  printf("Input server name: ");
// формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));
```

```
  printf("Input name for the local group: ");
  wscanf(L"%s", group_name);          // вводим имя группы

  // получаем информацию о локальной группе
  ret_status = NetLocalGroupGetInfo(
    server_name,            // имя сервера
    group_name,             // имя группы
    1,                      // получаем имя группы и комментарий
    (LPBYTE*)&buf_ptr);     // указатель на адрес буфера для данных
  // проверяем завершение функции
  if (ret_status != NERR_Success)
  {
    printf("Net local group get info failed.\n");
    NetApiBufferFree(buf_ptr);  // освобождаем буфер для данных

    return ret_status;
  }

  // выводим на консоль имя локальной группы и комментарий
  wprintf(L"Local group name: %s\n", buf_ptr->lgrpi1_name);
  wprintf(L"Comment: %s\n", buf_ptr->lgrpi1_comment);

  NetApiBufferFree(buf_ptr);     // освобождаем буфер для данных

  return 0;
}
```

38.3. Перечисление локальных групп

Для перечисления локальных групп, учетные записи которых зарегистрированны на заданном сервере, используется функция `NetLocalGroupEnum`, которая имеет следующий прототип:

```
  NET_API_STATUS  NetLocalGroupEnum(
    LPCWSTR  servername,   // имя сервера
    DWORD    level,        // уровень информации
    LPBYTE   *bufptr,      // буфер для информации
    DWORD    prefmaxlen,   // длина буфера
    LPDWORD  entriesread,  // количество прочитанных структур
    LPDWORD  totalentries, // общее количество групп
```

```
    LPDWORD  resumehandle  // дескриптор для продолжения перечисления
  );
```

В случае успешного завершения функция `NetLocalGroupEnum` возвращает значение `NERR_Success`, а в случае неудачи возможны следующие коды завершения:

- `ERROR_ACCESS_DENIED` — пользователю отказано в доступе;
- `ERROR_MORE_DATA` — не все данные прочитаны;
- `NERR_InvalidComputer` — неправильное имя компьютера;
- `NERR_BufTooSmall` — буфер очень мал.

Опишем параметры функции `NetLocalGroupEnum`.

Параметр `servername` должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение `NULL`.

Параметр `level` указывает тип структуры, которая содержит информацию о локальной группе. Этот параметр может принимать одно из следующих значений:

- 0 — используется структура типа `LOCALGROUP_INFO_0`;
- 1 — используется структура типа `LOCALGROUP_INFO_1`.

В примере будет использована структура типа `LOCALGROUP_INFO_1`, тип которой приведен в *разд. 38.1*. Описание типа структуры `LOCALGROUP_INFO_0` может быть найдено в MSDN.

Параметр `bufptr` должен содержать адрес указателя, в который система запишет адрес буфера с информацией о локальных группах. Память под буфер распределяется системой. Поэтому после прочтения информации из буфера эту память нужно освободить при помощи вызова функции `NetApiBufferFree`, которая была описана в *разд. 37.2*. Эту память нужно освободить также и в том случае, если функция вернула код ошибки `ERROR_MORE_DATA`.

Параметр `prefmaxlen` должен содержать предпочтительную длину буфера в байтах. Если памяти для данных не хватит, то функция вернет код ошибки `ERROR_MORE_DATA`. В этом параметре можно установить значение `MAX_PREFERRED_LENGTH`. Тогда система сама определит необходимую длину буфера для информации об учетных записях.

Параметр `entriesread` должен указывать на переменную типа `DWORD`, в которую функция запишет количество локальных групп, информация о которых записана в буфер данных.

Параметр `totalentries` должен указывать на переменную типа `DWORD`, в которую функция запишет общее количество локальных групп, зарегистрированных на сервере.

Параметр `resume_handle` должен указывать на переменную типа `DWORD`, в которую функция запишет дескриптор для продолжения перечисления локальных групп. При первом вызове функции этот параметр должен быть равен 0. Если продолжение чтения информации об учетных записях не предполагается, то в этом параметре нужно установить значение `NULL`.

В листинге 38.3 приведена программа, которая перечисляет локальные группы, зарегистрированные на сервере, используя для этого функцию `NetLocalGroupEnum`.

Листинг 38.3. Перечисление локальных групп

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )   // подключаем сетевую библиотеку

int main()
{
  DWORD  entries_read;           // количество элементов
  DWORD  total_entries;          // нумерация элементов
  LOCALGROUP_INFO_1  *buf_ptr = NULL;  // адрес буфера для данных
  NET_API_STATUS  ret_status;    // код возврата из функции

  // перечисляем локальные группы
  ret_status = NetLocalGroupEnum(
    NULL,                // локальный компьютер
    1,                   // получаем имя группы и комментарий
    (LPBYTE*)&buf_ptr,   // указатель на адрес буфера для данных
    MAX_PREFERRED_LENGTH,  // длина буфера по требованию
    &entries_read,       // указатель на количество элементов
    &total_entries,      // указатель на нумерацию элементов
    NULL);
  // проверяем завершение функции
  if (ret_status != NERR_Success)
  {
```

```
    printf("Net local group enumeration failed.\n");
    NetApiBufferFree(buf_ptr);   // освобождаем буфер для данных

    return ret_status;
  }

  printf("Local groups enumeration:\n");
  // выводим на консоль имена локальных групп и комментарии
  for (DWORD i = 0; i < entries_read; ++i)
  {
    wprintf(L"Name: %s\n", buf_ptr[i].lgrpi1_name);
    wprintf(L"Comment: %s\n", buf_ptr[i].lgrpi1_comment);
  }

  NetApiBufferFree(buf_ptr);     // освобождаем буфер для данных

  return 0;
}
```

38.4. Изменение информации о локальной группе

Для изменения имени локальной группы и комментариев о локальной группе используется функция `NetLocalGroupSetInfo`, которая имеет следующий прототип:

```
NET_API_STATUS  NetLocalGroupSetInfo(
  LPCWSTR  servername,   // имя сервера
  LPCWSTR  groupname,    // имя группы
  DWORD    level,        // уровень информации
  LPBYTE   buf,          // буфер с информацией
  LPDWORD  parm_err      // индексирование ошибки
);
```

Отметим, что успешно эту функцию могут выполнить только пользователи, которые являются администраторами или операторами учетных записей.

В случае успешного завершения функция `NetLocalGroupSetInfo` возвращает значение `NERR_Success`, а в случае неудачи возможны следующие коды завершения:

- `ERROR_ACCESS_DENIED` — пользователю отказано в доступе;
- `ERROR_INVALID_PARAMETER` — неправильный параметр;

- ERROR_NO_SUCH_ALIAS — не существует указанной локальной группы;
- NERR_InvalidComputer — неправильное имя компьютера;
- NERR_NotPrimary — операция может выполняться только на первичном контроллере домена.

Параметры функции NetLocalGroupSetInfo имеют следующее назначение.

Параметр servername должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение NULL.

Параметр groupname должен указывать на строку с именем группы, информация о которой изменяется. Эта строка должна иметь кодировку Unicode.

Параметр level указывает тип структуры, которая содержит информацию о локальной группе. Этот параметр может принимать одно из следующих значений:

- 0 — используется структура типа LOCALGROUP_INFO_0;
- 1 — используется структура типа LOCALGROUP_INFO_1;
- 1002 — используется структура типа LOCALGROUP_INFO_1002.

В примере будут использованы структуры типов LOCALGROUP_INFO_0 и LOCALGROUP_INFO_1. Тип структуры LOCALGROUP_INFO_1 был описан в *разд. 38.1*, поэтому здесь после описания функции приведем тип структуры LOCALGROUP_INFO_0 и опишем назначение полей этой структуры. Описание типа структуры LOCALGROUP_INFO_1002 может быть найдено в MSDN.

Параметр buf должен указывать на буфер с информацией о локальной группе. Информация должна храниться в структуре, тип которой задан параметром level.

В переменную, на которую указывает параметр parm_err, функция NetLocalGroupSetInfo помещает индекс первого поля в структуре с информацией о локальной группе, которое содержит неправильные данные и, как следствие, вызвало завершение функции NetLocalGroupSetInfo с ошибкой. В двойное слово, на которое указывает параметр, функция NetLocalGroupSetInfo может записать одно из следующих значений:

- LOCALGROUP_NAME_PARMNUM — неправильное имя группы;
- LOCALGROUP_COMMENT_PARMNUM — неправильный комментарий.

В этом параметре может быть установлено значение NULL. В этом случае индекс поля с неправильными данными не возвращается.

Теперь опишем структуру LOCALGROUP_INFO_0, которая имеет следующий тип:

```
typedef struct _LOCALGROUP_INFO_0 {
  LPWSTR  lgrpi0_name;     // имя группы
}LOCALGROUP_INFO_0, *PLOCALGROUP_INFO_0, *LPLOCALGROUP_INFO_0;
```

В единственном поле `lgrpi0_name` хранится указатель на строку с именем локальной группы. Длина строки не должна превышать `GNLEN` байт, а сама строка должна иметь кодировку Unicode.

В листинге 38.4 приведена программа, которая изменяет имя локальной группы, используя для этого функцию `NetLocalGroupSetInfo`.

Листинг 38.4. Изменение имени локальной группы

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )   // подключаем сетевую библиотеку

int main()
{
  wchar_t server_name[256] = L"\\\\";    // имя сервера
  wchar_t  old_name[GNLEN];          // старое имя локальной группы
  wchar_t  new_name[GNLEN];          // новое имя локальной группы
  wchar_t comment[MAXCOMMENTSZ];     // новый комментарий

  LOCALGROUP_INFO_0 group_info_0;  // информация об имени группы
  LOCALGROUP_INFO_1 group_info_1;  // описание новой группы

  NET_API_STATUS    ret_status;    // код возврата из функции

  printf("Input a server name: ");
// формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));

  printf("Input a local group name: ");
  wscanf(L"%s", old_name);           // вводим имя группы

  printf("Input a new name for the group: ");
  wscanf(L"%s", new_name);           // вводим новое имя группы

  // устанавливаем адрес имени в структуру
  group_info_0.lgrpi0_name = new_name;
```

```
  // изменяем имя группы
  ret_status = NetLocalGroupSetInfo(
    server_name,       // имя сервера
    old_name,          // имя группы
    0,                 // новое имя группы
    (LPBYTE)&group_info_0,  // новая информация о группе
    NULL);             // индексирования информации нет
  // проверяем завершение функции
  if (ret_status != NERR_Success)
  {
    printf("Net local group set name failed.\n");

    return ret_status;
  }
  printf("Local group name was changed.\n");

  printf("Input group comments: ");
  getwchar();          // очищаем поток
  _getws(comment);     // читаем комментарии о пользователе
// устанавливаем новый комментарий
  group_info_1.lgrpi1_comment = comment;

  // изменяем описание группы
  ret_status = NetLocalGroupSetInfo(
    server_name,       // имя сервера
    new_name,          // имя группы
    1,                 // получаем имя группы и комментарий
    (LPBYTE)&group_info_1,  // новое описание группы
    NULL);             // индексирования информации нет
  // проверяем завершение функции
  if (ret_status != NERR_Success)
  {
    printf("Net local group set info failed.\n");

    return ret_status;
  }
  printf("Local group comment was changed.\n");

  return 0;
}
```

38.5. Добавление членов локальной группы

Для добавления новых членов в локальную группу используется функция NetLocalGroupAddMembers, которая имеет следующий прототип:

```
NET_API_STATUS  NetLocalGroupAddMembers(
  LPCWSTR  servername,    // имя сервера
  LPCWSTR  groupname,     // имя группы
  DWORD    level,         // уровень информации
  LPBYTE   buf,           // буфер с информацией
  DWORD    totalentries   // количество включаемых членов
);
```

Отметим, что успешно эту функцию могут выполнить только те пользователи, которые являются администраторами или операторами учетных записей.

В случае успешного завершения функция NetLocalGroupAddMembers возвращает значение NERR_Success, а в случае неудачи возможны следующие коды завершения:

- NERR_GroupNotFound — группа не найдена;
- ERROR_NO_SUCH_MEMBER — не существует членов, которые добавляются в группу;
- ERROR_MEMBER_IN_ALIAS — некоторые члены уже являются членами группы;
- ERROR_INVALID_MEMBER — неправильный тип учетной записи добавляемого члена.

Параметры функции NetLocalGroupAddMembers имеют следующее назначение.

Параметр servername должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение NULL.

Параметр groupname должен указывать на строку с именем группы, в которую добавляются новые члены. Эта строка должна иметь кодировку Unicode.

Параметр level указывает тип структуры, которая содержит информацию о добавляемых в группу членах. Этот параметр может принимать одно из следующих значений:

- 0 — используется структура типа LOCALGROUP_MEMBERS_INFO_0;
- 3 — используется структура типа LOCALGROUP_MEMBERS_INFO_3.

Описание этих структур будет приведено после описания функции.

Параметр `buf` должен указывать на буфер с информацией о членах, которые добавляются в локальную группу. Информация должна храниться в структурах, тип которых задан параметром `level`.

Параметр `totalentries` должен содержать размерность массива структур, на который указывает параметр `buf`.

Теперь приведем типы структур, на которые может указывать параметр `level`.

Структура `LOCALGROUP_MEMBERS_INFO_0` имеет следующий тип:

```
typedef struct  _LOCALGROUP_MEMBERS_INFO_0 {
  PSID  lgrmi0_sid;      // указатель на идентификатор безопасности
} LOCALGROUP_MEMBERS_INFO_0, *PLOCALGROUP_MEMBERS_INFO_0,
  *LPLOCALGROUP_MEMBERS_INFO_0;
```

В единственном поле `lgrmi0_sid` этой структуры хранится указатель на идентификатор безопасности члена, который добавляется в локальную группу.

Структура `LOCALGROUP_MEMBERS_INFO_3` имеет следующее определение:

```
typedef struct  _LOCALGROUP_MEMBERS_INFO_3 {
  LPWSTR  lgrmi3_domainandname;  // указатель на имя учетной записи
} LOCALGROUP_MEMBERS_INFO_3, *PLOCALGROUP_MEMBERS_INFO_3,
  *LPLOCALGROUP_MEMBERS_INFO_3;
```

В единственном поле `lgrmi3_domainandname` этой структуры хранится указатель на строку с именем учетной записи, которая становится членом группы. Эта строка должна иметь кодировку Unicode, а имя должно быть задано в формате: `<имя_домена>\<имя_учетной_записи>`.

Теперь перейдем к примерам. Сначала, в листинге 38.5, приведем программу, которая добавляет члена локальной группы по его имени, используя для этого функцию `NetLocalGroupSetInfo`.

Листинг 38.5. Добавление члена локальной группы по имени

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )   // подключаем сетевую библиотеку

int main()
{
```

```
  wchar_t server_name[256] = L"\\\\";      // имя сервера
  wchar_t  group_name[GNLEN];              // имя локальной группы
  wchar_t user_name[UNLEN];                // имя пользователя
  LOCALGROUP_MEMBERS_INFO_3  member_info;  // информация о члене группы

  NET_API_STATUS  ret_status;      // код возврата из функции

  printf("Input server name: ");
// формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));

  printf("Input a local group name: ");
  wscanf(L"%s", group_name);         // вводим имя группы

  printf("Input a domain name: ");
  wscanf(L"%s", user_name);          // вводим имя домена

  wcscat(user_name, L"\\");          // присоединяем символ '\'

  printf("Input a user name: ");
// вводим имя пользователя
  wscanf(L"%s", user_name + wcslen(user_name));

  // устанавливаем информацию о пользователе
  member_info.lgrmi3_domainandname = user_name;

  // добавляем пользователя в локальную группу
  ret_status = NetLocalGroupAddMembers(
    server_name,           // имя сервера
    group_name,            // имя группы
    3,                     // уровень информации
    (LPBYTE)&member_info,  // имя учетной записи
    1);                    // добавляем одного члена группы
  if (ret_status != NERR_Success)
  {
    printf("Net local group add members failed.\n");

    return ret_status;
```

```
  }

  printf("The member is added.\n");

  return 0;
}
```

Теперь, в листинге 38.6, приведем программу, которая добавляет члена локальной группы по его идентификатору безопасности, используя для этого функцию `NetLocalGroupSetInfo`. Заметим, что в этой программе идентификатор безопасности учетной записи определяется при помощи функции `LookupAccountName`, которая будет подробно рассмотрена в *гл. 39*, посвященной работе с идентификаторами безопасности.

Листинг 38.6. Добавление члена локальной группы по дескриптору безопасности

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )   // подключаем сетевую библиотеку

int main()
{
  wchar_t server_name[256] = L"\\\\";      // имя сервера
  wchar_t  group_name[GNLEN];              // имя локальной группы
  wchar_t user_name[UNLEN];                // имя пользователя
  LOCALGROUP_MEMBERS_INFO_0  member_info;  // информация о члене группы

  NET_API_STATUS  ret_status;    // код возврата из функции
  DWORD  dwErrCode;

  DWORD  dwLengthOfSID = 0;          // длина SID
  DWORD  dwLengthOfDomainName = 0;   // длина имени домена
  PSID  lpSID = NULL;                // указатель на SID
```

```
  LPTSTR  lpDomainName = NULL;       // указатель на имя домена
  SID_NAME_USE type_of_SID;          // тип учетной записи

  printf("Input server name: ");
// формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));

  printf("Input a local group name: ");
  wscanf(L"%s", group_name);         // вводим имя группы

  printf("Input a user name: ");
  wscanf(L"%s", user_name);          // вводим имя пользователя

  // определяем длину SID пользователя
  LookupAccountName(
    NULL,              // ищем имя на локальном компьютере
    user_name,         // имя пользователя
    NULL,              // определяем длину SID
    &dwLengthOfSID,    // длина SID
    NULL,              // определяем имя домена
    &dwLengthOfDomainName,   // длина имени домена
    &type_of_SID);     // тип учетной записи
  // проверяем, вернула ли функция длину SID
  if (dwLengthOfSID == 0)
  {
    dwErrCode = GetLastError();
    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwErrCode);

    return dwErrCode;
  }

  // распределяем память для SID и имени домена
  lpSID = (PSID) new char[dwLengthOfSID];
  lpDomainName = (LPTSTR) new char[dwLengthOfDomainName];


  // определяем SID и имя домена пользователя
  if(!LookupAccountName(
```

```
    NULL,               // ищем имя на локальном компьютере
    user_name,          // имя пользователя
    lpSID,              // определяем длину SID
    &dwLengthOfSID,     // длина SID
    lpDomainName,       // определяем имя домена
    &dwLengthOfDomainName,  // длина имени домена
    &type_of_SID))      // тип учетной записи
  {
    dwErrCode = GetLastError();
    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwErrCode);

    return dwErrCode;
  }

  // распечатываем имя домена
  wprintf(L"%s\n", lpDomainName);


  // устанавливаем SID в информацию о члене группы
  member_info.lgrmi0_sid = lpSID;

  // добавляем пользователя в локальную группу
  ret_status = NetLocalGroupAddMembers(
    server_name,      // имя сервера
    group_name,       // имя группы
    0,                // уровень информации
    (LPBYTE)&member_info,  // информация о SID
    1);               // добавляем одного члена группы
  if (ret_status != NERR_Success)
  {
    printf("Net local group add members failed.\n");

    return ret_status;
  }

  printf("The member is added.\n");

  return 0;
}
```

38.6. Установка членов локальной группы

Под установкой членов локальной группы понимаются следующие действия. Задается список членов, которые должны входить в локальную группу. После этого из локальной группы исключаются все члены, которые не принадлежат этому списку, и включаются только члены из списка. Для установки членов локальной группы используется функция `NetLocalGroupSetMembers`, которая имеет следующий прототип:

```
NET_API_STATUS  NetLocalGroupSetMembers(
  LPCWSTR  servername,     // имя сервера
  LPCWSTR  groupname,      // имя группы
  DWORD    level,          // уровень информации
  LPBYTE   buf,            // буфер с информацией
  DWORD    totalentries    // количество структур в буфере
);
```

Отметим, что успешно эту функцию могут выполнить только пользователи, которые являются администраторами или операторами учетных записей.

В случае успешного завершения функция `NetLocalGroupSetMembers` возвращает значение `NERR_Success`, а в случае неудачи возможны следующие коды завершения:

- `NERR_GroupNotFound` — группа не найдена;
- `ERROR_NO_SUCH_MEMBER` — не существует членов, которые добавляются в группу;
- `ERROR_INVALID_MEMBER` — неправильный тип учетной записи добавляемого члена.

Параметры функции `NetLocalGroupAddMembers` имеют следующее назначение.

Параметр `servername` должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение `NULL`.

Параметр `groupname` должен указывать на строку с именем группы, в которую добавляются новые члены. Эта строка должна иметь кодировку Unicode.

Параметр `level` указывает тип структуры, которая содержит информацию о членах, которые должны входить в локальную группу. Этот параметр может принимать одно из следующих значений:

- 0 — используется структура типа `LOCALGROUP_MEMBERS_INFO_0`;
- 3 — используется структура типа `LOCALGROUP_MEMBERS_INFO_3`.

Типы этих структур были приведены в предыдущем разделе.

Параметр `buf` должен указывать на буфер с информацией о членах, которые должны входить в локальную группу. Информация должна храниться в структурах, тип которых задан параметром `level`.

Параметр `totalentries` должен содержать размерность массива, на который указывает параметр `buf`.

В листинге 38.7 приведена программа, которая устанавливает членов локальной группы, используя для этого функцию `NetLocalGroupSetMembers`.

Программа 38.7. Установка членов локальной группы.

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )   // подключаем сетевую библиотеку

int main()
{
  wchar_t server_name[256] = L"\\\\";      // имя сервера
  wchar_t  group_name[GNLEN];              // имя локальной группы
  LOCALGROUP_MEMBERS_INFO_3  *member_info; // информация о членах группы
  DWORD n;     // количество остающихся членов локальной группы

  NET_API_STATUS  ret_status;      // код возврата из функции

  printf("Input server name: ");
// формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));

  printf("Input a local group name: ");
  wscanf(L"%s", group_name);               // вводим имя группы

  printf("Input number of members: ");
  scanf("%d", &n);

// захватываем память для информации о пользователях
  member_info = new LOCALGROUP_MEMBERS_INFO_3[n];

  for (DWORD i = 0; i < n; ++i)
```

```
  {
    // захватываем память под имя пользователя
    member_info[i].lgrmi3_domainandname = new wchar_t[256];
    printf("Input a domain name: ");
    // вводим имя домена
    wscanf(L"%s", member_info[i].lgrmi3_domainandname);

    // присоединяем символ '\'
    wcscat(member_info[i].lgrmi3_domainandname, L"\\");

    printf("Input a user name: ");
    // вводим имя пользователя
    wscanf(L"%s", member_info[i].lgrmi3_domainandname +
            wcslen(member_info[i].lgrmi3_domainandname));
}

  // вводим имя домена
  for (i = 0; i < n; ++i)
    wprintf(L"%s\n", member_info[i].lgrmi3_domainandname);

  // устанавливаем членов в локальной группе
  ret_status = NetLocalGroupSetMembers(
    server_name,           // имя сервера
    group_name,            // имя группы
    3,                     // уровень информации
    (LPBYTE)member_info,   // информация о SID
    n);                    // количество остающихся членов группы
  if (ret_status != NERR_Success)
  {
    printf("Net local group set members failed.\n");

    return ret_status;
  }

  printf("The members are set.\n");

  return 0;
}
```

38.7. Перечисление членов локальной группы

Для перечисления членов локальной группы используется функция NetLocalGroupGetMembers, которая имеет следующий прототип:

```
NET_API_STATUS  NetLocalGroupGetMembers(
  LPCWSTR  servername,     // имя сервера
  LPCWSTR  localgroupname, // имя локальной группы
  DWORD    level,          // уровень информации
  LPBYTE   *bufptr,        // буфер для информации
  DWORD    prefmaxlen,     // длина буфера
  LPDWORD  entriesread,    // количество прочитанных структур
  LPDWORD  totalentries,   // общее количество структур
  LPDWORD  resumehandle    // дескриптор для продолжения чтения
);
```

В случае успешного завершения функция NetLocalGroupGetMembers возвращает значение NERR_Success, а в случае неудачи возможны следующие коды завершения:

- ERROR_ACCESS_DENIED — пользователю отказано в доступе;
- NERR_InvalidComputer — неправильное имя компьютера;
- ERROR_MORE_DATA — не все данные прочитаны;
- ERROR_NO_SUCH_ALIAS — не существует указанной локальной группы.

Параметры функции NetLocalGroupGetMembers имеют следующее назначение.

Параметр servername должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение NULL.

Параметр localgroupname должен указывать на строку с именем группы, члены которой перечисляются. Эта строка должна иметь кодировку Unicode.

Параметр level указывает тип структуры, которая содержит информацию о перечисляемых членах группы. Этот параметр может принимать одно из следующих значений:

- 0 — используется структура типа LOCALGROUP_MEMBERS_INFO_0;
- 1 — используется структура типа LOCALGROUP_MEMBERS_INFO_1;
- 2 — используется структура типа LOCALGROUP_MEMBERS_INFO_2;
- 3 — используется структура типа LOCALGROUP_MEMBERS_INFO_3.

В примере, который приведен в листинге 38.8, будет использована структура LOCALGROUP_MEMBERS_INFO_1, тип которой будет приведен после описания функции. Типы остальных структур можно найти в MSDN.

Параметр bufptr должен содержать адрес указателя, в который система запишет адрес буфера с информацией о членах локальной группы. Память под буфер распределяется системой, поэтому после прочтения информации из буфера эту память нужно освободить при помощи функции NetApiBufferFree, которая была описана в *разд. 37.2*. Эту память нужно освободить также и в том случае, если функция вернула код ошибки ERROR_MORE_DATA.

Параметр prefmaxlen должен содержать предпочтительную длину буфера в байтах. Если памяти для данных не хватит, то функция вернет код ошибки ERROR_MORE_DATA. В этом параметре можно установить значение MAX_PREFERRED_LENGTH — тогда система сама определит необходимую длину буфера для информации об учетных записях.

Параметр entriesread должен указывать на переменную типа DWORD, в которую функция запишет количество учетных записей, информация о которых записана в буфер данных.

Параметр totalentries должен указывать на переменную типа DWORD, в которую функция запишет общее количество учетных записей, которые являются членами локальной группы.

Параметр resume_handle должен указывать на переменную типа DWORD, в которую функция запишет дескриптор для продолжения перечисления учетных записей членов локальной группы. При первом вызове функции этот параметр должен содержать значение 0. Если продолжение чтения информации об учетных записях не предполагается, то в этом параметре нужно установить значение NULL.

Теперь опишем тип структуры LOCALGROUP_MEMBERS_INFO_1, которая будет использоваться в листинге 38.8.

```
typedef struct _LOCALGROUP_MEMBERS_INFO_1 {
  PSID    lgrmi1_sid;     // указатель на идентификатор безопасности
  SID_NAME_USE  lgrmi1_sidusage;   // тип учетной записи
  LPWSTR  lgrmi1_name;   // имя учетной записи
} LOCALGROUP_MEMBERS_INFO_1, *PLOCALGROUP_MEMBERS_INFO_1,
  *LPLOCALGROUP_MEMBERS_INFO_1;
```

Поля этой структуры имеют следующее назначение.

Поле lgrmi1_sid предназначено для хранения указателя на идентификатор безопасности члена локальной группы.

В поле lgrmi1_sidusage хранится тип учетной записи члена локальной группы. Этот тип может принимать одно из следующих значений, принадлежащих перечислимому типу SID_NAME_USE:

- SidTypeUser — учетная запись пользователя;

- SidTypeGroup — учетная запись группы;
- SidTypeWellKnownGroup — учетная запись предопределенной группы;
- SidTypeDeletedAccount — учетная запись удалена;
- SidTypeUnknown — тип учетной записи не определен.

В поле lgrmi1_name хранится указатель на имя учетной записи. Это имя не должно содержать имя домена.

В листинге 38.8 приведена программа, которая перечисляет членов локальной группы, используя для этого функцию NetLocalGroupGetMembers.

Листинг 38.8. Перечисление членов локальной группы

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )  // подключаем сетевую библиотеку

int main()
{
  wchar_t server_name[256] = L"\\\\";    // имя сервера
  wchar_t  group_name[GNLEN];            // имя локальной группы
  // информация о членах группы
  LOCALGROUP_MEMBERS_INFO_1  *member_info = NULL;
  DWORD entries_read;            // прочитанное количество членов группы
  DWORD total_entries;           // общее количество членов группы

  NET_API_STATUS  ret_status;    // код возврата из функции

  printf("Input server name: ");
  // формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));

  printf("Input a local group name: ");
  wscanf(L"%s", group_name);         // вводим имя группы

  // получаем информацию о членах локальной группы
  ret_status = NetLocalGroupGetMembers(
    server_name,         // имя сервера
```

```
    group_name,          // имя группы
    1,                   // уровень информации
    (LPBYTE*)&member_info,     // информация о пользователях
    MAX_PREFERRED_LENGTH,      // читаем информацию о всех членах группы
    &entries_read,       // прочитанное количество членов группы
    &total_entries,      // полное количество членов группы
    NULL);               // читаем в один прием
  if (ret_status != NERR_Success)
  {
    printf("Net local group get members failed.\n");
    NetApiBufferFree(member_info);      // освобождаем буфер

    return ret_status;
  }

  // распечатываем пользователей
  for (DWORD i = 0; i < entries_read; ++i)
    wprintf(L"The group member: %s\n", member_info[i].lgrmi1_name);

  NetApiBufferFree(member_info);    // освобождаем буфер
  return 0;
}
```

38.8. Удаление членов локальной группы

Для удаления членов локальной группы используется функция `NetLocalGroupDelMembers`, которая имеет следующий прототип:

```
NET_API_STATUS  NetLocalGroupDelMembers(
  LPCWSTR  servername,     // имя сервера
  LPCWSTR  groupname,      // имя группы
  DWORD    level,          // уровень информации
  LPBYTE   buf,            // буфер с информацией
  DWORD    totalentries    // количество структур в буфере
);
```

Отметим, что успешно эту функцию могут выполнить только пользователи, которые являются администраторами или операторами учетных записей.

В случае успешного завершения функция NetLocalGroupDelMembers возвращает значение NERR_Success, а в случае неудачи возможны следующие коды завершения:

- NERR_GroupNotFound — группа не найдена;
- ERROR_NO_SUCH_MEMBER — не существует членов, которые удаляются из группы;
- ERROR_MEMBER_IN_ALIAS — некоторые учетные члены не являются членами локальной группы, поэтому из группы не был удален ни один член.

Параметры функции NetLocalGroupDelMembers имеют следующее назначение.

Параметр servername должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение NULL.

Параметр groupname должен указывать на строку с именем группы, из которой удаляются члены. Эта строка должна иметь кодировку Unicode.

Параметр level указывает тип структуры, которая содержит информацию о членах, которые должны быть удалены из локальной группы. Этот параметр может принимать одно из следующих значений:

- 0 — используется структура типа LOCALGROUP_MEMBERS_INFO_0;
- 3 — используется структура типа LOCALGROUP_MEMBERS_INFO_3;

Типы этих структур были приведены в *разд. 38.5*.

Параметр buf должен указывать на буфер с информацией о членах, которые должны быть удалены из локальной группы. Информация должна храниться в структурах, тип которых задан параметром level.

Параметр totalentries должен содержать размерность массива структур, на который указывает параметр buf.

В листинге 38.9 приведена программа, которая удаляет члена локальной группы по имени при помощи функции NetLocalGroupDelMembers.

Листинг 38.9. Удаление члена локальной группы по имени

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )  // подключаем сетевую библиотеку

int main()
```

```
{
  wchar_t server_name[256] = L"\\\\";  // имя сервера
  wchar_t group_name[GNLEN];           // имя локальной группы
  wchar_t user_name[UNLEN];            // имя пользователя
  LOCALGROUP_MEMBERS_INFO_3  member_info;  // информация о члене группы

  NET_API_STATUS  ret_status;          // код возврата из функции

  printf("Input server name: ");
  // формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));

  printf("Input a local group name: ");
  wscanf(L"%s", group_name);          // вводим имя группы

  printf("Input a domain name: ");
  wscanf(L"%s", user_name);           // вводим имя домена

  wcscat(user_name, L"\\");           // присоединяем символ '\'

  printf("Input a user name: ");
  // вводим имя пользователя
  wscanf(L"%s", user_name + wcslen(user_name));

  // устанавливаем информацию о пользователе
  member_info.lgrmi3_domainandname = user_name;

  // удаляем пользователя из локальной группы
  ret_status = NetLocalGroupDelMembers(
    server_name,    // имя сервера
    group_name,     // имя группы
    3,              // уровень информации
    (LPBYTE)&member_info,      // информация о SID
    1);             // добавляем одного члена группы
  if (ret_status != NERR_Success)
  {
    printf("Net local group del members.\n");

    return ret_status;
```

```
  }

  printf("The member is deleted.\n");

  return 0;
}
```

В листинге 38.10 приведена программа, в которой член локальной группы удаляется по идентификатору безопасности при помощи функции `NetLocalGroupDelMembers`. При этом отметим, что идентификатор безопасности пользователя определяется при помощи функции `LookupAccountName`, которая будет подробно рассмотрена в *гл. 39*, посвященной работе с идентификаторами безопасности.

Листинг 38.10. Удаление члена локальной группы по идентификатору безопасности

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )  // подключаем сетевую библиотеку

int main()
{
  wchar_t server_name[256] = L"\\\\";    // имя сервера
  wchar_t  group_name[GNLEN];            // имя локальной группы
  wchar_t user_name[UNLEN];              // имя пользователя
  LOCALGROUP_MEMBERS_INFO_0  member_info;  // информация о члене группы

  NET_API_STATUS  ret_status;        // код возврата из функции
  DWORD  dwErrCode;

  DWORD  dwLengthOfSID = 0;          // длина SID
  DWORD  dwLengthOfDomainName = 0;   // длина имени домена
  PSID  lpSID = NULL;                // указатель на SID
```

```
LPTSTR  lpDomainName = NULL;       // указатель на имя домена
SID_NAME_USE type_of_SID;          // тип учетной записи

printf("Input server name: ");
// формируем имя сервера
wscanf(L"%s", server_name + wcslen(server_name));

printf("Input a local group name: ");
wscanf(L"%s", group_name);         // вводим имя группы

printf("Input a user name: ");
wscanf(L"%s", user_name);          // вводим имя пользователя

// определяем длину SID пользователя
LookupAccountName(
  NULL,            // ищем имя на локальном компьютере
  user_name,       // имя пользователя
  NULL,            // определяем длину SID
  &dwLengthOfSID,  // длина SID
  NULL,            // определяем имя домена
  &dwLengthOfDomainName,    // длина имени домена
  &type_of_SID);   // тип учетной записи
// проверяем, вернула ли функция длину SID
if (dwLengthOfSID == 0)
{
  dwErrCode = GetLastError();
  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwErrCode);

  return dwErrCode;
}

// распределяем память для SID и имени домена
lpSID = (PSID) new char[dwLengthOfSID];
lpDomainName = (LPTSTR) new char[dwLengthOfDomainName];

// определяем SID и имя домена пользователя
if(!LookupAccountName(
  NULL,            // ищем имя на локальном компьютере
```

```
    user_name,        // имя пользователя
    lpSID,            // определяем длину SID
    &dwLengthOfSID,   // длина SID
    lpDomainName,     // определяем имя домена
    &dwLengthOfDomainName,   // длина имени домена
    &type_of_SID))    // тип учетной записи
  {
    dwErrCode = GetLastError();
    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwErrCode);

    return dwErrCode;
  }

  // распечатываем имя домена
  wprintf(L"%s\n", lpDomainName);

  // устанавливаем SID в информацию о члене группы
  member_info.lgrmi0_sid = lpSID;

  // удаляем члена локальной группы
  ret_status = NetLocalGroupDelMembers(
    server_name,      // имя сервера
    group_name,       // имя группы
    0,                // уровень информации
    (LPBYTE)&member_info,     // информация о SID
    1);               // добавляем одного члена группы
  if (ret_status != NERR_Success)
  {
    printf("Net local group del members failed.\n");

    return ret_status;
  }

  printf("The member is deleted.\n");

  return 0;
}
```

38.9. Удаление локальной группы

Для удаления учетной записи локальной группы из базы данных менеджера учетных записей используется функция NetLocalGroupDel, которая имеет следующий прототип:

```
NET_API_STATUS  NetLocalGroupDel(
  LPCWSTR  servername,     // имя сервера
  LPCWSTR  groupname       // имя группы
);
```

Отметим, что успешно эту функцию могут выполнить только пользователи, которые являются администраторами или операторами учетных записей.

В случае успешного завершения функция NetLocalGroupDel возвращает значение NERR_Success, а в случае неудачи возможны следующие коды завершения:

- ERROR_ACCESS_DENIED — пользователю отказано в доступе;
- NERR_InvalidComputer — неправильное имя компьютера;
- NERR_NotPrimary — операция может выполняться только на первичном контроллере домена;
- NERR_GroupNotFound — группа не найдена;
- ERROR_NO_SUCH_ALIAS — группа не существует.

Параметры функции NetLocalGroupDelMembers имеют следующее назначение.

Параметр servername должен указывать на строку с именем сервера, на котором будет исполняться функция. Эта строка должна иметь кодировку Unicode и начинаться с символов \\. Если функция должна исполняться на локальном компьютере, то этот параметр должен иметь значение NULL.

Параметр groupname должен указывать на строку с именем группы, которая удаляется из базы данных менеджера учетных записей. Эта строка должна иметь кодировку Unicode.

В листинге 38.11 приведена программа, которая удаляет локальную группу при помощи функции NetLocalGroupDel.

Листинг 38.11. Удаление локальной группы

```
#include <stdio.h>
#include <windows.h>
#include <lm.h>

#pragma comment( lib, "netapi32.lib" )  // подключаем сетевую библиотеку
```

```
int main()
{
  wchar_t server_name[256] = L"\\\\";  // имя сервера
  wchar_t group_name[GNLEN];     // имя удаляемой локальной группы
  NET_API_STATUS ret_status;     // код возврата из функции

  printf("Input server name: ");
  // формируем имя сервера
  wscanf(L"%s", server_name + wcslen(server_name));

  printf("Input a name of the local group: ");
  wscanf(L"%s", group_name);     // вводим имя группы

  // удаляем локальную группу
  ret_status = NetLocalGroupDel(server_name, group_name);
  // проверяем код завершения
  if (ret_status != NERR_Success)
  {
    printf("Net local group del failed.\n");
    printf("Error code: %d\n", ret_status);

    return ret_status;
  }

  printf("The local group is deleted.\n");

  return 0;
}
```

Глава 39

Работа с идентификаторами безопасности

39.1. Структура идентификатора безопасности

Как было уже сказано в *разд. 36.5*, идентификатор безопасности (SID) является бинарным представлением учетной записи, которое используется операционной системой для управления доступом к объектам. В операционных системах Windows идентификаторы безопасности представляются структурой следующего типа:

```
typedef struct  _SID {
  BYTE  Revision;           // версия SID
  BYTE  SubAuthorityCount;  // количество относительных идентификаторов
  SID_IDENTIFIER_AUTHORITY  IdentifierAuthority;  // авторизация
  DWORD  SubAuthority[ANYSIZE_ARRAY]; // относительные идентификаторы
} SID;
```

Отсюда видно, что структура `SID` имеет переменную длину, которая зависит от количества относительных идентификаторов учетной записи. Идентификатор безопасности должен содержать идентификатор учетной записи, который хранится в поле `IdentifierAuthority` и, по крайней мере, один относительный идентификатор учетной записи.

Идентификатор учетной записи, который определяет ее уровень авторизации или, другими словами, принадлежность учетной записи какому-то множеству учетных записей, задается структурой типа:

```
typedef struct  _SID_IDENTIFIER_AUTHORITY {
  BYTE  Value[6];      // значение идентификатора учетной записи
} SID_IDENTIFIER_AUTHORITY, *PSID_IDENTIFIER_AUTHORITY;
```

Например, идентификатор безопасности, указывающий на локальный уровень авторизации, определяется следующим образом:

```
#define  SECURITY_LOCAL_SID_AUTHORITY  {0,0,0,0,0,2}
```

В операционных системах Windows предопределены следующие идентификаторы безопасности:

- `SECURITY_NULL_SID_AUTHORITY` — 0;
- `SECURITY_WORLD_SID_AUTHORITY` — 1;
- `SECURITY_LOCAL_SID_AUTHORITY` — 2;
- `SECURITY_CREATOR_SID_AUTHORITY` – 3;
- `SECURITY_NON_UNIQUE_AUTHORITY` — 4;
- `SECURITY_NT_AUTHORITY` — 5.

Каждый из них, естественно, определяется как массив байт.

39.2. Создание идентификатора безопасности

В этом разделе будут рассмотрены следующие функции: `GetSidLengthRequired`, `InitializeSid`, `GetSidSubAuthority`, `AllocateAndInitializeSid`, `IsValidSid`, которые используются при создании идентификаторов безопасности. Прежде чем переходить к описанию этих функций, кратко расскажем о порядке создания идентификатора безопасности. Так как структура `SID` имеет переменную длину, зависящую от количества относительных идентификаторов учетной записи, то, прежде чем создавать идентификатор безопасности, нужно определиться с количеством относительных идентификаторов, а затем зарезервировать память для структуры `SID`. Для определения длины структуры `SID`, учитывая количество относительных идентификаторов, используется функция `GetSidLengthRequired`. После этого резервируется память для структуры `SID`. Сама структура `SID` инициализируется при помощи функции `InitializeSid`, причем эта функция инициализирует только первых три члена структуры `SID`. Относительные идентификаторы должны инициализироваться отдельно, вручную. Для вычисления адреса относительного идентификатора безопасности по его индексу в структуре `SID` используется функция `GetSidSubAuthority`. После инициализации `SID` нужно проверить его структуру — для этого используется функция `IsValidSid`. Все вышеперечисленные действия можно также выполнить одной функцией `AllocateAndInitializeSid`, которая резервирует память для структуры типа `SID` и инициализирует ее. Однако в этом случае зарезервированную память нужно освобождать, используя функцию `FreeSid`.

Для определения длины буфера памяти, который потребуется для создания идентификатора безопасности, используется функция GetSidLengthRequired, которая имеет следующий прототип:

```
DWORD GetSidLengthRequired(
  UCHAR  nSubAuthorityCount // количество относительных идентификаторов
);
```

Единственный параметр этой функции должен содержать количество относительных идентификаторов, которые предполагается записать в идентификатор безопасности. Функция GetSidLengthRequired всегда успешно завершает свою работу и возвращает количество байт, требуемых для хранения идентификатора безопасности. Пример использования функции GetSidLengthRequired будет приведен далее, в программе из листинга 39.1, которая инициализирует идентификатор безопасности при помощи функции InitializeSid.

Для инициализации идентификатора безопасности используется функция InitializeSid, которая имеет следующий прототип:

```
BOOL  InitializeSid(
  PSID  pSid,               // указатель на идентификатор безопасности
  // указатель на идентификатор учетной записи
  PSID_IDENTIFIER_AUTHORITY  pIdentifierAuthority,
  BYTE  nSubAuthorityCount  // количество относительных идентификаторов
);
```

В случае успешного завершения функция InitializeSid возвращает ненулевое значение, а в случае неудачи — FALSE. При неудаче код ошибки можно определить посредством вызова функции GetLastError. Параметры функции InitializeSid имеют следующее назначение.

Параметр pSid должен указывать на структуру типа SID, которая будет инициализироваться. Память под структуру должна быть зарезервирована вызывающей программой.

Параметр pIdentifierAuthority должен указывать на структуру типа SID_IDENTIFIER_AUTHORITY, которая определяет идентификатор учетной записи. Значение, заданное в этой структуре, будет записано в идентификатор безопасности, на который указывает параметр pSid.

Параметр nSubAuthorityCount должен указывать на количество относительных идентификаторов учетной записи, которые будут записаны в идентификатор безопасности, соответствующий этой учетной записи. Функция InitializeSid записывает это значение в идентификатор безопасности. Сами значения относительных идентификаторов должны быть записаны в идентификатор безопасности отдельно. Для этого может использоваться функция GetSidSubAuthority.

Для определения адреса относительного идентификатора учетной записи в идентификаторе безопасности используется функция GetSidSubAuthority, которая имеет следующий прототип:

```
PDWORD  GetSidSubAuthority(
  PSID   pSid,           // указатель на идентификатор безопасности
  DWORD  nSubAuthority   // номер относительного идентификатора
);
```

В случае успешного завершения функция GetSidSubAuthority возвращает указатель на относительный идентификатор безопасности. Функция GetSidSubAuthority может завершиться неудачей в одном из двух случаев, а именно, если идентификатор безопасности имеет неправильную структуру или индекс относительного идентификатора выходит за пределы допустимых значений. В случае неудачного завершения функция GetSidSubAuthority вернет неопределенное значение. При неудаче код ошибки можно определить посредством вызова функции GetLastError.

Параметры функции GetSidSubAuthority имеют следующее назначение.

Параметр pSid должен указывать на структуру типа SID, в которой будет определяться адрес относительного идентификатора безопасности.

Параметр nSubAuthorityCount должен указывать на номер относительного идентификатора, адрес которого будет определяться.

Для проверки того, правильно ли в созданном идентификаторе безопасности установлена версия и не выходит ли количество относительных идентификаторов за допустимые пределы, используется функция IsValidSid, которая имеет следующий прототип:

```
BOOL  IsValidSid(
  PSID  pSid     // указатель на идентификатор безопасности
);
```

Единственный параметр этой функции должен указывать на идентификатор безопасности, который проверяется на правильность версии и допустимого количества относительных идентификаторов. В случае успешного завершения функция InitializeSid возвращает значение TRUE, а в случае неудачи — FALSE. При неудаче код ошибки можно определить посредством вызова функции GetLastError.

В листинге 39.1 приведена программа, в которой инициализируется идентификатор безопасности, используя функцию InitializeSid. Причем предварительно при помощи функции GetSidLengthRequired определяется длина идентификатора безопасности. После инициализации идентификатора безопасности при помощи функции GetSidSubAuthority определяется индекс относительного идентификатора, после чего устанавливается его значение. Затем проверяется правильность идентификатора безопасности при помощи вызова функции IsValidSid.

Листинг 39.1. Инициализация идентификатора безопасности

```
#include <stdio.h>
#include <windows.h>

int main()
{
  DWORD dwSidLength;   // длина памяти для SID
  SID *pSid = NULL;    // указатель на SID

  // указатель на относительный идентификатор учетной записи
  DWORD *pSubAuthority = NULL;
  // идентификатор учетной записи
  SID_IDENTIFIER_AUTHORITY sia = SECURITY_LOCAL_SID_AUTHORITY;

  // определеем длину SID с одним относительным идентификатором учетной записи
  dwSidLength = GetSidLengthRequired(1);

  printf("SID length: %u\n", dwSidLength);

  // захватываем память под SID
  pSid = (SID*) new UCHAR[dwSidLength];

  // инициализируем SID
  if(!InitializeSid(
    pSid,    // указатель на SID
    &sia,    // идентификатор учетной записи
    1))      // количество относительных идентификаторов
  {
    printf( "Initialized SID failed.\n");
    return 1;
  }
  printf("SID is initialized.\n");

  // определяем указатель на относительный идентификатор
  pSubAuthority = GetSidSubAuthority(pSid, 0);
  // устанавливаем значение относительного идентификатора
```

```
  *pSubAuthority = SECURITY_LOCAL_RID;

  // проверяем достоверность SID
  if(!IsValidSid(pSid))
  {
    printf( "The SID is not valid.\n");
    return 1;
  }

  printf("The SID is valid.\n");

  // освобождаем SID
  delete[] pSid;

  return 0;
}
```

Распределить память под идентификатор безопасности и инициализировать его поля можно одной функцией `AllocateAndInitializeSid`, которая имеет следующий прототип:

```
BOOL  AllocateAndInitializeSid (
  PSID_IDENTIFIER_AUTHORITY  pIdentifierAuthority,
  BYTE   nSubAuthorityCount,
  DWORD  nSubAuthority0,
  DWORD  nSubAuthority1,
  DWORD  nSubAuthority2,
  DWORD  nSubAuthority3,
  DWORD  nSubAuthority4,
  DWORD  nSubAuthority5,
  DWORD  nSubAuthority6,
  DWORD  nSubAuthority7,
  PSID   *pSid
);
```

В случае успешного завершения функция `AllocateAndInitializeSid` возвращает ненулевое значение, а в случае неудачи — `FALSE`. При неудаче код ошибки можно определить посредством вызова функции `GetLastError`. Параметры функции `AllocateAndInitializeSid` имеют следующее назначение.

Параметр `pIdentifierAuthority` должен указывать на структуру типа `SID_IDENTIFIER_AUTHORITY`, которая определяет идентификатор учетной

записи. Значение, заданное в этой структуре, будет записано в создаваемый идентификатор безопасности.

Параметр `nSubAuthorityCount` должен содержать количество относительных идентификаторов учетной записи, которые будут записаны в идентификатор безопасности, соответствующий этой учетной записи.

В параметрах `nSubAuthority0`, ..., `nSubAuthority7` должны быть установлены значения относительных идентификаторов, которые функция запишет в идентификатор безопасности учетной записи. В идентификатор безопасности будут записаны только первые `nSubAuthorityCount` относительных идентификаторов.

Параметр `pSid` должен содержать адрес переменной, в которую функция `AllocateAndInitializeSid` поместит указатель на созданную ей структуру типа `SID`. Память под структуру резервируется самой функцией.

После окончания работы с идентификатором безопасности, созданным функцией `AllocateAndInitializeSid`, эту память нужно освободить посредством вызова функции `FreeSid`, которая имеет следующий прототип:

```
PVOID  FreeSid(
  PSID  pSid     // указатель на идентификатор безопасности
);
```

В листинге 39.2 приведена программа, в которой идентификатор безопасности создается при помощи функции `AllocateAndInitializeSid`.

Листинг 39.2. Создание идентификатора безопасности

```
#include <stdio.h>
#include <windows.h>

int main()
{
  int n;   // количество идентификаторов
  // указатель на идентификатор учетной записи
  SID_IDENTIFIER_AUTHORITY *sia;
DWORD dwSubAuthority;    // относительный идентификатор учетной записи
  PSID pSid = NULL;      // указатель на SID

  // предопределенные идентификаторы учетной записи
  SID_IDENTIFIER_AUTHORITY s0 = SECURITY_NULL_SID_AUTHORITY;
  SID_IDENTIFIER_AUTHORITY s1 = SECURITY_WORLD_SID_AUTHORITY;
  SID_IDENTIFIER_AUTHORITY s2 = SECURITY_LOCAL_SID_AUTHORITY;
```

```
SID_IDENTIFIER_AUTHORITY s5 = SECURITY_NT_AUTHORITY;

// ввод идентификатора безопасности
printf("Choose one of the predefined SID.\n");
printf("Null SID:\t\t0\n");
printf("World SID:\t\t1\n");
printf("Local SID:\t\t2\n");
printf("Anonimous logon SID:\t5\n\n");

printf("SID = ");

scanf("%d", &n);     // читаем идентификатор учетной записи
switch (n)
{
case 0:
  sia = &s0;
  dwSubAuthority = SECURITY_NULL_RID;
  break;
case 1:
  sia = &s1;
  dwSubAuthority = SECURITY_NULL_RID;
  break;
case 2:
  sia = &s2;
  dwSubAuthority = SECURITY_NULL_RID;
  break;
case 5:
  sia = &s5;
  dwSubAuthority = SECURITY_NULL_RID;
  break;
default:
  printf("Unknown uiniversal SID.\n");
  return 1;
}

// создаем SID
if(!AllocateAndInitializeSid(
    sia,     // идентификатор учетной записи
    1,       // количество относительных идентификаторов учетной записи
```

```
      dwSubAuthority,        // первый RID
      0, 0, 0, 0, 0, 0,      // остальные RID равны 0
      &pSid) )
  {
    DWORD dwRetCode = GetLastError();
    printf( "Allocate and initialize sid failed %u\n", dwRetCode);

    return dwRetCode;
  }

  printf("The SID is allocated and initialized.\n");

  // освобождаем SID
  FreeSid(pSid);

  printf("The SID is freed.\n");

  return 0;
}
```

39.3. Определение учетной записи по идентификатору безопасности

Для определения имени учетной записи по ее идентификатору безопасности используется функция `LookupAccountSid`, которая имеет следующий прототип:

```
BOOL  LookupAccountSid(
  LPCSTR   lpSystemName,     // имя компьютера
  PSID     pSid,             // указатель на SID
  LPSTR    lpName,           // имя учетной записи
  LPDWORD  cbName,           // длина имени учетной записи
  LPSTR    ReferencedDomainName,     // имя домена
  LPDWORD  cbReferencedDomainName,   // длина имени домена
  PSID_NAME_USE  peUse       // тип идентификатора безопасности
);
```

В случае успешного завершения функция `LookupAccountSid` возвращает ненулевое значение, а в случае неудачи — `FALSE`. При неудаче код ошибки можно определить посредством вызова функции `GetLastError`.

Функция `LookupAccountSid` кроме имени учетной записи возвращает также и имя домена, в котором найдена учетная запись. Поиск учетной записи выполняется в следующем порядке. Сначала проверяются предопределенные общеизвестные учетные записи. Затем проверяются учетные записи, зарегистрированные на локальном компьютере. После этого проверяются учетные записи, зарегистрированные в первичном домене. И, в последнюю очередь, проверяются учетные записи из доверительных доменов. Параметры функции `LookupAccountSid` имеют следующее назначение.

Параметр `lpSystemName` должен указывать на строку с именем компьютера, на котором ищется имя учетной записи. Если этот параметр имеет значение `NULL`, то учетная запись ищется на локальном компьютере.

Параметр `pSid` должен указывать на структуру типа `SID`, которая содержит идентификатор безопасности учетной записи, имя которой нужно определить.

Параметр `lpName` должен указывать на буфер, в который функция `LookupAccountSid` запишет имя учетной записи, соответствующей идентификатору безопасности, заданному параметром `pSid`.

Параметр `cbName` должен указывать на переменную типа `DWORD`, которая содержит длину буфера для имени учетной записи, на который указывает параметр `lpName`. Если длина буфера недостаточна, то выполнение функции закончится неудачей, а в двойное слово, на которое указывает параметр `cbName`, функция запишет необходимую длину буфера.

Параметр `ReferencedDomainName` должен содержать адрес буфера, в который функция запишет имя домена, в котором найдена учетная запись. Если установить значение этого параметра в `NULL`, то функция запишет в двойное слово, на которое указывает параметр `cbReferencedDomainName`, необходимую длину буфера для имени домена. Отметим, что строка с именем домена имеет кодировку Unicode.

Параметр `cbReferencedDomainName` должен указывать на переменную типа `DWORD`, которая содержит длину буфера для имени домена, на который указывает параметр `ReferencedDomainName`. Если длина буфера недостаточна, то выполнение функции закончится неудачей, а в переменную, на которую указывает параметр `cbReferencedDomainName`, функция запишет необходимую длину буфера.

Параметр `peUse` должен указывать на переменную типа перечисление `SID_NAME_USE`, в которую функция запишет тип учетной записи. Типы учетных записей определяются как значения этого перечисления, которое определено следующим образом:

```
typedef enum  _SID_NAME_USE {
  SidTypeUser = 1,       // SID пользователя
  SidTypeGroup,          // SID группы
  SidTypeDomain,         // SID домена
```

```
    SidTypeAlias,          // SID синонима
    SidTypeWellKnownGroup, // SID предопределенной группы
    SidTypeDeletedAccount, // SID удаленной учетной записи
    SidTypeInvalid,        // недействительный SID
    SidTypeUnknown,        // неизвестный SID
    SidTypeComputer        // SID компьютера
  } SID_NAME_USE, *PSID_NAME_USE;
```

В листинге 39.3 приведена программа, которая определяет имя учетной записи и имя домена по идентификатору безопасности учетной записи, используя для этого функцию `LookupAccountSid`.

Листинг 39.3. Определение имени учетной записи по идентификатору безопасности

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <windows.h>
#include <lm.h>

int main()
{
  wchar_t user_name[UNLEN];    // имя пользователя

  HANDLE  hProcess;        // дескриптор процесса
  HANDLE  hTokenHandle;    // дескриптор маркера доступа

  DWORD  dwErrCode;        // код возврата

  TOKEN_OWNER  *lpTokenOwner = NULL;    // буфер для информации
  DWORD  dwLength= 0;      // длина буфера

  DWORD  dwLengthOfUserName = UNLEN;   // длина имени учетной записи
  DWORD  dwLengthOfDomainName = 0;     // длина имени домена
  LPTSTR  lpDomainName = NULL;         // указатель на имя домена
  SID_NAME_USE type_of_SID;            // тип учетной записи

  // получаем маркер доступа процесса
```

```
hProcess = GetCurrentProcess();
// получить маркер доступа процесса
if (!OpenProcessToken(
  hProcess,        // дескриптор процесса
  TOKEN_QUERY,     // запрос информации из маркера
  &hTokenHandle))  // дескриптор маркера
{
  dwErrCode = GetLastError();
  printf( "Open process token failed: %u\n", dwErrCode);

  return dwErrCode;
}
// определяем длину SID
if (!GetTokenInformation(
  hTokenHandle,  // дескриптор маркера доступа
  TokenOwner,    // получаем SID владельца
  NULL,          // нужна длина буфера
  0,             // пока длина равна 0
  &dwLength))    // для длины буфера
{
  dwErrCode = GetLastError();
  if (dwErrCode == ERROR_INSUFFICIENT_BUFFER)
    // захватываем память под SID
    lpTokenOwner = (TOKEN_OWNER*)new char[dwLength];
  else
  {
    // выходим из программы
    printf( "Get token information for length failed: %u\n",
       dwErrCode);
    return dwErrCode;
  }
}

// определяем имя владельца
if (!GetTokenInformation(
  hTokenHandle,  // дескриптор маркера доступа
  TokenOwner,    // нужен SID маркера доступа
  lpTokenOwner,  // адрес буфера для SID
  dwLength,      // длина буфера
```

```
  &dwLength))    // длина буфера
{
  dwErrCode = GetLastError();
  printf( "Get token information failed: %u\n", dwErrCode);

  return dwErrCode;
}

// определяем длину имени домена
if(!LookupAccountSid(
  NULL,                  // ищем на локальном компьютере
  lpTokenOwner->Owner,   // указатель на SID
  user_name,             // имя пользователя
  &dwLengthOfUserName,    // длина имени пользователя
  lpDomainName,          // определяем имя домена
  &dwLengthOfDomainName, // длина имени домена
  &type_of_SID))         // тип учетной записи
{
  dwErrCode = GetLastError();

  if (dwErrCode == ERROR_INSUFFICIENT_BUFFER)
      // распределяем память под имя домена
      lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];
  else
  {
    printf("Lookup account SID for length failed.\n");
    printf("Error code: %d\n", dwErrCode);

    return dwErrCode;
  }
}

// определяем имя учетной записи по SID
if(!LookupAccountSid(
  NULL,                  // ищем на локальном компьютере
  lpTokenOwner->Owner,   // указатель на SID
  user_name,             // имя пользователя
  &dwLengthOfUserName,   // длина имени пользователя
  lpDomainName,          // определяем имя домена
```

```
    &dwLengthOfDomainName, // длина имени домена
    &type_of_SID))         // тип учетной записи
  {
    dwErrCode = GetLastError();

    printf("Lookup account SID failed.\n");
    printf("Error code: %d\n", dwErrCode);

    return dwErrCode;
  }

  wprintf(L"User name: %s\n", user_name);
  wprintf(L"Domain name: %s\n", lpDomainName);

  // освобождаем память
  delete[] lpDomainName;
  delete[] lpTokenOwner;

  return 0;
}
```

Отметим, что в этой программе используются функции для работы с маркерами доступа. Более подробно работа с этими функциями будет рассмотрена в *гл. 41*, посвященной работе с маркерами доступа.

39.4. Определение идентификатора безопасности по имени учетной записи

Для определения идентификатора безопасности по имени учетной записи используется функция `LookupAccountName`, которая имеет следующий прототип:

```
BOOL  LookupAccountName(
  LPCSTR   lpSystemName,     // имя системы
  LPCSTR   lpAccountName,    // имя учетной записи
  PSID     pSid,             // буфер для идентификатора безопасности
  LPDWORD  cbSid,            // длина буфера для идентификатора
  LPSTR    ReferencedDomainName,   // буфер для имени домена
  LPDWORD  cbReferencedDomainName, // длина буфера для имени домена
  PSID_NAME_USE  peUse       // тип учетной записи
);
```

В случае успешного завершения функция LookupAccountName возвращает ненулевое значение, а в случае неудачи — FALSE. При неудаче код ошибки можно определить посредством вызова функции GetLastError.

Отметим, что кроме идентификатора безопасности функция LookupAccountName возвращает также имя домена, в котором найдена учетная запись. Поиск учетной записи выполняется в следующем порядке: сначала проверяются предопределенные общеизвестные учетные записи, затем проверяются учетные записи, зарегистрированные на локальном компьютере, после этого проверяются учетные записи, зарегистрированные в первичном домене, и, в последнюю очередь, проверяются учетные записи из доверительных доменов. Параметры функции LookupAccountName имеют следующее назначение.

Параметр lpSystemName должен указывать на строку с именем компьютера, на котором ищется имя учетной записи. Если этот параметр имеет значение NULL, то учетная запись ищется на локальном компьютере.

Параметр lpAccountName должен указывать на строку с именем учетной записи, для которой определяется идентификатор безопасности и имя домена.

Параметр pSid должен содержать адрес буфера, в который функция запишет структуру SID. Если установить значение этого параметра в NULL, то функция запишет в двойное слово, на которое указывает параметр cbSid, необходимую длину буфера для идентификатора безопасности.

Параметр cbSid должен указывать на переменную типа DWORD, которая содержит длину буфера для идентификатора безопасности, на который указывает параметр pSid. Если длина буфера недостаточна, то выполнение функции закончится неудачей, а в двойное слово, на которое указывает параметр cbSid, функция запишет необходимую длину буфера. Если в параметре pSid установлено значение NULL, то двойное слово, на которое указывает параметр cbSid, должно содержать 0.

Параметр ReferencedDomainName должен содержать адрес буфера, в который функция запишет имя домена, в котором найдена учетная запись. Если установить значение этого параметра в NULL, то функция запишет в переменную типа DWORD, на которую указывает параметр cbReferencedDomainName, необходимую длину буфера для имени домена. Отметим, что строка с именем домена имеет кодировку Unicode.

Параметр cbReferencedDomainName должен указывать на переменную типа DWORD, которая содержит длину буфера для имени домена, на который указывает параметр ReferencedDomainName. Если длина буфера недостаточна, то выполнение функции закончится неудачей, а в переменную, на которую указывает параметр cbReferencedDomainName, функция запишет необходимую длину буфера. Если в параметре ReferencedDomainName установлено значение NULL, то переменная, на которую указывает параметр cbReferencedDomainName, должна содержать 0.

Параметр `peUse` должен указывать на переменную типа `SID_NAME_USE`, в которую функция запишет тип учетной записи. Возможные значения этого перечисления были приведены в предыдущем разделе.

В листинге 39.8 приведена программа, которая определяет идентификатор безопасности и имя домена по имени учетной записи, используя для этого функцию `LookupAccountName`.

Листинг 39.4. Определение идентификатора безопасности по имени учетной записи

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <windows.h>
#include <lm.h>

int main()
{
  wchar_t user_name[UNLEN];     // имя пользователя

  DWORD dwErrCode;              // код возврата

  DWORD  dwLengthOfSID = 0;           // длина SID
  DWORD  dwLengthOfDomainName = 0;    // длина имени домена
  DWORD  dwLengthOfUserName = UNLEN; // длина имени учетной записи
  SID    *lpSID = NULL;               // указатель на SID
  LPTSTR  lpDomainName = NULL;        // указатель на имя домена
  SID_NAME_USE type_of_SID;           // тип учетной записи

  printf("Input a user name: ");
  wscanf(L"%s", user_name);     // вводим имя пользователя

  // определяем длину SID пользователя
  if (!LookupAccountName(
    NULL,           // ищем имя на локальном компьютере
    user_name,      // имя пользователя
    NULL,           // определяем длину SID
```

```
    &dwLengthOfSID,    // длина SID
    NULL,          // определяем имя домена
    &dwLengthOfDomainName,   // длина имени домена
    &type_of_SID)) // тип учетной записи
  {
    dwErrCode = GetLastError();

    if (dwErrCode == ERROR_INSUFFICIENT_BUFFER)
    {
      // распределяем память для SID и имени домена
      lpSID = (SID*) new char[dwLengthOfSID];
      lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];
    }
    else
    {
      // выходим из программы
      printf("Lookup account name failed.\n");
      printf("Error code: %d\n", dwErrCode);

      return dwErrCode;
    }
  }

  // определяем SID и имя домена пользователя
  if(!LookupAccountName(
    NULL,            // ищем имя на локальном компьютере
    user_name,       // имя пользователя
    lpSID,           // указатель на SID
    &dwLengthOfSID,  // длина SID
    lpDomainName,    // указатель на имя домена
    &dwLengthOfDomainName,   // длина имени домена
    &type_of_SID))   // тип учетной записи
  {
    dwErrCode = GetLastError();

    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwErrCode);

    return dwErrCode;
```

```
  }

  // выводим SID на консоль
  printf("SID revision: %u\n", lpSID->Revision);
  printf("SubAuthorityCount: %u\n", lpSID->SubAuthorityCount);
  printf("IdentifierAuthority: ");
  for (int i = 0; i < 6; ++i)
    printf("%u ", lpSID->IdentifierAuthority.Value[i]);
  printf("\n");
  printf("SubAuthorities: ");
  for (i = 0; i < lpSID->SubAuthorityCount; ++i)
    printf("%u ", lpSID->SubAuthority[i]);
  printf("\n");

  // выводим имя домена
  wprintf(L"Domain name: %s\n", lpDomainName);

  delete[] lpDomainName;
  delete[] lpSID;

  return 0;
}
```

39.5. Получение характеристик идентификатора безопасности

Для определения длины идентификатора безопасности используется функция `GetLengthSid`, которая имеет следующий прототип:

```
DWORD  GetLengthSid (
  PSID  pSid     // указатель на SID
);
```

Единственным параметром этой функции является указатель на идентификатор безопасности. Если идентификатор безопасности имеет правильную структуру, то функция возвращает длину этого идентификатора безопасности, в противном случае — возвращаемое значение не определено. Поэтому перед вызовом функции `GetLengthSid` нужно вызывать функцию `IsValidSid`, которая проверяет структуру идентификатора безопасности.

Для определения указателя на идентификатор авторизации в структуре SID используется функция GetSidIdentifierAuthority, которая имеет следующий прототип:

```
PSID_IDENTIFIER_AUTHORITY  GetSidIdentifierAuthority (
  PSID  pSid     // указатель на SID
);
```

Единственным параметром этой функции является указатель на идентификатор безопасности. Если идентификатор безопасности имеет правильную структуру, то функция возвращает указатель на поле IdentifierAuthority в идентификаторе безопасности, в противном случае — возвращаемое значение не определено. Поэтому перед вызовом функции GetSidIdentifierAuthority нужно вызывать функцию IsValidSid, которая проверяет структуру идентификатора безопасности.

Для определения количества относительных идентификаторов в идентификаторе безопасности используется функция GetSidSubauthorityCount, которая имеет следующий прототип:

```
PUCHAR  GetSidSubAuthorityCount(
  PSID  pSid     // указатель на SID
);
```

Единственным параметром этой функции является указатель на идентификатор безопасности. Если идентификатор безопасности имеет правильную структуру, то функция возвращает указатель на поле SubAuthorityCount в идентификаторе безопасности, в противном случае — возвращаемое значение не определено. Поэтому перед вызовом функции GetSidSubauthorityCount нужно вызывать функцию IsValidSid, которая проверяет структуру идентификатора безопасности.

После того как получено количество относительных идентификаторов, адрес конкретного относительного идентификатора можно определить посредством вызова функции GetSidSubAuthority, которая была подробно рассмотрена в *разд. 39.2*.

В листинге 39.5 приведена программа, которая выводит на консоль содержимое идентификатора безопасности, используя описанные ранее функции.

Листинг 39.5. Получение характеристик идентификатора безопасности

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
```

```
#include <windows.h>
#include <lm.h>

int main()
{
  wchar_t user_name[UNLEN];  // имя пользователя

  DWORD dwErrCode;           // код возврата

  DWORD  dwLengthOfSID = 0;          // длина SID
  DWORD  dwLengthOfDomainName = 0;   // длина имени домена
  DWORD  dwLengthOfUserName = UNLEN; // длина имени учетной записи
  SID    *lpSID = NULL;              // указатель на SID
  LPTSTR  lpDomainName = NULL;       // указатель на имя домена
  SID_NAME_USE type_of_SID;          // тип учетной записи

  // указатель на идентификатор авторизации SID
  PSID_IDENTIFIER_AUTHORITY  lpSia;
  // указатель на количество относительных идентификаторов
  PUCHAR  lpSubAuthorityCount;
  // указатель на относительный идентификатор
  DWORD  *lpSubAuthority;

  printf("Input a user name: ");
  wscanf(L"%s", user_name);      // вводим имя пользователя

  // определяем длину SID пользователя
  LookupAccountName(
    NULL,              // ищем имя на локальном компьютере
    user_name,         // имя пользователя
    NULL,              // определяем длину SID
    &dwLengthOfSID,    // длина SID
    NULL,              // определяем имя домена
    &dwLengthOfDomainName,   // длина имени домена
    &type_of_SID);     // тип учетной записи

  // проверяем, вернула ли функция длину SID
  if (dwLengthOfSID == 0)
  {
```

```
  dwErrCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwErrCode);

  return dwErrCode;
}

// распределяем память для SID и имени домена
lpSID = (SID*) new char[dwLengthOfSID];
lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];

// определяем SID и имя домена пользователя
if(!LookupAccountName(
  NULL,               // ищем имя на локальном компьютере
  user_name,          // имя пользователя
  lpSID,              // указатель на SID
  &dwLengthOfSID,     // длина SID
  lpDomainName,       // указатель на имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &type_of_SID))      // тип учетной записи
{
  dwErrCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwErrCode);

  return dwErrCode;
}

// получаем характеристики SID и выводим их на консоль
dwLengthOfSID = GetLengthSid(lpSID);
printf("Length of SID: %u\n", dwLengthOfSID);

lpSia = GetSidIdentifierAuthority(lpSID);
printf("IdentifierAuthority: ");
for (int i = 0; i < 6; ++i)
  printf("%u ", lpSia->Value[i]);
printf("\n");
```

```
  lpSubAuthorityCount = GetSidSubAuthorityCount(lpSID);
  printf("SubAuthorityCount: %u\n", *lpSubAuthorityCount);

  printf("SubAuthorities: ");
  for (i = 0; i < *lpSubAuthorityCount; ++i)
  {
    lpSubAuthority = GetSidSubAuthority(lpSID, i);
    printf("%u ", *lpSubAuthority);
  }
  printf("\n");

  delete[] lpDomainName;
  delete[] lpSID;

  return 0;
}
```

39.6. Копирование и сравнение идентификаторов безопасности

Для копирования идентификатора безопасности используется функция `CopySid`, которая имеет следующий прототип:

```
BOOL  CopySid(
  DWORD   nDestinationSidLength,   // длина буфера для SID
  PSID    pDestinationSid,         // указатель на буфер
  PSID    pSourceSid               // указатель на SID
);
```

В случае успешного завершения функция `CopySid` возвращает ненулевое значение, а в случае неудачи — `FALSE`. При неудаче код ошибки можно определить посредством вызова функции `GetLastError`. Параметры функции `CopySid` имеют следующее назначение.

Параметр `nDestinationSidLength` должен содержать длину буфера, в который будет копироваться `SID`.

Параметр `DestinationSid` должен указывать на буфер, в который будет копироваться `SID`.

Параметр `pSourceSid` должен указывать на `SID`, который будет копироваться в буфер.

Для сравнения двух идентификаторов безопасности используется функция EqualSid, которая имеет следующий прототип:

```
BOOL  EqualSid(
  PSID  pSid1,   // указатель на первый сравниваемый SID
  PSID  pSid2    // указатель на второй сравниваемый SID
);
```

Параметрами функции являются указатели на идентификаторы безопасности, которые будут сравниваться. Если сравниваемые идентификаторы безопасности совпали, то функция EqualSid возвратит ненулевое значение, в противном случае — FALSE. При неудаче код ошибки можно определить посредством вызова функции GetLastError. Если хотя бы один из сравниваемых идентификаторов безопасности имеет неправильную структуру, то возвращаемое функцией значение не определено.

Идентификатор безопасности без последнего относительного идентификатора называется префиксом идентификатора безопасности. Сравнение префиксов используется для определения принадлежности двух учетных записей одной области авторизации. Например, учетные записи групп, принадлежащих одному домену, будут иметь одинаковые префиксы:

$$S - R - I - SA_0$$

Для сравнения префиксов двух идентификаторов безопасности используется функция EqualPrefixSid, которая имеет следующий прототип:

```
BOOL  EqualPrefixSid(
  PSID  pSid1,
  PSID  pSid2
);
```

Параметрами функции являются указатели на идентификаторы безопасности, префиксы которых будут сравниваться. Если сравниваемые префиксы совпали, то функция EqualPrefixSid возвратит ненулевое значение, в противном случае — FALSE. При неудаче код ошибки можно определить посредством вызова функции GetLastError.

В листинге 39.6 приведена программа, которая сравнивает префиксы двух групп, используя функцию EqualPrefixSid.

Листинг 39.6. Сравнение префиксов идентификаторов безопасности

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
```

```
#include <windows.h>
#include <lm.h>

int main()
{
  wchar_t group_name[GNLEN];   // имя группы

  DWORD dwErrCode;             // код возврата

  DWORD  dwLengthOfSID = 0;          // длина SID
  DWORD  dwLengthOfDomainName = 0;   // длина имени домена
  DWORD  dwLengthOfUserName = UNLEN; // длина имени учетной записи
  SID    *lpSID_1 = NULL;            // указатель на SID первой группы
  SID    *lpSID_2 = NULL;            // указатель на SID второй группы
  LPTSTR  lpDomainName_1 = NULL;     // указатель на имя домена
  LPTSTR  lpDomainName_2 = NULL;     // указатель на имя домена
  SID_NAME_USE type_of_SID;          // тип учетной записи

  printf("Input a name of the first group: ");
  wscanf(L"%s", group_name);       // вводим имя группы

  // определяем длину SID группы
  LookupAccountName(
    NULL,              // ищем имя на локальном компьютере
    group_name,        // имя пользователя
    NULL,              // определяем длину SID
    &dwLengthOfSID,    // длина SID
    NULL,              // определяем имя домена
    &dwLengthOfDomainName,   // длина имени домена
    &type_of_SID);     // тип учетной записи

  // проверяем, вернула ли функция длину SID
  if (dwLengthOfSID == 0)
  {
    dwErrCode = GetLastError();

    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwErrCode);

    return dwErrCode;
```

```
}

// распределяем память для SID и имени домена
lpSID_1 = (SID*) new char[dwLengthOfSID];
lpDomainName_1 = (LPTSTR) new wchar_t[dwLengthOfDomainName];

// определяем SID и имя домена группы
if(!LookupAccountName(
  NULL,              // ищем имя на локальном компьютере
  group_name,        // имя группы
  lpSID_1,           // указатель на SID
  &dwLengthOfSID,    // длина SID
  lpDomainName_1,    // указатель на имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &type_of_SID))     // тип учетной записи
{
  dwErrCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwErrCode);

  return dwErrCode;
}

// проверяем тип учетной записи
if (type_of_SID != SidTypeAlias)
{
  printf("This is not a group.\n");
  return 1;
}

printf("Input a name of the second group: ");
wscanf(L"%s", group_name);       // вводим имя группы

// определяем длину SID группы
LookupAccountName(
  NULL,              // ищем имя на локальном компьютере
  group_name,        // имя пользователя
  NULL,              // определяем длину SID
```

```
  &dwLengthOfSID,     // длина SID
  NULL,               // определяем имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &type_of_SID);      // тип учетной записи

// проверяем, вернула ли функция длину SID
if (dwLengthOfSID == 0)
{
  dwErrCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwErrCode);

  return dwErrCode;
}

// распределяем память для SID и имени домена
lpSID_2 = (SID*) new char[dwLengthOfSID];
lpDomainName_2 = (LPTSTR) new wchar_t[dwLengthOfDomainName];

// определяем SID и имя домена группы
if(!LookupAccountName(
  NULL,               // ищем имя на локальном компьютере
  group_name,         // имя группы
  lpSID_2,            // указатель на SID
  &dwLengthOfSID,     // длина SID
  lpDomainName_2,     // указатель на имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &type_of_SID))      // тип учетной записи
{
  dwErrCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwErrCode);

  return dwErrCode;
}

// проверяем тип учетной записи
```

```
  if (type_of_SID != SidTypeAlias)
  {
    printf("This is not a group.\n");
    return 1;
  }

  // сравниваем префиксы SID групп
  if (EqualPrefixSid(lpSID_1, lpSID_2))
    printf("SID prefixes are equal\n");
  else
    printf("SID prefixes are not equal\n");

  delete[] lpDomainName_1;
  delete[] lpDomainName_2;
  delete[] lpSID_1;
  delete[] lpSID_2;

  return 0;
}
```

39.7. Строковое представление идентификатора безопасности

Для преобразования информации, хранящейся в структуре `SID`, в строку символов используется функция `ConvertSidToStringSid`, которая имеет следующий прототип:

```
BOOL  ConvertSidToStringSid(
  PSID   pSid,           // указатель на SID
  LPSTR  *StringSid      // адрес указателя на строку
);
```

В случае успешного завершения функция возвратит ненулевое значение, а в случае неудачи — `FALSE`. При неудаче код ошибки можно определить посредством вызова функции `GetLastError`, которая может вернуть один из двух возможных кодов:

- `ERROR_NOT_ENOUGH_MEMORY` — недостаточно памяти;
- `ERROR_NONE_MAPPED` — не найдена учетная запись, соответствующая `SID`.

Параметры функции имеют следующее назначение.

Параметр `pSid` должен указывать на идентификатор безопасности, который будет преобразован в строковое представление.

Параметр `StringSid` должен содержать адрес указателя, в который функция поместит адрес на строку с символьным представлением идентификатора безопасности. Память под строку резервирует сама функция. После завершения работы со строковым представлением идентификатора безопасности эту память нужно освободить при помощи функции `LocalFree`.

В листинге 39.5 приведена программа, в которой идентификатор безопасности преобразуется в строковое представление, используя функцию `ConvertSidToStringSid`.

Листинг 39.7. Преобразование SID в строку

```
#define _WIN32_WINNT 0x0500

#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <windows.h>
#include <lm.h>
#include <sddl.h>

int main()
{
  wchar_t user_name[UNLEN];     // имя пользователя

  DWORD dwErrCode;              // код возврата

  DWORD  dwLengthOfSID = 0;     // длина SID
  DWORD  dwLengthOfDomainName = 0;   // длина имени домена
  PSID  lpSID = NULL;           // указатель на SID
  LPTSTR  lpDomainName = NULL; // указатель на имя домена
  SID_NAME_USE type_of_SID;     // тип учетной записи

  LPTSTR  lpStringSID = NULL;  // указатель на строку с SID

  printf("Input a user name: ");
```

```
wscanf(L"%s", user_name);      // вводим имя пользователя

// определяем длину SID пользователя
LookupAccountName(
  NULL,               // ищем имя на локальном компьютере
  user_name,          // имя пользователя
  NULL,               // определяем длину SID
  &dwLengthOfSID,     // длина SID
  NULL,               // определяем имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &type_of_SID);      // тип учетной записи

// проверяем, вернула ли функция длину SID
if (dwLengthOfSID == 0)
{
  dwErrCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwErrCode);

  return dwErrCode;
}

// распределяем память для SID и имени домена
lpSID = (PSID) new wchar_t[dwLengthOfSID];
lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];


// определяем SID и имя домена пользователя
if(!LookupAccountName(
  NULL,               // ищем имя на локальном компьютере
  user_name,          // имя пользователя
  lpSID,              // указатель на SID
  &dwLengthOfSID,     // длина SID
  lpDomainName,       // указатель на имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &type_of_SID))      // тип учетной записи
{
  dwErrCode = GetLastError();
```

```
    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwErrCode);

    return dwErrCode;
  }

  // преобразуем SID в строку
  if (!ConvertSidToStringSid(lpSID, &lpStringSID))
  {
    printf("Convert SID to string SID failed.");

    return GetLastError();
  }
  // распечатываем SID
  wprintf(L"SID: %s\n", lpStringSID);
  // распечатываем имя домена
  wprintf(L"Domain name: %s\n", lpDomainName);

  // освобождаем память
  LocalFree(lpStringSID);

  delete[] lpDomainName;
  delete[] lpSID;

  return 0;
}
```

Для преобразования символьного представления идентификатора безопасности в структуру типа `SID` используется функция `ConvertStringSidToSid`, которая имеет следующий прототип:

```
BOOL  ConvertStringSidToSid(
  LPCSTR  StringSid,   // указатель на строку
  PSID    *pSid        // адрес указателя на SID
);
```

В случае успешного завершения функция возвратит ненулевое значение, а в случае неудачи — `FALSE`. При неудаче код ошибки можно определить посредством вызова функции `GetLastError`, которая может вернуть один из двух возможных кодов:

- `ERROR_INVALID_PARAMETER` — неверный параметр;
- `ERROR_INVALID_SID` — неверный SID.

Параметры функции имеют следующее назначение.

Параметр `StringSid` должен указывать на строку с символьным представлением идентификатора безопасности.

Параметр `pSid` должен содержать адрес указателя, в который функция поместит адрес на структуру типа `SID`. Память под структуру резервирует сама функция. После завершения работы со структурой эту память нужно освободить при помощи функции `LocalFree`.

В листинге 39.8 приведена программа, которая преобразует символьное представление идентификатора безопасности в структуру типа `SID`.

Листинг 39.8. Преобразование строки с идентификатором безопасности в структуру типа SID

```
#define _WIN32_WINNT 0x0500

#include <stdio.h>
#include <windows.h>
#include <lm.h>
#include <sddl.h>

int main()
{
  int n;   // количество относительных идентификаторов учетной записи
  LPTSTR  lpStringSID = NULL;  // указатель на строку с SID
  PSID  lpSID = NULL;          // указатель на SID
  LPTSTR  lpString;            // текущий указатель на SID

  printf("Input a number of Sub Authorities: ");
  scanf("%d", &n);
  lpStringSID = (LPTSTR) new char[10 + 5 * n];

  lpStringSID[0] = 'S';        // признак строки SID
  lpStringSID[1] = '-';
  lpStringSID[2] = '1';        // версия 1
  lpStringSID[3] = '-';

  // устанавливаем указатель на идентификатор учетной записи
  lpString = lpStringSID + 4;

  printf("Input an Identifier Authority: ");
  scanf("%s", lpString);       // вводим идентификатор учетной записи
```

```
  lpString += strlen(lpString);

  *lpString++ = '-';

  for (int i = 0; i < n - 1; ++i)
  {
    printf("Input Sub Authority #%d: ", i);
    // вводим относительный идентификатор учетной записи
    scanf("%s", lpString);
    lpString += strlen(lpString);

    *lpString++ = '-';
  }
  printf("Input Sub Authority #%d: ", n - 1);
  // вводим последний относительный идентификатор учетной записи
  scanf("%s", lpString);

  // распечатываем полученный SID
  printf("SID: %s\n", lpStringSID);

  // преобразуем строку в SID
  if (!ConvertStringSidToSid(lpStringSID, &lpSID))
  {
    DWORD dwErrCode = GetLastError();
    printf("Convert string SID to SID failed.\n");
    printf("Last error code: %u\n", dwErrCode);

    return dwErrCode;
  }

  // проверяем SID на достоверность
  if(!IsValidSid(lpSID))
  {
    printf( "The SID is not valid.\n");
    return 1;
  }
  printf("The SID is valid.\n");

  // освобождаем память
  LocalFree(lpSID);

  return 0;
}
```

Глава 40

Работа с дескрипторами безопасности

40.1. Форматы дескрипторов безопасности

Как уже говорилось в *гл. 36*, каждый охраняемый объект в операционных системах Windows имеет дескриптор безопасности, который используется операционной системой для ограничения доступа к этому объекту. Дескриптор безопасности может храниться в двух форматах: абсолютном и относительном. В обоих случаях дескриптор безопасности описывается структурой типа:

```
typedef struct  _SECURITY_DESCRIPTOR {
  BYTE  Revision;     // версия
  BYTE  Sbz1;         // 0 для выравнивания на границу 16 бит
  SECURITY_DESCRIPTOR_CONTROL  Control;    // управляющие флаги
  PSID  Owner;        // указатель на SID владельца объекта
  PSID  Group;        // указатель на SID первичной группы владельца
  PACL  Sacl;         // указатель на список SACL
  PACL  Dacl;         // указатель на список DACL
} SECURITY_DESCRIPTOR, *PISECURITY_DESCRIPTOR;
```

Тип `SECURITY_DESCRIPTOR_CONTROL` определяется следующим образом:

```
typedef  WORD  SECURITY_DESCRIPTOR_CONTROL;
```

То есть является символическим именем типа `WORD`, который определяет слово памяти. В этом слове могут устанавливаться следующие флаги, управляющие состоянием дескриптора безопасности:

- `SE_OWNER_DEFAULTED` — владелец задан по умолчанию;
- `SE_GROUP_DEFAULTED` — первичная группа владельца задана по умолчанию;

- `SE_DACL_PRESENT` — присутствует список DACL;
- `SE_DACL_DEFAULTED` — список DACL задан по умолчанию;
- `SE_SACL_PRESENT` — присутствует список SACL;
- `SE_SACL_DEFAULTED` — список SACL задан по умолчанию;
- `SE_SELF_RELATIVE` — дескриптор безопасности в относительной форме.

Кроме того, начиная с версии Windows 2000, в дескрипторе безопасности могут быть установлены следующие флаги:

- `SE_DACL_AUTO_INHERITED` — DACL установлен механизмом наследования;
- `SE_SACL_AUTO_INHERITED` — SACL установлен механизмом наследования;
- `SE_DACL_PROTECTED` — DACL защищен от наследования;
- `SE_SACL_PROTECTED` — SACL защищен от наследования.

Различие между абсолютным и относительным форматами дескриптора безопасности заключается в том, что в случае абсолютного формата дескриптор безопасности содержит только указатели на данные, которые описывают ограничения на доступ к объекту, а не сами данные. А при относительном формате дескриптор безопасности также содержит и данные, которые описывают ограничения доступа. Эти данные хранятся вслед за структурой в одном буфере, хотя порядок хранения этих данных не определен. В случае абсолютного формата структура типа `SECURITY_DESCRIPTOR` также содержит адреса данных, но сами данные могут храниться в любой области памяти. Таким образом, можно сказать, что дескриптор безопасности в относительном формате содержит как указатели на данные, так и сами данные, а дескриптор безопасности в абсолютном формате содержит только адреса данных.

Если функция Win32 API возвращает дескриптор безопасности, то этот дескриптор безопасности представлен в относительном формате. В качестве параметра функции дескриптор безопасности может передаваться как в абсолютном, так и в относительном формате.

Возможно преобразование дескриптора безопасности из одного формата в другой. Для преобразования дескриптора безопасности из абсолютного формата в относительный используется функция `MakeSelfRelativeSD`, которая имеет следующий прототип:

```
BOOL  MakeSelfRelativeSD(
  PSECURITY_DESCRIPTOR  pAbsoluteSD,       // адрес абсолютного SD
  PSECURITY_DESCRIPTOR  pSelfRelativeSD,   // адрес относительного SD
  LPDWORD  lpdwBufferLength                // адрес длины буфера
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. При неудачном завершении функции код

ошибки можно получить посредством вызова функции GetLastError. Параметры функции MakeSelfRelativeSD имеют следующее назначение.

Параметр pAbsoluteSD должен содержать адрес исходного дескриптора безопасности в абсолютном формате.

Параметр pSelfRelativeSD должен указывать на буфер, в который функция запишет дескриптор безопасности в относительном формате, полученный из дескриптора безопасности в абсолютном формате.

Параметр lpdwBufferLength должен указывать на переменную типа DWORD, в которой хранится длина буфера для дескриптора безопасности в относительном формате. Если эта длина меньше необходимой, то функция закончится неудачей и запишет по адресу lpdwBufferLength необходимую длину буфера.

Для преобразования дескриптора безопасности из относительного формата в абсолютный используется функция MakeAbsoluteSD, которая имеет следующий прототип:

```
BOOL  MakeAbsoluteSD(
  PSECURITY_DESCRIPTOR  pSelfRelativeSD,   // адрес относительного SD
  PSECURITY_DESCRIPTOR  pAbsoluteSD,       // адрес абсолютного SD
  LPDWORD  lpdwAbsoluteSDSize,   // указатель на длину абсолютного SD
  PACL     pDacl,                // адрес DACL
  LPDWORD  lpdwDaclSize,         // адрес длины DACL
  PACL     pSacl,                // адрес SACL
  LPDWORD  lpdwSaclSize,         // адрес длины SACL
  PSID     pOwner,               // адрес SID владельца
  LPDWORD  lpdwOwnerSize,        // адрес длины SID владельца
  PSID     pPrimaryGroup,        // адрес SID первичной группы
  LPDWORD  lpdwPrimaryGroupSize  // адрес длины SID первичной группы
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — FALSE. При неудачном завершении функции код ошибки можно получить посредством вызова функции GetLastError. Параметры функции MakeAbsoluteSD имеют следующее назначение.

Параметр pSelfRelativeSD должен указывать на исходный дескриптор безопасности в относительном формате.

Параметр pAbsoluteSD должен указывать на буфер, в который функция запишет дескриптор безопасности в абсолютном формате, полученный из дескриптора безопасности в относительном формате.

Параметр lpdwAbsoluteSDSize должен указывать на переменную типа DWORD, в которой хранится длина буфера для дескриптора безопасности

в абсолютном формате. Если эта длина меньше необходимой, то функция закончится неудачей и запишет по адресу `lpdwAbsoluteSDSize` необходимую длину буфера.

Параметр `pDacl` должен указывать на буфер, куда функция запишет список DACL.

Параметр `lpdwDaclSize` должен указывать на переменную типа `DWORD`, в которой хранится длина буфера для списка DACL. Если эта длина меньше необходимой, то функция закончится неудачей и запишет необходимую длину буфера по адресу `lpdwDaclSize`.

Параметр `pSacl` должен указывать на буфер, куда функция запишет список SACL.

Параметр `lpdwSaclSize` должен указывать на переменную типа `DWORD`, в которой хранится длина буфера для списка SACL. Если эта длина меньше необходимой, то функция закончится неудачей и запишет по адресу `lpdwSaclSize` необходимую длину буфера.

Параметр `pOwner` должен указывать на буфер, куда функция запишет идентификатор безопасности владельца объекта.

Параметр `lpdwOwnerSize` должен указывать на переменную типа `DWORD`, в которой хранится длина буфера для идентификатора безопасности владельца объекта. Если эта длина меньше необходимой, то функция закончится неудачей и запишет по адресу `lpdwOwnerSize` необходимую длину буфера.

Параметр `pPrimaryGroup` должен указывать на буфер, куда функция запишет идентификатор безопасности первичной группы владельца объекта.

Параметр `lpdwPrimaryGroupSize` должен указывать на переменную типа `DWORD`, в которой хранится длина буфера для идентификатора безопасности первичной группы владельца объекта. Если эта длина меньше необходимой, то функция закончится неудачей и запишет по адресу `lpdwPrimaryGroupSize` необходимую длину буфера.

В *разд. 40.3* приведена программа, в которой дескриптор из абсолютного формата преобразуется в дескриптор в относительном формате, используя функцию `MakeSelfRelativeSD`.

40.2. Создание нового дескриптора безопасности

Сначала кратко опишем порядок создания дескриптора безопасности, а затем рассмотрим функции, которые используются при его создании. Очевидно, что дескриптор безопасности создается в абсолютной форме, т. к. данные, описывающие ограничения на доступ к объекту, обычно хранятся в различных областях памяти. Первым делом нужно зарезервировать память

для дескриптора безопасности. Если память резервируется статически, то нужно просто объявить переменную типа SECURITY_DESCRIPTOR. При динамическом распределении памяти необходимо зарезервировать блок памяти, длина которого равна, по крайней мере, SECURITY_DESCRIPTOR_MIN_LENGTH байт. После этого нужно инициализировать поля дескриптора безопасности. Для этого используется следующие функции. Версия дескриптора безопасности устанавливается при помощи функции InitializeSecurityDesriptor. Владелец дескриптора безопасности устанавливается посредством вызова функции SetSecurityDescriptorOwner. Первичная группа владельца дескриптора безопасности устанавливается посредством вызова функции SetSecurityDescriptorGroup. Списки управления доступом к объекту устанавливаются посредством вызова функций SetSecurityDescriptorDacl и SetSecurityDescriptorSacl. После инициализации дескриптора безопасности можно проверить его структуру. Для этого используется функция IsValidSecurityDescriptor.

В этом разделе создадим дескриптор безопасности, который только содержит владельца объекта и его первичную группу. Создание списков управления доступом к объекту и подключение их к дескриптору безопасности будет рассмотрено в следующем разделе. Там же будут подробно рассмотрены и функции подключения этих списков к дескриптору безопасности, а именно функции SetSecurityDescriptorDacl и SetSecurityDescriptorSacl. Теперь же перейдем к подробному описанию остальных упомянутых функций.

Для инициализации версии дескриптора безопасности используется функция InitializeSecurityDesriptor, которая имеет следующий прототип:

```
BOOL  InitializeSecurityDescriptor(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,   // указатель на SD
  DWORD  dwRevision                            // версия SD
);
```

В случае успешного завершения функция возвращает ненулевое значение, в противном случае — FALSE. При неудачном завершении функции код ошибки можно получить посредством вызова функции GetLastError. Параметры функции InitializeSecurityDesriptor имеют следующее назначение.

Параметр pSecurityDescriptor должен указывать на структуру типа SECURITY_DESCRIPTOR, версия которой инициализируется.

Параметр dwRevision должен содержать значение SECURITY_DESCRIPTOR_REVISION.

Для установки владельца дескриптора безопасности используется функция SetSecurityDescriptorOwner, которая имеет следующий прототип:

```
BOOL  SetSecurityDescriptorOwner(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,   // указатель на SD
  PSID  pOwner,            // указатель на SID владельца
```

```
  BOOL  bOwnerDefaulted    // флаг задания владельца по умолчанию
);
```

В случае успешного завершения функция возвращает ненулевое значение, в противном случае — FALSE. При неудачном завершении функции код ошибки можно получить посредством вызова функции GetLastError. Параметры функции SetSecurityDescriptorOwner имеют следующее назначение.

Параметр pSecurityDescriptor должен указывать на структуру типа SECURITY_DESCRIPTOR, в которую устанавливается владелец.

Параметр pOwner должен указывать на идентификатор безопасности учетной записи, которая будет владельцем объекта.

В параметре bOwnerDefaulted должно быть установлено значение TRUE или FALSE. Если установлено значение TRUE, то в поле Control дескриптора безопасности будет установлен флаг SE_OWNER_DEFAULTED. В противном случае этот флаг будет сброшен. Если в дескрипторе безопасности установлен флаг SE_OWNER_DEFAULTED, то для определения владельца объекта система будет использовать механизм, заданный по умолчанию. В противном случае владелец объекта будет определяться из дескриптора безопасности.

Для установки первичной группы владельца дескриптора безопасности используется функция SetSecurityDescriptorGroup, которая имеет следующий прототип:

```
BOOL  SetSecurityDescriptorGroup(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,   // указатель на SD
  PSID  pGroup,          // указатель на SID первичной группы
  BOOL  bGroupDefaulted  // флаг первичной группы по умолчанию
);
```

В случае успешного завершения функция возвращает ненулевое значение, в противном случае — FALSE. При неудачном завершении функции код ошибки можно получить посредством вызова функции GetLastError.

Опишем параметры функции SetSecurityDescriptorGroup.

Параметр pSecurityDescriptor должен указывать на структуру типа SECURITY_DESCRIPTOR, в которую устанавливается группа-владелец.

Параметр pGroup должен указывать на идентификатор безопасности учетной записи первичной группы владельцем объекта. Значение этого параметра может быть равно NULL. В этом случае предполагается, что владелец объекта не имеет первичной группы.

В параметре bGroupDefaulted должно быть установлено значение TRUE или FALSE. Если установлено значение TRUE, то в поле Control дескриптора безопасности будет установлен флаг SE_GROUP_DEFAULTED. В противном случае этот флаг будет сброшен. Если в дескрипторе безопасности установлен

флаг SE_GROUP_DEFAULTED, то для определения первичной группы владельца объекта система будет использовать механизм, заданный по умолчанию. В противном случае первичная группа владельца объекта будет определяться из дескриптора безопасности.

После инициализации дескриптора безопасности нужно проверить его структуру. Для этого используется функция IsValidSecurityDescriptor, которая имеет следующий прототип:

```
BOOL  IsValidSecurityDescriptor(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor  // указатель на SD
);
```

В случае успешного завершения функция возвращает ненулевое значение, в противном случае — FALSE. При неудачном завершении функции код ошибки можно получить посредством вызова функции GetLastError. Единственный параметр этой функции должен указывать на дескриптор безопасности, структура которого проверяется.

В листинге 40.1 приведена программа, в которой инициализируется дескриптор безопасности для нового каталога. При этом владелец каталога и первичная группа владельца каталога определяются операционной системой, используя механизм, заданный по умолчанию. В этом случае владельцем каталога является пользователь, от имени которого запущена программа. Так как списки управления доступом создаются пустыми, то доступ к каталогу разрешен всем пользователям. Подробно работа со списками доступа будет рассмотрена в следующем разделе.

Листинг 40.1. Создание дескриптора безопасности для нового объекта

```
#include <windows.h>
#include <stdio.h>
#include <lm.h>

int main()
{

  char chDirName[248];        // имя каталога
  SECURITY_DESCRIPTOR sd;     // дескриптор безопасности каталога
  SECURITY_ATTRIBUTES sa;     // атрибуты защиты каталога
  DWORD dwErrCode;            // код возврата

  // инициализируем версию дескриптора безопасности
  if (!InitializeSecurityDescriptor(
```

```
    &sd,
    SECURITY_DESCRIPTOR_REVISION))
  {
    dwErrCode = GetLastError();
    printf("Initialize security descroptor failed.\n");
    printf("Error code: %d\n", dwErrCode);

    return dwErrCode;
  }

  // устанавливаем SID владельца объекта
  if (!SetSecurityDescriptorOwner(
    &sd,          // адрес дескриптора безопасности
    NULL,         // не задаем владельца
    SE_OWNER_DEFAULTED))  // определить владельца по умолчанию
  {
    dwErrCode = GetLastError();
    perror("Set security descriptor owner failed.\n");
    printf("The last error code: %u\n", dwErrCode);

    return dwErrCode;
  }

  // устанавливаем SID первичной группы владельца
  if (!SetSecurityDescriptorGroup(
    &sd,          // адрес дескриптора безопасности
    NULL,         // не задаем первичную группу
    SE_GROUP_DEFAULTED))  // определить первичную группу по умолчанию
  {
    dwErrCode = GetLastError();
    perror("Set security descriptor group failed.\n");
    printf("The last error code: %u\n", dwErrCode);

    return dwErrCode;
  }

  // проверяем структуру дескриптора безопасности
  if (!IsValidSecurityDescriptor(&sd))
  {
```

```
    dwErrCode = GetLastError();
    perror("Security descriptor is invalid.\n");
    printf("The last error code: %u\n", dwErrCode);

    return dwErrCode;
  }

  // инициализируем атрибуты безопасности
  sa.nLength = sizeof(sa);   // устанавливаем длину атрибутов защиты
  sa.lpSecurityDescriptor = &sd;   // устанавливаем адрес SD
  sa.bInheritHandle = FALSE;       // дескриптор каталога ненаследуемый

  printf("Input a directory name: ");
  scanf("%s", chDirName);          // вводим имя каталога

  // создаем каталог
  if (!CreateDirectory(chDirName, &sa))
  {
    dwErrCode = GetLastError();
    perror("Security descriptor is invalid.\n");
    printf("The last error code: %u\n", dwErrCode);

    return dwErrCode;
  }

  printf("The directory is created.\n");

  return 0;
}
```

Для того чтобы эта программа создала новый каталог, нужно, чтобы процесс обладал привилегией SE_TAKE_OWNERSHIP. Отметим, что, если процесс, создающий новый объект, пытается установить владельцем этого объекта пользователя, отличного от пользователя, от имени которого запущен процесс, то система может выдать ошибку. Это произойдет потому, что данный процесс обычно обладает только привилегией SE_TAKE_OWNERSHIP. А для того, чтобы процесс мог установить владельцем объекта любого пользователя, который может быть владельцем объекта, этот процесс должен обладать привилегией SE_RESTORE_NAME. Владельцем же объекта может быть только пользователь, который наделен привилегией SE_TAKE_OWNERSHIP_NAME. Подробно о работе с привилегиями будет рассказано в *гл. 43*.

40.3. Определение длины дескриптора безопасности

Для определения длины дескриптора безопасности используется функция `GetSecurityDescriptorLength`, которая имеет следующий прототип:

```
DWORD  GetSecurityDescriptorLength(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor  // указатель на SD
);
```

Если дескриптор безопасности имеет правильную структур, то функция возвратит его длину, включая длину всех связанных с этим дескриптором полей, в противном случае — неопределенное значение. Поэтому перед вызовом функции `GetSecurityDescriptorLength` нужно вызвать функцию `IsValidSecurityDescriptor`. Единственным параметром этой функции является указатель на дескриптор безопасности, длину которого определяет функция. Можно сказать, что вызов этой функции позволяет определить длину буфера, необходимую для преобразования дескриптора из абсолютной формы в относительную форму.

В листинге 40.2 приведена программа, которая определяет длину дескриптора безопасности, используя функцию `GetSecurityDescriptorLength`.

Листинг 40.2. Определение длины дескриптора безопасности и преобразование дескриптора безопасности из абсолютной в относительную форму

```
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hProcess;         // дескриптор процесса
  HANDLE  hTokenHandle;     // дескриптор маркера доступа

  TOKEN_OWNER  *lpOwnerSid = NULL;   // указатель на буфер
  DWORD  dwOwnerLength= 0;           // длина буфера

  TOKEN_OWNER  *lpPrimaryGroupSid = NULL;  // указатель на буфер
  DWORD  dwPrimaryGroupLength= 0;          // длина буфера

  SECURITY_DESCRIPTOR  sd;     // SD в абсолютной форме
  SECURITY_DESCRIPTOR  *lpSd;  // указатель на SD в относительной форме
```

```
  DWORD dwErrCode;       // код возврата
DWORD dwLength;        // длина дескриптора безопасности

// получить дескриптор процесса
hProcess = GetCurrentProcess();
// получить маркер доступа процесса
if (!OpenProcessToken(
  hProcess,        // дескриптор процесса
  TOKEN_QUERY,     // запрос информации из маркера
  &hTokenHandle))  // дескриптор маркера
{
  dwErrCode = GetLastError();
  cout << "Open process token failed: " << dwErrCode << endl;

  return dwErrCode;
}
// определяем длину SID владельца
if (!GetTokenInformation(
  hTokenHandle,      // дескриптор маркера доступа
  TokenOwner,        // получаем SID владельца
  NULL,              // нужна длина буфера
  0,                 // пока длина равна 0
  &dwOwnerLength))   // для длины буфера
{
  if (dwOwnerLength != 0)
    // захватываем память под SID
    lpOwnerSid = (TOKEN_OWNER*)new char[dwOwnerLength];
  else
  {
    // выходим из программы
    dwErrCode = GetLastError();
    cout << "Get token information for length failed: "
         << dwErrCode << endl;

    return dwErrCode;
  }
}

// определяем SID владельца
```

```
if (!GetTokenInformation(
  hTokenHandle,      // дескриптор маркера доступа
  TokenOwner,        // нужен SID маркера доступа
  lpOwnerSid,        // адрес буфера для SID
  dwOwnerLength,     // длина буфера
  &dwOwnerLength))   // длина буфера
{
  dwErrCode = GetLastError();
  cout << "Get token information failed: " << dwErrCode << endl;

  return dwErrCode;
}

// определяем длину SID первичной группы владельца
if (!GetTokenInformation(
  hTokenHandle,            // дескриптор маркера доступа
  TokenPrimaryGroup,       // получаем SID первичной группы
  NULL,                    // нужна длина буфера
  0,                       // пока длина равна 0
  &dwPrimaryGroupLength))  // для длины буфера
{
  if (dwPrimaryGroupLength != 0)
    // захватываем память под SID
    lpPrimaryGroupSid = (TOKEN_OWNER*)new char[dwPrimaryGroupLength];
  else
  {
    // выходим из программы
    dwErrCode = GetLastError();
    cout << "Get token information for length failed: "
         << dwErrCode << endl;

    return dwErrCode;
  }
}

// определяем SID первичной группы владельца
if (!GetTokenInformation(
  hTokenHandle,            // дескриптор маркера доступа
  TokenPrimaryGroup,       // нужен SID первичной группы
```

```
    lpPrimaryGroupSid,        // адрес буфера для SID
    dwPrimaryGroupLength,     // длина буфера
    &dwPrimaryGroupLength))   // длина буфера
  {
    dwErrCode = GetLastError();
    cout << "Get token information failed: " << dwErrCode << endl;

    return dwErrCode;
  }

  // инициализируем дескриптор безопасности
  if (!InitializeSecurityDescriptor(     // инициализируем версию
    &sd,
    SECURITY_DESCRIPTOR_REVISION))
  {
    dwErrCode = GetLastError();
    cerr << "Initialize security descriptor failed." << endl
      << "The last error code: " << dwErrCode << endl;
    cout << "Press any key to finish." << endl;
    cin.get();

    return dwErrCode;
  }

  // устанавливаем SID владельца объекта
  if (!SetSecurityDescriptorOwner(
    &sd,
    lpOwnerSid->Owner,
    FALSE))
  {
    dwErrCode = GetLastError();
    cerr << "Set security descriptor owner failed." << endl
      << "The last error code: " << dwErrCode << endl;
    cout << "Press any key to finish." << endl;
    cin.get();

    return dwErrCode;
  }
```

```
// устанавливаем SID первичной группы владельца
if (!SetSecurityDescriptorGroup(
  &sd,
  lpPrimaryGroupSid->Owner,
  FALSE))
{
  dwErrCode = GetLastError();
  cerr << "Set security descriptor group failed." << endl
    << "The last error code: " << dwErrCode << endl;
  cout << "Press any key to finish." << endl;
  cin.get();

  return dwErrCode;
}

// проверяем структуру дескриптора безопасности
if (!IsValidSecurityDescriptor(&sd))
{
  dwErrCode = GetLastError();
  cerr << "The security descriptor is invalid." << endl
    << "The last error code: " << dwErrCode << endl;
  cout << "Press any key to finish." << endl;
  cin.get();

  return dwErrCode;
}

// печатаем длину структуры SECURITY_DESCRIPTOR
cout << "Length of SECURITY_DESCRIPTOR structure: "
     << sizeof(sd) << endl;

// получаем и печатаем длину дескриптора безопасности
// в абсолютной форме
dwLength = GetSecurityDescriptorLength(&sd);
cout << "Length of security descriptor in absolute form: "
     << dwLength << endl;

// создаем дескриптор безопаснотсти в относительной форме
lpSd = (SECURITY_DESCRIPTOR*)new char[dwLength];
```

```
  if (!MakeSelfRelativeSD(&sd, lpSd, &dwLength))
  {
    dwErrCode = GetLastError();
    cerr << "Make self relative SD failed." << endl
      << "The last error code: " << dwErrCode << endl;
    cout << "Press any key to finish." << endl;
    cin.get();

    return dwErrCode;
  }

  cout << "The self realtive security descriptor is made." << endl;

  // освобождаем память
  delete[] lpOwnerSid;
  delete[] lpPrimaryGroupSid;
  delete[] lpSd;

  // закрываем дескрипторы
  CloseHandle(hTokenHandle);

  return 0;
}
```

40.4. Получение дескриптора безопасности по имени объекта

Для получения дескриптора безопасности объекта по имени этого объекта используется функция `GetNamedSecurityInfo`, которая имеет следующий прототип:

```
  DWORD  GetNamedSecurityInfo(
    LPSTR  pObjectName,          // указатель на имя объекта
    SE_OBJECT_TYPE  ObjectType,  // тип объекта
    SECURITY_INFORMATION  SecurityInfo,  // управляющие флаги
    PSID  *ppsidOwner,         // адрес указателя на SID владельца
    PSID  *ppsidGroup,         // адрес указателя на SID группы
    PACL  *ppDacl,             // адрес указателя на DACL
    PACL  *ppSacl,             // адрес указателя на SACL
    PSECURITY_DESCRIPTOR  *ppSecurityDescriptor  // адрес указателя на SD
);
```

В случае успешного завершения функция возвращает значение ERROR_SUCCESS, а в случае неудачи — код ошибки. Параметры функции GetNamedSecurityInfo имеют следующее назначение.

Параметр pObjectName должен указывать на имя объекта, дескриптор безопасности которого возвращает функция.

Параметр ObjectType должен содержать тип объекта, дескриптор безопасности которого возвращает функция. Тип объекта должен быть одним из значений перечисления SE_OBJECT_TYPE:

```
typedef enum  _SE_OBJECT_TYPE
{
  SE_UNKNOWN_OBJECT_TYPE = 0,  // неизвестный тип
  SE_FILE_OBJECT,         // файл
  SE_SERVICE,             // сервис
  SE_PRINTER,             // принтер
  SE_REGISTRY_KEY,        // ключ регистра
  SE_LMSHARE,             // разделяемый объект локальной сети
  SE_KERNEL_OBJECT,       // объект ядра
  SE_WINDOW_OBJECT,       // оконная станция на локальном компьютере
  SE_DS_OBJECT,           // сервис каталогов
  SE_DS_OBJECT_ALL,       // сервис каталогов с их свойствами
  SE_PROVIDER_DEFINED_OBJECT // Windows Management Instrumentation
}  SE_OBJECT_TYPE;
```

Отметим, что последние три значения определены только для версий старше Windows 2000 (включительно).

Параметр SecurityInfo должен содержать переменную типа SECURITY_INFORMATION, в которой установлены флаги, отмечающие информацию, которую должен содержать дескриптор безопасности. Тип SECURITY_INFORMATION определен следующим образом:

```
typedef  DWORD  SECURITY_INFORMATION,  *PSECURITY_INFORMATION;
```

То есть представляет собой переменную типа DWORD. При работе с функцией GetNamedSecurityInfo в этой переменной могут быть установлены любые комбинации следующих флагов:

- OWNER_SECURITY_INFORMATION — получить владельца объекта;
- GROUP_SECURITY_INFORMATION — получить первичную группу владельца;
- DACL_SECURITY_INFORMATION — получить список DACL;
- SACL_SECURITY_INFORMATION — получить список SACL.

Параметр ppsidOwner должен содержать адрес указателя, в который функция запишет указатель на идентификатор безопасности владельца объекта.

Этот указатель будет записан только в том случае, если в параметре SecurityInfo установлен флаг OWNER_SECURITY_INFORMATION. Если SID владельца объекта не нужен, то значение этого параметра может быть установлено в NULL.

Параметр ppsidGroup должен содержать адрес указателя, в который функция запишет указатель на идентификатор безопасности первичной группы владельца объекта. Этот указатель будет записан только в том случае, если в параметре SecurityInfo установлен флаг GROUP_SECURITY_INFORMATION. Если SID первичной группы владельца объекта не нужен, то значение этого параметра может быть установлено в NULL.

Параметр ppDacl должен содержать адрес указателя, в который функция запишет указатель на список DACL из дескриптора безопасности объекта. Этот указатель будет записан только в том случае, если в параметре SecurityInfo установлен флаг DACL_SECURITY_INFORMATION. Если список DACL объекта не нужен, то значение этого параметра может быть установлено в NULL.

Параметр ppSacl должен содержать адрес указателя, в который функция запишет указатель на список SACL из дескриптора безопасности объекта. Этот указатель будет записан только в том случае, если в параметре SecurityInfo установлен флаг SACL_SECURITY_INFORMATION. Если список SACL объекта не нужен, то значение этого параметра может быть установлено в NULL.

Замечание

Для получения идентификаторов безопасности владельца объекта и первичной группы владельца объекта, а также списка DACL необходимо, чтобы процесс открыл доступ к объекту в режиме READ_CONTROL. Для этого нужно, чтобы процесс имел соответствующие права доступа к объекту, которые должны быть установлены в списке DACL. Кроме того, для получения списка SACL процесс должен обладать привилегией SE_SECURITY_NAME.

В листинге 40.3 приведена программа, в которой функция GetNamedSecurityInfo используется для получения идентификаторов безопасности владельца объекта и первичной группы владельца объекта по имени объекта.

Листинг 40.3. Получение идентификаторов безопасности владельца объекта и первичной группы по имени объекта

```
#define _WIN32_WINNT 0x0500

#include <stdio.h>
#include <windows.h>
```

```
#include <lm.h>
#include <aclapi.h>
#include <sddl.h>

int main()
{
  char chFileName[256];  // имя файла

  PSID  pSidOwner;       // указатель на SID владельца объекта
  PSID  pSidGroup;       // указатель на SID первичной группы объекта

  PSECURITY_DESCRIPTOR  pSecurityDescriptor;  // указатель на SD

  LPTSTR lpStringSid;    // указатель на строку SID

  DWORD dwRetCode;       // код возврата

  // вводим имя файла, например: C:\\test.txt
  printf("Input a full path to your file: ");
  gets(chFileName);

  // получаем дескриптор безопасности файла
  dwRetCode = GetNamedSecurityInfo(
    chFileName,          // имя файла
    SE_FILE_OBJECT,      // объект файл
    GROUP_SECURITY_INFORMATION | OWNER_SECURITY_INFORMATION,
    &pSidOwner,          // адрес указателя на SID владельца
    &pSidGroup,          // адрес указателя на первичную группу
    NULL,                // указатель на DACL не нужен
    NULL,                // указатель на SACL не нужен
    &pSecurityDescriptor);  // адрес указателя на SD
  if (dwRetCode != ERROR_SUCCESS)
  {
    printf("Get named security info failed.\n");
    printf("Error code: %u\n", dwRetCode);

    return dwRetCode;
  }
```

```
  // преобразуем SID владельца в строку
  if (!ConvertSidToStringSid(pSidOwner, &lpStringSid))
  {
    printf("Convert SID to string SID failed.");
    dwRetCode = GetLastError();

    return dwRetCode;
  }
  // печатаем SID владельца
  printf("Owner SID: %s\n", lpStringSid);
  // освобождаем память для строки
  LocalFree(lpStringSid);

  // преобразуем SID первичной группы в строку
  if (!ConvertSidToStringSid(pSidGroup, &lpStringSid))
  {
    printf("Convert SID to string SID failed.");
    dwRetCode = GetLastError();

    return dwRetCode;
  }
  // печатаем SID первичной группы
  printf("Group SID: %s\n", lpStringSid);
  // освобождаем память для строки
  LocalFree(lpStringSid);

  // освобождаем память для дескриптора
  LocalFree(pSecurityDescriptor);

  return 0;
}
```

40.5. Получение дескриптора безопасности по дескриптору объекта

Для получения дескриптора безопасности объекта по дескриптору этого объекта используется функция `GetSecurityInfo`, которая имеет следующий прототип:

```
  DWORD  GetSecurityInfo(
    HANDLE  handle,           // дескриптор объекта
```

```
    SE_OBJECT_TYPE  ObjectType,          // тип объекта
    SECURITY_INFORMATION  SecurityInfo,  // управляющие флаги
    PSID  *ppsidOwner,       // адрес указателя на SID владельца
    PSID  *ppsidGroup,       // адрес указателя на SID группы
    PACL  *ppDacl,           // адрес указателя на DACL
    PACL  *ppSacl,           // адрес указателя на SACL
    PSECURITY_DESCRIPTOR  *ppSecurityDescriptor  // адрес указателя на SD
);
```

В случае успешного завершения функция возвращает значение `ERROR_SUCCESS`, а в случае неудачи — код ошибки. Все параметры этой функции за исключением первого имеют то же назначение, что и соответствующие параметры функции `GetNamedSecurityInfo`. Первый же параметр `handle` должен содержать дескриптор объекта, для которого функция `GetSecurityInfo` получает дескриптор безопасности.

В листинге 40.4 приведена программа, в которой функция `GetSecurityInfo` используется для получения идентификаторов безопасности владельца объекта и первичной группы владельца объекта по дескриптору объекта.

Программа 40.4. Получение идентификаторов безопасности владельца объекта и первичной группы по дескриптору объекта

```
#define _WIN32_WINNT 0x0500

#include <stdio.h>
#include <windows.h>
#include <lm.h>
#include <aclapi.h>
#include <sddl.h>

int main()
{
  char  chFileName[256];   // имя файла
  HANDLE  hFile;       // дескриптор файла

  PSID  pSidOwner;     // указатель на SID владельца объекта
  PSID  pSidGroup;     // указатель на SID первичной группы объекта
  PSECURITY_DESCRIPTOR  pSecurityDescriptor;  // указатель на SD

  LPTSTR  lpStringSid;   // указатель на строку SID
```

```
DWORD  dwRetCode;      // код возврата

// вводим полное имя файла, например: C:\\test.txt
printf("Input a full path to your file: ");
gets(chFileName);

// открываем файл
hFile = CreateFile(
  chFileName,        // имя файла
  READ_CONTROL,      // разрешаем доступ к дескриптору безопасности
  0,                 // не разделяем доступ
  NULL,              // ненаследуемый дескриптор
  OPEN_EXISTING,     // открываем существующий файл
  FILE_ATTRIBUTE_NORMAL,   // обычный файл
  NULL);             // шаблона нет
if(hFile == INVALID_HANDLE_VALUE)
{
  dwRetCode = GetLastError();
  printf("Create file failed.\n");
  printf("Error code: %u\n", dwRetCode);

  return dwRetCode;
}

// получаем дескриптор безопасности файла
dwRetCode = GetSecurityInfo(
  hFile,           // дескриптор файла
  SE_FILE_OBJECT,      // объект файл
  GROUP_SECURITY_INFORMATION |
  OWNER_SECURITY_INFORMATION,  // тип информации
  &pSidOwner,      // адрес указателя на SID владельца
  &pSidGroup,      // адрес указателя на первичную группу
  NULL,            // указатель на DACL не нужен
  NULL,            // указатель на SACL не нужен
  &pSecurityDescriptor);   // адрес указателя на SD
if (dwRetCode != ERROR_SUCCESS)
{
  printf("Get named security info failed.\n");
  printf("Error code: %u\n", dwRetCode);
```

```
    return dwRetCode;
  }

  // преобразуем SID владельца в строку
  if (!ConvertSidToStringSid(pSidOwner, &lpStringSid))
  {
    printf("Convert SID to string SID failed.");
    dwRetCode = GetLastError();

    return dwRetCode;
  }
  // печатаем SID владельца
  printf("Owner SID: %s\n", lpStringSid);
  // освобождаем память для строки
  LocalFree(lpStringSid);

  // преобразуем SID первичной группы в строку
  if (!ConvertSidToStringSid(pSidGroup, &lpStringSid))
  {
    printf("Convert SID to string SID failed.");
    dwRetCode = GetLastError();

    return dwRetCode;
  }
  // печатаем SID первичной группы
  printf("Group SID: %s\n", lpStringSid);
  // освобождаем память для строки
  LocalFree(lpStringSid);

  // освобождаем память для дескриптора
  LocalFree(pSecurityDescriptor);

  // закрываем файл
  CloseHandle(hFile);

  return 0;
}
```

40.6. Получение данных из дескриптора безопасности

Из дескриптора безопасности можно получить следующие данные: идентификаторы безопасности владельца объекта и первичной группы владельца объекта, а также списки управления доступом к объекту DACL и SACL. Для получения этих данных используются соответственно функции `GetSecurityDescriptorOwner`, `GetSecurityDescriptorGroup`, `GetSecurityDescriptorDacl` и `GetSecurityDescriptorSacl`. В этом разделе будут рассмотрены только первые две из перечисленных выше функций. Функции для получения из дескриптора безопасности информации о списках DACL и SACL будут рассмотрены в *гл. 41*.

Для получения из дескриптора безопасности информации о владельце объекта используется функция `GetSecurityDescriptorOwner`, которая имеет следующий прототип:

```
BOOL  GetSecurityDescriptorOwner(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,   // указатель на SD
  PSID    *pOwner,           // указатель на SID владельца объекта
  LPBOOL  lpbOwnerDefaulted  // указатель на флаг
);
```

В случае успешного завершения функция возвращает ненулевое значение, в противном случае — `FALSE`. При неудачном завершении функции код ошибки можно получить посредством вызова функции `GetLastError`. Параметры функции `GetSecurityDescriptorOwner` имеют следующее назначение.

Параметр `pSecurityDescriptor` должен указывать на структуру типа `SECURITY_DESCRIPTOR`, из которой определется владелец объекта.

Параметр `pOwner` должен содержать адрес переменной, в которую функция запишет указатель на идентификатор безопасности владельца объекта. Если дескриптор не содержит идентификатор безопасности владельца объекта, то функция устанавливает значение переменной в `NULL` и параметр `lpbOwnerDefaulted` игнорируется.

Параметр `lpbOwnerDefaulted` должен указывать на булеву переменную. Если в дескрипторе безопасности установлен флаг `SE_OWNER_DEFAULTED`, то функция установит значение этой булевой переменной в `TRUE`, в противном случае — в `FALSE`.

Для получения из дескриптора безопасности информации о первичной группе владельца объекта используется функция `GetSecurityDescriptorGroup`, которая имеет следующий прототип:

```
BOOL  GetSecurityDescriptorGroup(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,  // указатель на SD
```

```
    PSID    *pGroup,              // указатель на SID владельца
    LPBOOL  lpbGroupDefaulted     // указатель на флаг
  );
```

В случае успешного завершения функция возвращает ненулевое значение, в противном случае — FALSE. При неудачном завершении функции код ошибки можно получить посредством вызова функции GetLastError. Параметры функции GetSecurityDescriptorGroup имеют следующее назначение.

Параметр pSecurityDescriptor должен указывать на структуру типа SECURITY_DESCRIPTOR, из которой определяется первичная группа владельца объекта.

Параметр pGroup должен содержать адрес переменной, в которую функция запишет указатель на идентификатор безопасности первичной группы владельца объекта. Если дескриптор не содержит идентификатор безопасности первичной группы владельца объекта, то функция устанавливает значение переменной в NULL и параметр lpbGroupDefaulted игнорируется.

Параметр lpbGroupDefaulted должен указывать на булеву переменную. Если в дескрипторе безопасности установлен флаг SE_GROUP_DEFAULTED, то функция установит значение этой булевой переменной в TRUE, в противном случае — в FALSE.

В листинге 40.5 приведена программа, которая получает из дескриптора безопасности файла указатели на идентификаторы безопасности владельца этого файла и его первичной группы, используя для этого функции GetSecurityDescriptorOwner и GetSecurityDescriptorGroup. В связи с этой программой отметим, что функция GetFileSecurity вернет дескриптор безопасности файла только в том случае, если программа запущена от имени владельца файла.

Листинг 40.5. Получение идентификаторов безопасности владельца объекта и его первичной группы

```
#define _WIN32_WINNT 0x0500

#include <stdio.h>
#include <windows.h>
#include <sddl.h>

int main()
{
  char chFileName[256];     // имя файла

  PSID pSidOwner = NULL;    // указатель на SID владельца объекта
```

```
PSID pSidGroup = NULL;   // указатель на SID первичной группы объекта
PSECURITY_DESCRIPTOR pSD = NULL;  // указатель на SD
BOOL bOwnerDefaulted = FALSE;   // флаг владельца по умолчанию
BOOL bGroupDefaulted = FALSE;   // флаг первичной группы по умолчанию

LPTSTR lpStringSid;    // указатель на строку SID

DWORD dwLength = 0;    // длина дескриптора безопасности
DWORD dwRetCode;       // код возврата

// вводим имя файла, к которому вы имеете доступ
printf("Input a full path to your file or directory: ");
gets(chFileName);

// получаем длину дескриптора безопасности файла
if (!GetFileSecurity(
  chFileName,      // имя файла
  GROUP_SECURITY_INFORMATION |
  OWNER_SECURITY_INFORMATION,    // информация, которую нужно получить
  pSD,             // адрес буфера для дескриптора безопасности
  dwLength,        // длина буфера
  &dwLength))      // необходимая длина

  if (dwLength != 0)
    // захватываем память для дескриптора безопасности
    pSD = (PSECURITY_DESCRIPTOR) new char[dwLength];
  else
  {
    printf("Get file security for length failed.\n");
    dwRetCode = GetLastError();
    printf("Error code: %u\n", dwRetCode);

    return dwRetCode;
  }

// получаем дескриптор безопасности файла
if (!GetFileSecurity(
  chFileName,      // имя файла
  GROUP_SECURITY_INFORMATION |
```

```
  OWNER_SECURITY_INFORMATION,    // информация, которую нужно получить
  pSD,             // адрес буфера для дескриптора безопасности
  dwLength,        // длина буфера
  &dwLength))      // необходимая длина
{
  printf("Get file security failed.\n");
  dwRetCode = GetLastError();
  printf("Error code: %u\n", dwRetCode);

  return dwRetCode;
}

// получаем идентификатор безопасности владельца объекта
if (!GetSecurityDescriptorOwner(
  pSD,
  &pSidOwner,
  &bOwnerDefaulted))
{
  printf("Get security descriptor owner failed.\n");
  dwRetCode = GetLastError();
  printf("Error code: %u\n", dwRetCode);

  return dwRetCode;
}

// получаем SD первичной группы владельца объекта
if (!GetSecurityDescriptorGroup(
  pSD,
  &pSidGroup,
  &bGroupDefaulted))
{
  printf("Get security descriptor group failed.\n");
  dwRetCode = GetLastError();
  printf("Error code: %u\n", dwRetCode);

  return dwRetCode;
}

// преобразуем SID владельца в строку
```

```
  if (!ConvertSidToStringSid(pSidOwner, &lpStringSid))
  {
    printf("Convert SID to string SID failed.");
    dwRetCode = GetLastError();

    return dwRetCode;
  }
  // печатаем SID владельца
  printf("Owner SID: %s\n", lpStringSid);
  // освобождаем память для строки
  LocalFree(lpStringSid);

  // преобразуем SID первичной группы в строку
  if (!ConvertSidToStringSid(pSidGroup, &lpStringSid))
  {
    printf("Convert SID to string SID failed.");
    dwRetCode = GetLastError();

    return dwRetCode;
  }
  // печатаем SID первичной группы
  printf("Group SID: %s\n", lpStringSid);
  // освобождаем память для строки
  LocalFree(lpStringSid);

  // освобождаем память для дескриптора
  delete[] pSD;

  return 0;
}
```

В заключение этого раздела отметим, что функции `GetSecurityDescriptorOwner`, `GetSecurityDescriptorGroup`, `GetSecurityDescriptorDacl` и `GetSecurityDescriptorSacl` используются совместно с функциями, которые возвращают только дескриптор безопасности объекта без указателей на данные, хранящиеся в этом дескрипторе безопасности. К ним относятся сле-

дующие функции: GetFileSecurity, GetKernelObjectSecurity, GetUserObjectSecurity, QueryServiceObjectSecurity, RegGetKeySecurity. Как видно из названий этих функций, они предназначены для извлечения дескриптора безопасности из объектов определенного типа, а именно — файлов, объектов ядра, сервисов и ключей регистра соответственно. При этом заметим, что вместо всех этих функций могут использоваться функции GetNamedSecurityInfo и GetSecurityInfo, которые сразу возвращают дескриптор безопасности и указатели на данные, хранящиеся в нем. Небольшое ограничение существует для объектов ядра, т. е. функции GetNamedSecurityInfo и SetNamedSecurityInfo работают только со следующими объектами ядра: семафорами, событиями, мьютексами, ожидающими таймерами и отображениями файлов в памяти. Поэтому можно сказать, что функции для получения дескрипторов безопасности из объектов конкретного типа являются устаревшими.

40.7. Получение состояния управляющих флагов дескриптора безопасности

Для получения информации о состоянии управляющих флагов дескриптора безопасности используется функция GetSecurityDescriptorControl, которая имеет следующий прототип:

```
BOOL  GetSecurityDescriptorControl(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,  // указатель на SD
  PSECURITY_DESCRIPTOR_CONTROL  pControl,     // управляющие флаги SD
  LPDWORD    lpdwRevision                     // версия SD
);
```

В случае успешного завершения функция возвращает ненулевое значение, в противном случае — FALSE. При неудачном завершении функции код ошибки можно получить посредством вызова функции GetLastError. Параметры функции GetSecurityDescriptorControl имеют следующее назначение.

Параметр pSecurityDescriptor должен указывать на структуру типа SECURITY_DESCRIPTOR, из которой определяются состояния управляющих флагов.

Параметр pControl должен указывать на переменную типа SECURITY_DESCRIPTOR_CONTROL, в которую функция запишет состояния управляющих флагов. Тип SECURITY_DESCRIPTOR_CONTROL определяется следующим образом:

```
typedef WORD SECURITY_DESCRIPTOR_CONTROL;
```

То есть фактически является символическим именем типа WORD.

Параметр `lpdwRevision` должен указывать на переменную типа `DWORD`, в которую функция запишет версию дескриптора безопасности.

В листинге 40.6 приведена программа, которая получает управляющие флаги из дескриптора безопасности, используя для этого функцию `GetSecurityDescriptorControl`. После этого в программе проверяются состояния управляющих флагов, характеризующих список управления доступом DACL. Полный список управляющих флагов, которые могут использоваться в дескрипторе безопасности, приведен в *разд. 40.10*.

Листинг 40.6. Получение состояния управляющих флагов из дескриптора безопасности

```
#define _WIN32_WINNT 0x0500

#include <stdio.h>
#include <windows.h>
#include <aclapi.h>

int main()
{
  char chFileName[256];     // имя файла
  HANDLE hFile;             // дескриптор файла

  PSECURITY_DESCRIPTOR pSecurityDescriptor;  // указатель на SD

  SECURITY_DESCRIPTOR_CONTROL wControl;  // управляющие флаги из SD
  DWORD dwRevision;       // версия дескриптора безопасности

  DWORD dwRetCode;        // код возврата

  // вводим полное имя файла, например: C:\\test.txt
  printf("Input a full path to your file: ");
  gets(chFileName);

  // открываем файл
  hFile = CreateFile(
    chFileName,         // имя файла
    READ_CONTROL,       // разрешаем доступ к дескриптору безопасности
    0,                  // не разделяем доступ
    NULL,               // ненаследуемый дескриптор
```

```
  OPEN_EXISTING,     // открываем существующий файл
  FILE_ATTRIBUTE_NORMAL,   // обычный файл
  NULL);             // шаблона нет
if(hFile == INVALID_HANDLE_VALUE)
{
  dwRetCode = GetLastError();
  printf("Create file failed.\n");
  printf("Error code: %u\n", dwRetCode);

  return dwRetCode;
}

// получаем дескриптор безопасности файла
dwRetCode = GetSecurityInfo(
  hFile,          // дескриптор файла
  SE_FILE_OBJECT,    // объект файл
  DACL_SECURITY_INFORMATION,   // тип информации
  NULL,           // указатель на SID владельца не нужен
  NULL,           // указатель на первичную группу не нужен
  NULL,           // указатель на DACL не нужен
  NULL,           // указатель на SACL не нужен
  &pSecurityDescriptor);       // адрес указателя на SD
if (dwRetCode != ERROR_SUCCESS)
{
  printf("Get named security info failed.\n");
  printf("Error code: %u\n", dwRetCode);

  return dwRetCode;
}

// получаем управляющую информацию из дескриптора безопасности
if(!GetSecurityDescriptorControl(
  pSecurityDescriptor,
  &wControl,
  &dwRevision))
{
  dwRetCode = GetLastError();
  printf("Get security descriptor control failed.\n");
  printf("Error code: %u\n", dwRetCode);
```

```
    return dwRetCode;
  }

  printf("The following flags are set: \n");

  // определяем информацию из управляющего слова
  if(wControl & SE_DACL_AUTO_INHERITED)
    printf("SE_DACL_AUTO_INHERITED\n");
  if(wControl & SE_DACL_DEFAULTED)
    printf("SE_DACL_DEFAULTED\n");
  if(wControl & SE_DACL_PRESENT)
    printf("SE_DACL_PRESENT\n");
  if(wControl & SE_DACL_PROTECTED)
    printf("SE_DACL_PROTECTED\n");

  // выводим на печать версию дескриптора безопасности
  printf("Descriptor revision: %u\n", dwRevision);

  // освобождаем память для дескриптора
  LocalFree(pSecurityDescriptor);

  // закрываем файл
  CloseHandle(hFile);

  return 0;
}
```

40.8. Изменение дескриптора безопасности по имени объекта

Для изменения дескриптора безопасности по имени объекта используется функция `SetNamedSecurityInfo`, которая имеет следующий прототип:

```
DWORD  SetNamedSecurityInfo(
  LPSTR  pObjectName,    // указатель на имя объекта
  SE_OBJECT_TYPE  ObjectType,          // тип объекта
  SECURITY_INFORMATION  SecurityInfo,  // управляющие флаги
  PSID  psidOwner,       // указатель на SID владельца
  PSID  psidGroup,       // указатель на SID группы
```

```
    PACL  pDacl,           // указатель на список DACL
    PACL  pSacl            // указатель на список SACL
  );
```

В случае успешного завершения функция возвращает значение ERROR_SUCCESS, а в случае неудачи — код ошибки. Параметры функции SetNamedSecurityInfo имеют следующее назначение.

Параметр pObjectName должен указывать на имя объекта, дескриптор безопасности которого изменяет функция.

Параметр ObjectType должен содержать тип объекта, дескриптор безопасности которого изменяет функция. Тип объекта должен быть одним из значений перечисления SE_OBJECT_TYPE, которое имеет следующий тип

```
typedef enum  _SE_OBJECT_TYPE
{
  SE_UNKNOWN_OBJECT_TYPE = 0,  // неизвестный тип
  SE_FILE_OBJECT,         // файл
  SE_SERVICE,             // сервис
  SE_PRINTER,             // принтер
  SE_REGISTRY_KEY,        // ключ регистра
  SE_LMSHARE,             // разделяемый объект локальной сети
  SE_KERNEL_OBJECT,       // объект ядра
  SE_WINDOW_OBJECT,       // оконная станция на локальном компьютере
  SE_DS_OBJECT,           // сервис каталогов
  SE_DS_OBJECT_ALL,       // сервис каталогов с их свойствами
  SE_PROVIDER_DEFINED_OBJECT  // Windows Management Instrumentation
}  SE_OBJECT_TYPE;
```

Отметим, что последние три значения определены, начиная с версии Windows 2000 и старше.

Параметр SecurityInfo должен содержать переменную типа SECURITY_INFORMATION, в которой установлены флаги, отмечающие информацию, которую должна изменить функция в дескрипторе безопасности. Тип SECURITY_INFORMATION определен следующим образом:

```
typedef  DWORD  SECURITY_INFORMATION,  *PSECURITY_INFORMATION;
```

т. е. представляет собой двойное слово (тип DWORD). При работе с функцией SetNamedSecurityInfo в этом двойном слове может быть установлена любая комбинация следующих флагов:

- OWNER_SECURITY_INFORMATION — получить владельца объекта;
- GROUP_SECURITY_INFORMATION — получить первичную группу владельца;
- DACL_SECURITY_INFORMATION — получить список DACL;

- SACL_SECURITY_INFORMATION — получить список SACL;
- PROTECTED_DACL_SECURITY_INFORMATION — защита списка DACL от наследования;
- PROTECTED_SACL_SECURITY_INFORMATION — защита списка SACL от наследования.

Отметим, что последние два флага используются, только начиная с версии Windows 2000. Если установлен флаг DACL_SECURITY_INFORMATION, то флаг PROTECTED_DACL_SECURITY_INFORMATION игнорируется. В свою очередь, если установлен флаг SACL_SECURITY_INFORMATION, то игнорируется флаг PROTECTED_SACL_SECURITY_INFORMATION.

Параметр psidOwner должен указывать на идентификатор безопасности нового владельца объекта. Причем процесс, в котором выполняется функция, должен иметь режим доступа WRITE_OWNER к объекту, владельца которого он изменяет. Или новый владелец должен иметь привилегию SE_TAKE_OWNERSHIP_NAME, которая позволяет стать новым владельцем объекта без разрешения старого владельца объекта. Функция будет изменять владельца объекта только в том случае, если в параметре SecurityInfo установлен флаг OWNER_SECURITY_INFORMATION. Если изменять владельца объекта не нужно, то значение этого параметра может быть установлено в NULL.

Параметр psidGroup должен указывать на идентификатор безопасности новой первичной группы владельца объекта. Функция изменит первичную группу владельца объекта только в том случае, если в параметре SecurityInfo установлен флаг GROUP_SECURITY_INFORMATION. Если изменять первичную группу владельца объекта не нужно, то значение этого параметра может быть установлено в NULL.

Параметр pDacl должен указывать на новый список DACL объекта. Причем процесс, в котором выполняется функция, должен иметь режим доступа WRITE_DAC к объекту, список DACL которого он заменяет. Или процесс должен исполняться от имени владельца объекта. Новый список DACL будет записан только в том случае, если в параметре SecurityInfo установлен флаг DACL_SECURITY_INFORMATION. Если заменять список DACL объекта не нужно, то значение этого параметра может быть установлено в NULL.

Параметр pSacl должен указывать на новый список SACL объекта. Причем пользователь, от имени которого выполняется процесс, должен иметь привилегию SE_SECURITY_NAME. Новый список SACL будет записан только в том случае, если в параметре SecurityInfo установлен флаг SACL_SECURITY_INFORMATION. Если заменять список SACL объекта не нужно, то значение этого параметра может быть установлено в NULL.

В листинге 40.7 приведена программа, в которой функция SetNamedSecurityInfo используется для изменения владельца файла.

Программа 40.7. Изменение владельца файла по имени файла

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <windows.h>
#include <lm.h>
#include <aclapi.h>

int main()
{
  wchar_t wchFileName[256];      // имя файла
  wchar_t wchUserName[UNLEN];    // имя нового владельца файла

  PSID  lpSID = NULL;            // указатель на SID
  LPTSTR  lpDomainName = NULL;   // указатель на имя домена

  DWORD  dwLengthOfSID = 0;      // длина SID
  DWORD  dwLengthOfDomainName = 0;   // длина имени домена

  DWORD dwRetCode;               // код возврата

  SID_NAME_USE typeOfSID;        // тип учетной записи

  // вводим имя файла, например: C:\\test.txt
  printf("Input a full path to your file: ");
  _getws(wchFileName);

  // вводим имя нового владельца объекта
  printf("Input a user name: ");
  wscanf(L"%s", wchUserName);

  // определяем длину SID нового владельца файла
  LookupAccountName(
    NULL,              // ищем имя на локальном компьютере
    wchUserName,       // имя нового владельца файла
```

```
  NULL,              // определяем длину SID
  &dwLengthOfSID,    // длина SID
  NULL,              // определяем имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &typeOfSID);       // тип учетной записи

// проверяем, вернула ли функция длину SID
if (dwLengthOfSID == 0)
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// распределяем память для SID и имени домена
lpSID = (PSID) new wchar_t[dwLengthOfSID];
lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];

// определяем SID и имя домена пользователя
if(!LookupAccountName(
  NULL,              // ищем имя на локальном компьютере
  wchUserName,       // имя пользователя
  lpSID,             // указатель на SID
  &dwLengthOfSID,    // длина SID
  lpDomainName,      // указатель на имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &typeOfSID))       // тип учетной записи
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}
```

```
  // устанавливаем нового владельца файла
  dwRetCode = SetNamedSecurityInfo(
    wchFileName,       // имя файла
    SE_FILE_OBJECT,     // объект файл
    OWNER_SECURITY_INFORMATION,  // изменяем только имя владельца файла
    lpSID,          // адрес на SID нового владельца
    NULL,           // первичную группу не изменяем
    NULL,           // DACL не изменяем
    NULL);          // SACL не изменяем
  if (dwRetCode != ERROR_SUCCESS)
  {
    printf("Set named security info failed.\n");
    printf("Error code: %u\n", dwRetCode);
    return dwRetCode;
  }
  printf("The new owner of the file is set.\n");

  // освобождаем память
  delete[] lpSID;
  delete[] lpDomainName;
  return 0;
}
```

В заключение этого раздела отметим, что в данной программе функция `SetNamedSecurityInfo` использовалась только для изменения имени владельца файла. Если же функция `SetNamedSecurityInfo` изменяет также списки управления доступом к охраняемому объекту, то в наследуемые элементы этих списков включаются также списки управления дочерних объектов. Более подробно о наследовании элементов списков управления доступом сказано в *разд. 36.7*, *36.9*.

40.9. Изменение дескриптора безопасности по дескриптору объекта

Для изменения дескриптора безопасности по дескриптору объекта используется функция `SetSecurityInfo`, которая имеет следующий прототип:

```
  DWORD  SetSecurityInfo(
    HANDLE  handle,      // дескриптор объекта
    SE_OBJECT_TYPE  ObjectType,          // тип объекта
```

```
    SECURITY_INFORMATION  SecurityInfo,  // управляющие флаги
    PSID  psidOwner,      // указатель на SID владельца
    PSID  psidGroup,      // указатель на SID первичной группы
    PACL  pDacl,          // указатель на список DACL
    PACL  pSacl           // указатель на список SACL
  );
```

В случае успешного завершения функция возвращает значение `ERROR_SUCCESS`, а в случае неудачи — код ошибки. Все параметры этой функции за исключением первого имеют то же назначение, что и соответствующие параметры функции `SetNamedSecurityInfo`. Первый же параметр `handle` должен содержать дескриптор объекта, в котором функция `SetSecurityInfo` изменяет дескриптор безопасности.

В листинге 40.8 приведена программа, в которой функция `SetSecurityInfo` используется для изменения владельца файла.

Листинг 40.8. Изменение владельца файла по дескриптору файла

```
#ifndef UNICODE
#define UNICODE
#endif

#include <stdio.h>
#include <windows.h>
#include <lm.h>
#include <aclapi.h>

int main()
{
  wchar_t wchFileName[256];    // имя файла
  wchar_t wchUserName[UNLEN];  // имя нового владельца файла

  HANDLE hFile;                // дескритпор файла

  PSID lpSID = NULL;           // указатель на SID
  LPTSTR lpDomainName = NULL;  // указатель на имя домена

  DWORD dwLengthOfSID = 0;     // длина SID
  DWORD dwLengthOfDomainName = 0;  // длина имени домена

  DWORD dwRetCode;     // код возврата
```

```
SID_NAME_USE typeOfSID;      // тип учетной записи

// вводим имя файла, например: C:\\test.txt
printf("Input a full path to your file: ");
_getws(wchFileName);

// вводим имя нового владельца объекта
printf("Input a user name: ");
wscanf(L"%s", wchUserName);

// открываем файл
hFile = CreateFile(
  wchFileName,         // имя файла
  WRITE_OWNER,         // разрешаем изменение владельца файла
  0,                   // не разделяем доступ
  NULL,                // ненаследуемый дескриптор
  OPEN_EXISTING,       // открываем существующий файл
  FILE_ATTRIBUTE_NORMAL,   // обычный файл
  NULL);               // шаблона нет
if(hFile == INVALID_HANDLE_VALUE)
{
  dwRetCode = GetLastError();
  printf("Create file failed.\n");
  printf("Error code: %u\n", dwRetCode);

  return dwRetCode;
}

// определяем длину SID нового владельца файла
LookupAccountName(
  NULL,              // ищем имя на локальном компьютере
  wchUserName,       // имя нового владельца файла
  NULL,              // определяем длину SID
  &dwLengthOfSID,    // длина SID
  NULL,              // определяем имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &typeOfSID);       // тип учетной записи

// проверяем, вернула ли функция длину SID
```

```
if (dwLengthOfSID == 0)
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// распределяем память для SID и имени домена
lpSID = (PSID) new wchar_t[dwLengthOfSID];
lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];

// определяем SID и имя домена пользователя
if(!LookupAccountName(
  NULL,              // ищем имя на локальном компьютере
  wchUserName,       // имя пользователя
  lpSID,             // указатель на SID
  &dwLengthOfSID,    // длина SID
  lpDomainName,      // указатель на имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &typeOfSID))       // тип учетной записи
{
  dwRetCode = GetLastError();
  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);
  return dwRetCode;
}
// устанавливаем нового владельца файла
dwRetCode = SetSecurityInfo(
  hFile,           // дескриптор файла
  SE_FILE_OBJECT,  // объект файл
  OWNER_SECURITY_INFORMATION,  // изменяем только имя владельца файла
  lpSID,           // адрес на SID нового владельца
  NULL,            // первичную группу не изменяем
  NULL,            // DACL не изменяем
  NULL);           // SACL не изменяем
if (dwRetCode != ERROR_SUCCESS)
```

```
  {
    printf("Set named security info failed.\n");
    printf("Error code: %u\n", dwRetCode);

    return dwRetCode;
  }
  printf("The new owner of the file is set.\n");

  // освобождаем память
  delete[] lpSID;
  delete[] lpDomainName;
  // закрываем файл
  CloseHandle(hFile);
  return 0;
}
```

В заключение этого раздела сделаем следующее замечание. В данной программе функция `SetSecurityInfo` использовалась только для изменения имени владельца файла. Если же эта функция изменяет также списки управления доступом к охраняемому объекту, то в наследуемых элементы этих списков включаются также списки управления дочерних объектов. Более подробно о наследовании элементов списков управления доступом сказано в *разд. 36.7*, *36.9*.

40.10. Изменение состояния управляющих флагов дескриптора безопасности

Для изменения состояния управляющих флагов дескриптора безопасности используется функция `SetSecurityDescriptorControl`, которая имеет следующий прототип:

```
BOOL  SetSecurityDescriptorControl(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,    // указатель на SD
  SECURITY_DESCRIPTOR_CONTROL  ControlBitsOfInterest,  // флаги
  SECURITY_DESCRIPTOR_CONTROL  ControlBitsToSet        // новые флаги
  );
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного завершения функции можно получить посредством вызова функции `GetLastError`. Параметры функции `SetSecurityDescriptorControl` имеют следующее назначение.

Параметр pSecurityDescriptor должен указывать на дескриптор безопасности, состояния управляющих флагов которого будут изменяться.

В параметре ControlBitsOfInterest должны быть установлены управляющие флаги, значения которых будут изменяться.

В параметре ControlBitsToSet должны быть установлены новые значения управляющих флагов, которые будут изменяться.

Два последних параметра имеют тип SECURITY_DESCRIPTOR_CONTROL, который определяется следующим образом:

```
typedef  WORD  SECURITY_DESCRIPTOR_CONTROL;
```

т. е. фактически является символическим именем типа WORD, описывающего слово в памяти компьютера. При использовании функции SetSecurityDescriptorControl в этом слове могут устанавливаться следующие управляющие флаги:

- SE_DACL_AUTO_INHERIT_REQ — выполнить механизм наследования DACL;
- SE_SACL_AUTO_INHERIT_REQ — выполнить механизм наследования SACL;
- SE_DACL_AUTO_INHERITED — список DACL установлен механизмом наследования;
- SE_SACL_AUTO_INHERITED — список SACL установлен механизмом наследования;
- SE_DACL_PROTECTED — список DACL защищен от наследования;
- SE_SACL_PROTECTED — список SACL защищен от наследования.

В листинге 40.9 приведена программа, в которой устанавливается флаг защиты дескриптора безопасности от механизма наследования списка DACL, используя функцию SetSecurityDescriptorControl.

Листинг 40.9. Изменение флага защиты от наследования списка DACL

```
#define _WIN32_WINNT 0x0500

#ifndef PROTECTED_DACL_SECURITY_INFORMATION
#define PROTECTED_DACL_SECURITY_INFORMATION  0x80000000L
#endif

#include <stdio.h>
#include <windows.h>
#include <lm.h>
#include <aclapi.h>
#include <sddl.h>
```

```
int main()
{
  char chFileName[256];    // имя файла
  HANDLE hFile;            // дескриптор файла

  PSECURITY_DESCRIPTOR pSecurityDescriptor;  // указатель на SD
  PACL pDacl;          // указатель на DACL

  DWORD dwRetCode;    // код возврата

  // вводим полное имя файла, например: C:\\test.txt
  printf("Input a full path to your file: ");
  gets(chFileName);

  // открываем файл
  hFile = CreateFile(
    chFileName,      // имя файла
    READ_CONTROL | WRITE_DAC,  // доступ к дескриптору безопасности
    0,               // не разделяем доступ
    NULL,            // ненаследуемый дескриптор
    OPEN_EXISTING,   // открываем существующий файл
    FILE_ATTRIBUTE_NORMAL,   // обычный файл
    NULL);           // шаблона нет
  if(hFile == INVALID_HANDLE_VALUE)
  {
    dwRetCode = GetLastError();
    printf("Create file failed.\n");
    printf("Error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  // получаем дескриптор безопасности файла
  dwRetCode = GetSecurityInfo(
    hFile,           // дескриптор файла
    SE_FILE_OBJECT,  // объект файл
    DACL_SECURITY_INFORMATION,   // тип информации
    NULL,            // указатель на SID владельца не нужен
    NULL,            // указатель на первичную группу не нужен
```

```
  &pDacl,          // указатель на DACL
  NULL,            // указатель на SACL не нужен
  &pSecurityDescriptor);   // адрес указателя на SD
if (dwRetCode != ERROR_SUCCESS)
{
  printf("Get named security info failed.\n");
  printf("Error code: %u\n", dwRetCode);

  return dwRetCode;
}

// сбрасываем флаг автоматического наследования DACL
if(!SetSecurityDescriptorControl(
  pSecurityDescriptor,
  SE_DACL_PROTECTED,
  0))
{
  dwRetCode = GetLastError();
  printf("Set security descriptor control failed.");
  printf("Error code: %u\n", dwRetCode);

  return dwRetCode;
}

// устанавливаем новые свойства DACL
dwRetCode = SetSecurityInfo(
  hFile,         // дескриптор файла
  SE_FILE_OBJECT,    // объект файл
  DACL_SECURITY_INFORMATION |
  PROTECTED_DACL_SECURITY_INFORMATION,   // изменяем информацию о DACL
  NULL,          // владельца не изменяем
  NULL,          // первичную группу не изменяем
  pDacl,         // DACL изменяем
  NULL);         // SACL не изменяем
if (dwRetCode != ERROR_SUCCESS)
{
  printf("Set named security info failed.\n");
  printf("Error code: %u\n", dwRetCode);
```

```
    return dwRetCode;
  }

  // освобождаем память для дескриптора
  LocalFree(pSecurityDescriptor);

  printf("The flag SE_DACL_PROTECTED is reset.\n");
  // закрываем файл
  CloseHandle(hFile);
  return 0;
}
```

40.11. Строковое представление дескрипторов безопасности

Дескриптор безопасности может быть преобразован в строку, которая имеет специальный формат. Формат этой строки описывается при помощи специального языка, который называется SDDL (Security Descriptor Definition Language) — язык описания дескриптора безопасности. Далее приведено краткое описание этой нотации.

Строка, описывающая дескриптор безопасности, может быть разбита на четыре блока, которые последовательно описывают владельца объекта, первичную группу владельца объекта, списки SACL и DACL по аналогии со структурой `SECURITY_DESCRIPTOR`. Эти описания имеют следующие форматы:

- `O: SID_владельца_объекта`;
- `G: SID_первичной_группы_владельца_объекта`;
- `D: флаги_DACL(элемент_DACL) ... (элемент_DACL)`;
- `S: флаги_SACL(элемент_SACL) ... (элемент_SACL)`.

Сначала кратко рассмотрим форматы каждого из этих описаний.

Описание владельца объекта начинается с символов `O:`, за которыми следует символьное представление идентификатора безопасности владельца объекта.

Описание первичной группы владельца объекта начинается с символов `G:`, за которыми следует символьное представление идентификатора безопасности первичной группы владельца объекта.

Описание списка DACL начинается с символов `D:`, за которыми следуют символьные обозначения флагов, а затем перечисляются элементы списка DACL.

Описание списка SACL начинается с символов `S:`, за которыми следуют символьные обозначения флагов, а затем перечисляются элементы списка SACL.

Теперь более подробно опишем форматы подстрок, входящих в каждый блок строкового представления дескриптора безопасности.

Символьное представление идентификаторов безопасности

Напомним, что символьное представление идентификаторов безопасности рассматривалось в *разд. 36.5*. Эти же символьные представления используются в двух первых блоках строки для описания идентификаторов безопасности владельца объекта и первичной группы владельца объекта. В языке SDDL определены следующие символьные обозначения для общеизвестных идентификаторов безопасности:

- `AO` — `SDDL_ACCOUNT_OPERATORS` — операторы учетных записей;
- `AO` — `SDDL_ANONYMOUS` — анонимный пользователь;
- `AU` — `SDDL_AUTHENTICATED_USERS` — аутентифицированные пользователи;
- `BA` — `SDDL_BUILTIN_ADMINISTRATORS` — встроенные локальные администраторы;
- `BG` — `SDDL_BUILTIN_GUESTS` — встроенные локальные гости;
- `BO` — `SDDL_BACKUP_OPERATORS` — операторы резервных копий;
- `BU` — `SDDL_BUILTIN_USERS` — встроенные пользователи;
- `CA` — `_CERT_SERV_ADMINISTRATORS` — сертифицированные администраторы сервера;
- `CG` — `SDDL_CREATOR_GROUP` — создатель группы;
- `CO` — `SDDL_CREATOR_OWNER` — создатель владельца;
- `DA` — `SDDL_DOMAIN_ADMINISTRATORS` — администраторы домена;
- `DC` — `SDDL_DOMAIN_COMPUTERS` — компьютеры домена;
- `DD` — `SDDL_DOMAIN_DOMAIN_CONTROLLERS` — контроллеры домена;
- `DG` — `SDDL_DOMAIN_GUESTS` — гости домена;
- `DU` — `SDDL_DOMAIN_USERS` — пользователи домена;
- `EA` — `SDDL_ENTERPRISE_ADMINS` — администраторы предприятия;
- `ED` — `SDDL_ENTERPRISE_DOMAIN_CONTROLLERS` — контроллеры домена предприятия;
- `IU` — `SDDL_INTERACTIVE` — интерактивные пользователи;
- `LA` — `SDDL_LOCAL_ADMIN` — локальные администраторы;
- `LG` — `SDDL_LOCAL_GUEST` — локальный гость;
- `LS` — `SDDL_LOCAL_SERVICE` — локальный сервис;

- NO — SDDL_NETWORK_CONFIGURATION_OPS — оператор конфигурации сети;
- NS — SDDL_NETWORK_SERVICE — сетевой сервис;
- NU — SDDL_NETWORK — сетевой пользователь;
- PA — SDDL_GROUP_POLICY_ADMINS — администратор политик групп;
- PO — SDDL_PRINTER_OPERATORS — операторы принтеров;
- PS — SDDL_PERSONAL_SELF — персональный;
- PU — SDDL_POWER_USERS — полномочные пользователи;
- RC — SDDL_RESTRICTED_CODE — ограниченный код;
- RD — SDDL_REMOTE_DESKTOP — удаленный пользователь;
- RE — SDDL_REPLICATOR — репликатор;
- RS — SDDL_RAS_SERVERS — сервер удаленных запросов;
- RU — SDDL_ALIAS_PREW2KCOMPACC — для совместимости с Windows 2000;
- SA — SDDL_SCHEMA_ADMINISTRATORS — администраторы схемы;
- SO — SDDL_SERVER_OPERATORS — операторы сервера;
- SU — SDDL_SERVICE — сервис;
- SY — SDDL_LOCAL_SYSTEM — локальная система;
- WD — SDDL_EVERYONE — любой пользователь (World).

Символьное представление флагов

Флаги, управляющие списком DACL, имеют следующие символьные обозначения:

- P — SDDL_PROTECTED — установлен флаг SE_DACL_PROTECTED;
- AR — SDDL_AUTO_INHERIT_REQ — установлен флаг SE_DACL_AUTO_INHERIT_REQ;
- AI — SDDL_AUTO_INHERITED — установлен флаг SE_DACL_AUTO_INHERITED.

Такие же символьные обозначения имеют и флаги, управляющие списком SACL:

- P — SDDL_PROTECTED — установлен флаг SE_SACL_PROTECTED;
- AR — SDDL_AUTO_INHERIT_REQ — установлен флаг SE_SACL_AUTO_INHERIT_REQ;
- AI — SDDL_AUTO_INHERITED — установлен флаг SE_SACL_AUTO_INHERITED.

Заметим, что более подробно эти флаги рассматривались в предыдущем разделе.

Символьное представление элементов списков

Элементы списков управления доступом имеют следующее символьное представление:

```
тип_элемента; флаги; права; GUID_объекта; GUID_объекта_предка;
SID_учетной_записи
```

Здесь GUID является аббревиатурой для globally unique identifier, что переводится как "глобальный уникальный идентификатор". Такие идентификаторы используются для идентификации объектов. Теперь рассмотрим обозначения, используемые в языке SDDL для описания каждой подстроки из строки, описывающей элемент списка.

Подстрока тип_элемента описывает тип элемента списка DACL или SACL. Для типа элемента списка в языке SDDL используются следующие символьные обозначения:

- A — SDDL_ACCESS_ALLOWED — доступ разрешен;
- D — SDDL_ACCESS_DENIED — доступ запрещен;
- AU — SDDL_AUDIT — аудит;
- AL — SDDL_ALARM — тревога.

Подстрока флаги описывает управляющие флаги, установленные в элементе списка управления доступом. В языке SDDL управляющие флаги обозначаются следующим образом:

- CI — SDDL_CONTAINER_INHERIT — наследуется контейнером;
- OI — SDDL_OBJECT_INHERIT — наследуется не контейнером;
- NP — SDDL_NO_PROPAGATE — наследование не распространять;
- IO — SDDL_INHERIT_ONLY — только наследование, управления доступом нет;
- ID — SDDL_INHERITED — наследуемый элемент;
- SA — SDDL_AUDIT_SUCCESS — успешный аудит;
- FA — DDL_AUDIT_FAILURE — неудачный аудит.

Подстрока флаги представляет собой набор вышеперечисленных символьных обозначений для управляющих флагов.

Подстрока права описывает права доступа к объекту, которые контролирует данный элемент списка управления доступом. В языке SDDL права доступа обозначаются следующим образом.

Родовые права доступа:

- GA — SDDL_GENERIC_ALL — чтение, запись и исполнение;
- GR — SDDL_GENERIC_READ — только чтение;
- GW — SDDL_GENERIC_WRITE — только запись;
- GX — SDDL_GENERIC_EXECUTE — только исполнение.

Стандартные права доступа:

- RC — SDDL_READ_CONTROL — чтение управляющих флагов;
- WD — SDDL_WRITE_DAC — запись в список DACL;

- WO — SDDL_WRITE_OWNER — запись владельца;
- SD — SDDL_STANDARD_DELETE — стандартное удаление.

Права доступа к файлу:

- FA — SDDL_FILE_ALL — полный доступ к файлу;
- "FR — SDDL_FILE_READ — чтение файла;
- FW — SDDL_FILE_WRITE — запись в файл;
- FX — SDDL_FILE_EXECUTE — исполнение файла.

Права доступа к ключу регистра:

- KA — SDDL_KEY_ALL — полный доступ к ключу регистра;
- KP — SDDL_KEY_READ — чтение ключа регистра;
- KW — SDDL_KEY_WRITE — запись в ключ регистра;
- KX — SDDL_KEY_EXECUTE — исполнение ключа регистра

Глобальные уникальные идентификаторы объекта и объекта предка представляют собой символьные строки для обозначения 128-битовых уникальных значений. Такая строка имеет следующий формат:

$X_0X_1X_2X_3X_4X_5X_6X_7–X_8X_9X_{10}X_{11}–X_{12}X_{13}X_{14}X_{15}–X_{16}X_{17}X_{18}X_{19}–X_{20}X_{21}X_{22}X_{23}X_{24}X_{25}X_{26}X_{27}X_{28}X_{29}X_{30}X_{31}$

Символ X обозначает шестнадцатеричную цифру.

Подстрока SID_учетной_записи описывает идентификатор безопасности учетной записи, для которой элемент списка контролирует доступ к объекту. Формат этой подстроки был описан в *разд 36.5*, посвященном идентификаторам безопасности. Символьные обозначения для общеизвестных идентификаторов безопасности, которые используются в языке SDDL, были рассмотрены также в *разд. 36.5*.

Теперь перейдем к рассмотрению функций, которые используются для преобразования дескриптора безопасности в строковое представление и наоборот. Для преобразования дескриптора безопасности в строку используется функция ConvertSecurityDescriptorToStringSecurityDescriptor, которая имеет следующий прототип:

```
BOOL  ConvertSecurityDescriptorToStringSecurityDescriptor(
  PSECURITY_DESCRIPTOR  SecurityDescriptor,     // адрес SD
  DWORD  RequestedStringSDRevision,             // версия
  SECURITY_INFORMATION  SecurityInformation,    // тип информации
  LPSTR  *StringSecurityDescriptor,             // строка SD
  PULONG  StringSecurityDescriptorLen           // длина строки
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — FALSE. При неудаче код ошибки можно получить при

помощи функции GetLastError, которая может вернуть одно из следующих значений:

- ERROR_INVALID_PARAMETER — неправильный параметр;
- ERROR_UNKNOWN_REVISION — неизвестная версия дескриптора безопасности;
- ERROR_NONE_MAPPED — неверный SID в дескрипторе безопасности;
- ERROR_INVALID_ACL — установлен флаг SE_DACL_PRESENT, но самого списка DACL нет.

Параметры функции ConvertSecurityDescriptorToStringSecurityDescriptor имеют следующее назначение.

Параметр SecurityDescriptor должен указывать на дескриптор безопасности, который может находиться как в абсолютном формате, так и в относительном формате.

Параметр RequestedStringSDRevision должен содержать версию формата выходной строки, в которую будет преобразован дескриптор безопасности. В настоящее время в этом параметре должно быть установлено значение SDDL_REVISION_1.

В параметре SecurityInformation должна быть установлена любая комбинация следующих флагов:

- OWNER_SECURITY_INFORMATION — включить в строку информацию о владельце;
- GROUP_SECURITY_INFORMATION — включить в строку информацию о первичной группе;
- DACL_SECURITY_INFORMATION — включить в строку информацию о списке DACL;
- SACL_SECURITY_INFORMATION — включить в строку информацию о списке SACL.

Флаги указывают на то, какую информацию из дескриптора безопасности нужно включить в его строковое представление.

Параметр StringSecurityDescriptor должен содержать адрес указателя на строку. В этот указатель функция поместит адрес строки, в которую был преобразован дескриптор безопасности. После завершения работы со строкой программа должна освободить занимаемую этой строкой память при помощи функции LocalFree.

Параметр StringSecurityDescriptorLen должен указывать на переменную, в которую функция запишет длину памяти, которую занимает строковое представление дескриптора безопасности. Если эта длина не нужна, то в этом параметре может быть установлено значение NULL.

В листинге 40.10 приведена программа, которая выводит на консоль строку, описывающую дескриптор безопасности файла, используя для получения этой строки функцию `ConvertSecurityDescriptorToStringSecurityDescriptor`.

Листинг 40.10. Преобразование дескриптора безопасности в строковое представление

```
#define _WIN32_WINNT 0x0500

#include <stdio.h>
#include <windows.h>
#include <lm.h>
#include <aclapi.h>
#include <sddl.h>

int main()
{
  char chFileName[256];      // имя файла

  PSECURITY_DESCRIPTOR pSecurityDescriptor;    // указатель на SD
  LPSTR StringSecurityDescriptor;              // строка с SD
  DWORD StringSecurityDescriptorLen;           // длина строки с SD

  DWORD dwRetCode;        // код возврата

  // вводим имя файла, например: C:\\test.txt
  printf("Input a full path to your file: ");
  gets(chFileName);

  // получаем дескриптор безопасности файла
  dwRetCode = GetNamedSecurityInfo(
    chFileName,        // имя файла
    SE_FILE_OBJECT,    // объект файл
    DACL_SECURITY_INFORMATION,  // читаем список DACL
    NULL,        // указатель на владельца не нужен
    NULL,        // указатель на первичную группу не нужен
    NULL,        // указатель на DACL не нужен
    NULL,        // указатель на SACL не нужен
    &pSecurityDescriptor);   // адрес указателя на SD
```

```
  if (dwRetCode != ERROR_SUCCESS)
  {
    printf("Get named security info failed.\n");
    printf("Error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  // преобразуем дескриптор безопасности в строку
  if(!ConvertSecurityDescriptorToStringSecurityDescriptor(
    pSecurityDescriptor,       // адрес дескриптора безопасности
    SDDL_REVISION_1,           // версия языка описания
    OWNER_SECURITY_INFORMATION | GROUP_SECURITY_INFORMATION |
    DACL_SECURITY_INFORMATION | SACL_SECURITY_INFORMATION,
    &StringSecurityDescriptor,       // буфер для строки
    &StringSecurityDescriptorLen))   // длина буфера
  {
    dwRetCode = GetLastError();
    printf("Convert security descriptor to string security descriptor
    failed.");
    printf("Error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  printf("String security descriptor length: %u\n",
  StringSecurityDescriptorLen);
  printf("String security desriptor: %s\n", StringSecurityDescriptor);

  // освобождаем память
  LocalFree(pSecurityDescriptor);
  LocalFree(StringSecurityDescriptor);

  return 0;
}
```

Для преобразования дескриптора безопасности из строкового представления в бинарное представление используется функция ConvertStringSecurityDescriptorToSecurityDescriptor, которая имеет следующий прототип:

```
BOOL  ConvertStringSecurityDescriptorToSecurityDescriptor(
  LPCSTR  StringSecurityDescriptor,  // строка с SD
  DWORD  StringSDRevision,           // версия дескриптора безопасноти
  PSECURITY_DESCRIPTOR  *SecurityDescriptor,   // адрес буфера для SD
  PULONG  SecurityDescriptorSize     // длина буфера
);
```

В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — FALSE. При неудаче код ошибки можно получить при помощи функции GetLastError, которая может вернуть одно из следующих значений:

- ERROR_INVALID_PARAMETER — неправильный параметр;
- ERROR_UNKNOWN_REVISION — неизвестная версия дескриптора безопасности;
- ERROR_NONE_MAPPED — неверный SID в дескрипторе безопасности.

Параметры функции ConvertStringSecurityDescriptorToSecurityDescriptor имеют следующее назначение.

Параметр StringSecurityDescriptor должен указывать на строку с описанием дескриптора безопасности в формате, приведенном в начале раздела.

Параметр StringSDRevision должен содержать версию формата входной строки, из которой будет получено бинарное представление дескриптора безопасности. В настоящее время в этом параметре должно быть установлено значение SDDL_REVISION_1.

Параметр SecurityDescriptor должен содержать адрес указателя на структуру типа SECURITY_DESCRIPTOR. В этот указатель функция поместит адрес дескриптора безопасности, в который была преобразована строка. Дескриптор безопасности будет создан в относительном формате. После завершения работы с дескриптором безопасности программа должна освободить занимаемую дескриптором безопасности память при помощи функции LocalFree.

Параметр SecurityDescriptorSize должен указывать на переменную, в которую функция запишет длину памяти, которую занимает дескриптор безопасности. Если эта длина не нужна, то в этом параметре может быть установлено значение NULL.

Глава 41

Работа со списками управления доступом на высоком уровне

В этой главе будут рассмотрены функции для работы со списками управления доступом на высоком уровне. То есть доступ к элементам списка будет осуществляться при помощи специальных структур TRUSTEE и EXPLICIT_ACCESS, которые содержат информацию об элементах этих списков. А именно, структура типа TRUSTEE используется для идентификации учетных записей, которым соответствуют элементы списка управления доступом. А структуры типа EXPLICIT_ACCESS используются для доступа к самим элементам списка управления доступом. При этом структура типа EXPLICIT_ACCESS содержит поле, которое имеет тип TRUSTEE. Работа со списками управления доступом на низком уровне будет рассмотрена в *гл. 44*.

41.1. Структура *TRUSTEE*

Для идентификации учетных записей, которым соответствуют элементы из списка управления доступом, используется структура типа TRUSTEE. Тип этой структуры определен следующим образом:

```
typedef struct  _TRUSTEE {
  struct  _TRUSTEE  *pMultipleTrustee;  // не поддерживается
  MULTIPLE_TRUSTEE_OPERATION  MultipleTrusteeOperation; // замещение
  TRUSTEE_FORM  TrusteeForm;            // тип имени учетной записи
  TRUSTEE_TYPE  TrusteeType;            // тип учетной записи
  LPSTR      ptstrName;    // указатель на имя или SID учетной записи
} TRUSTEE_A, *PTRUSTEE_A, TRUSTEEA, *PTRUSTEEA;
```

Поле pMultipleTrustee должно указывать на структуру TRUSTEE, описывающую учетную запись сервера, которая может замещать (impersonates) данную структуру TRUSTEE. В настоящее время эта возможность не поддерживается и поэтому в поле pMultipleTrustee должно быть установлено значение NULL.

Поле MultipleTrusteeOperation должно содержать одно из значений перечисления типа:

```
typedef enum  _MULTIPLE_TRUSTEE_OPERATION {
  // учетная запись не замещена учетной записью сервера
  NO_MULTIPLE_TRUSTEE,
  // учетная запись замещена учетной записью сервера
  TRUSTEE_IS_IMPERSONATE,
}  MULTIPLE_TRUSTEE_OPERATION;
```

Если в этом поле установлено значение TRUSTEE_IS_IMPERSONATE, то поле pMultipleTrustee должно содержать информацию о структуре TRUSTEE сервера, которая замещает (impersonates) данную структуру TRUSTEE. Так как в настоящее время замещение структур TRUSTEE не поддерживается, то в этом поле должно быть установлено значение NO_MULTIPLE_TRUSTEE.

Поле TrusteeForm должно содержать одно из значений перечисления типа:

```
typedef enum  _TRUSTEE_FORM {
  // указатель ptstrName содержит адрес SID учетной записи
  TRUSTEE_IS_SID,
  // указатель ptstrName содержит адрес имени учетной записи
  TRUSTEE_IS_NAME,
  TRUSTEE_BAD_FORM     // недействительная форма
}  TRUSTEE_FORM;
```

Оно определяет тип информации, на которую указывает указатель ptstrName.

Поле TrusteeType должно содержать одно из значений перечисления типа:

```
typedef enum  _TRUSTEE_TYPE {
  TRUSTEE_IS_UNKNOWN,     // тип не известен
  TRUSTEE_IS_USER,        // пользователь
  TRUSTEE_IS_GROUP,       // группа
  TRUSTEE_IS_DOMAIN,      // домен
  TRUSTEE_IS_ALIAS,       // псевдоним
  TRUSTEE_IS_WELL_KNOWN_GROUP, // общеизвестная группа
  TRUSTEE_IS_DELETED,     // учетная запись удалена
  TRUSTEE_IS_INVALID,     // недействительный тип
}  TRUSTEE_TYPE;
```

Оно определяет тип учетной записи, которую идентифицирует данная структура TRUSTEE.

Поле ptstrName должно содержать адрес имени или идентификатора учетной записи, который идентифицирует данная структура TRUSTEE, в зависи-

мости от значения поля TrusteeForm. Если в поле TrusteeForm установлено значение TRUSTEE_IS_SID, то поле ptstrName указывает на идентификатор учетной записи с одним из следующих типов SID:

- SID учетной записи пользователя;
- SID учетной записи группы;
- универсальный SID;
- общеизвестный SID.

Если же в поле TrusteeForm установлено значение TRUSTEE_IS_NAME, то поле ptstrName указывает на имя учетной записи. Причем имя учетной записи может быть задано в одном из следующих форматов:

- полное имя учетной записи:
  - "имя_домена\имя_пользователя".
- имя предопределенной группы, например:
  - "EVERYONE";
  - "GUEST".
- одно из следующих специальных имен:
  - "CREATOR GROUP" — первичная группа создателя объекта;
  - "CREATOR OWNER" — создатель объекта;
  - "CURRENT_USER" — текущий процесс или поток.

Для работы со структурами типа TRUSTEE используются следующие функции: BuildTrusteeWithName, BuildTrusteeWithSid, GetTrusteeForm, GetTrusteeType и GetTrusteeName. Первые две из этих функций предназначены для инициализации структур типа TRUSTEE, а последние три — для получения информации из таких структур. Работа с этими функциями будет рассмотрена в последующих разделах этой главы.

41.2. Инициализация структуры *TRUSTEE*

Для инициализации структуры типа TRUSTEE, используя имя учетной записи, предназначена функция BuildTrusteeWithName, которая имеет следующий прототип:

```
VOID  BuildTrusteeWithName(
  PTRUSTEE  pTrustee,   // указатель на структуру типа TRUSTEE
  LPSTR     pName       // указатель на имя
);
```

Эта функция не возвращает значения. Первый параметр pTrustee этой функции должен указывать на структуру типа TRUSTEE, которая будет ини-

циализироваться. А параметр pName должен указывать на имя учетной записи, которым будет инициализирована структура. Функция BuildTrusteeWithName устанавливает следующие значения в поля инициализируемой структуры TRUSTEE:

```
// структуры TRUSTEE для сервера нет
pMultipleTrustee = NULL;
// замещения TRUSTEE нет
MultipleTrusteeOperation = NO_MULTIPLE_TRUSTEE;
// ptstrName указывает на имя учетной записи
TrusteeForm = TRUSTEE_IS_NAME;
// тип учетной записи не известен
TrusteeType = TRUSTEE_IS_UNKNOWN;
ptstrName = pName;        // указатель на имя учетной записи
```

В листинге 41.1 приведена программа, в которой структура типа TRUSTEE инициализируется, используя специальное имя, отмечающее текущего пользователя.

Листинг 41.1. Инициализация структуры типа TRUSTEE с использованием специального имени

```
#include <stdio.h>
#include <windows.h>
#include <aclapi.h>

int main()
{
  TRUSTEE  Trustee;
  char  trustee_name[] = "CURRENT_USER";   // имя пользователя
  // строим структуру TRUSTEE по имени
  BuildTrusteeWithName(&Trustee, trustee_name);
  // проверяем значения полей
  if (Trustee.pMultipleTrustee == NULL)
    printf("Server trustee: NULL\n");

  if (Trustee.MultipleTrusteeOperation == NO_MULTIPLE_TRUSTEE)
    printf("Multiple trustee: NO_MULTIPLE_TRUSTEE\n");

  if (Trustee.TrusteeForm == TRUSTEE_IS_NAME)
    printf("Trustee form: TRUSTEE_IS_NAME\n");
```

```
  if (Trustee.TrusteeType == TRUSTEE_IS_UNKNOWN)
    printf("Trustee type: TRUSTEE_IS_UNKNOWN\n");

  printf("Trustee name: %s\n", Trustee.ptstrName);

  return 0;
}
```

Для инициализации структуры типа `TRUSTEE`, используя идентификатор учетной записи, предназначена функция `BuildTrusteeWithSid`, которая имеет следующий прототип:

```
VOID  BuildTrusteeWithSid(
  PTRUSTEE  pTrustee,    // указатель на структуру типа TRUSTEE
  PSID      pSid         // указатель на идентификатор безопасности
);
```

Единственное отличие этой функции от функции `BuildTrusteeWithName` состоит в том, что второй параметр этой функции должен указывать на идентификатор безопасности учетной записи и, соответственно, значения полей `TrusteeForm` и `ptstrName` устанавливаются следующим образом:

```
// ptstrName указывает на SID учетной записи
TrusteeForm = TRUSTEE_IS_SID;
// указатель на SID учетной записи
ptstrName = pSid;
```

Ниже приведена программа, в которой структура типа `TRUSTEE` инициализируется, используя идентификатор безопасности.

Листинг 41.2. Инициализация структуры типа `TRUSTEE` с использованием идентификатора безопасности

```
#define _WIN32_WINNT 0x0500

#include <stdio.h>
#include <windows.h>
#include <aclapi.h>
#include <sddl.h>

int main()
{
  TRUSTEE Trustee;   // информация об учетной записи
```

```
// идентификатор учетной записи
SID_IDENTIFIER_AUTHORITY sia = SECURITY_WORLD_SID_AUTHORITY;
// относительный идентификатор учетной записи
DWORD dwSubAuthority = SECURITY_WORLD_RID;
PSID lpSid = NULL;           // указатель на SID
LPTSTR lpStringSid = NULL;   // указатель на строку с SID

// создаем SID for Everyone
if(!AllocateAndInitializeSid(
    &sia,  // адрес идентификатора учетной записи
    1,     // количество относительных идентификаторов учетной записи
    dwSubAuthority,        // первый RID
    0, 0, 0, 0, 0, 0, 0,   // остальные RID равны 0
    &lpSid) )
{
  DWORD dwRetCode = GetLastError();
  printf( "Allocate and initialize sid failed %u\n", dwRetCode);

  return dwRetCode;
}

// строим структуру TRUSTEE по идентификатору безопасности
BuildTrusteeWithSid(&Trustee, lpSid);

// проверяем значения полей
if (Trustee.pMultipleTrustee == NULL)
  printf("Server trustee: NULL\n");

if (Trustee.MultipleTrusteeOperation == NO_MULTIPLE_TRUSTEE)
  printf("Multiple trustee: NO_MULTIPLE_TRUSTEE\n");

if (Trustee.TrusteeForm == TRUSTEE_IS_SID)
  printf("Trustee form: TRUSTEE_IS_SID\n");

if (Trustee.TrusteeType == TRUSTEE_IS_UNKNOWN)
  printf("Trustee type: TRUSTEE_IS_UNKNOWN\n");

// преобразуем SID в строку
if (!ConvertSidToStringSid(Trustee.ptstrName, &lpStringSid))
```

```
  {
    printf("Convert SID to string SID failed.");

    return GetLastError();
  }
  // распечатываем SID
  printf("SID: %s\n", lpStringSid);

  // освобождаем SID
  FreeSid(lpSid);

  return 0;
}
```

41.3. Структура *EXPLICIT_ACCESS*

Структуры типа `TRUSTEE` идентифицируют учетные записи, для которых существуют элементы в списках управления доступом к охраняемым объектам. Сами же элементы этих списков для доступа на высоком уровне описываются структурами типа `EXPLICIT_ACCESS`. Тип этой структуры определен следующим образом:

```
typedef struct  _EXPLICIT_ACCESS {
  DWORD    grfAccessPermissions;  // права доступа
  ACCESS_MODE  grfAccessMode;     // режим доступа
  DWORD    grfInheritance;        // наследование элемента
  TRUSTEE  Trustee;               // учетная запись
}  EXPLICIT_ACCESS, *PEXPLICIT_ACCESS;
```

В поле `grfAccessPermissions` устанавливаются права доступа к объекту, которыми владеет или нет учетная запись, связанная с элементом списка управления доступом. При доступе субъекта к охраняемому объекту система сверяет затребованные субъектом режимы доступа к объекту с правами доступа, которые устанавливаются в этом поле. То есть эти права доступа контролируют доступ потоков к охраняемому объекту. Все права доступа к охраняемому объекту делятся на четыре категории:

- специфические права доступа (specific access rights), т. е. такие права, которые присущи только данному объекту;
- стандартные права доступа (standard access rights), т. е. такие права, которые присущи всем объектам;

- родовые права доступа (generic access rights), т. е. такие права, которые используются всеми объектами, но при их реализации отображаются в специфические права для каждого объекта;
- права доступа к списку SACL.

Ниже перечислены флаги, соответствующие правам доступа из каждой категории, исключая специфические права доступа, которые зависят от типа охраняемого объекта. Правда, для специфических прав доступа определена символьная константа `SPECIFIC_RIGHTS_ALL`, которая включает все права доступа. Эта константа устанавливает для охраняемого объекта флаги всех специфических прав доступа. Специфические права доступа к конкретным объектам перечислены в *гл. 45* (при рассмотрении управлением безопасностью объектов на низком уровне).

Стандартные права доступа задаются следующими символьными константами:

- `DELETE` — право удаления объекта;
- `READ_CONTROL` — право чтения управляющей информации из дескриптора безопасности, исключая содержимое списка DACL;
- `SYNCHRONIZE` — право использования объекта для синхронизации;
- `WRITE_DAC` — право записи в список DACL объекта;
- `WRITE_OWNER` — право записи владельца объекта;

Кроме того, для стандартных прав доступа определены символьные константы, которые включают несколько основных прав доступа:

- `STANDARD_RIGHTS_ALL` — включает все права доступа;
- `STANDARD_RIGHTS_REQUIRED` — включает права `DELETE`, `READ_CONTROL`, `WRITE_DAC` и `WRITE_OWNER`;
- `STANDARD_RIGHTS_READ` — эквивалентно `READ_CONTROL`;
- `STANDARD_RIGHTS_WRITE` — эквивалентно `READ_CONTROL`;
- `STANDARD_RIGHTS_EXECUTE` — эквивалентно `READ_CONTROL`.

Родовые права доступа:

- `GENERIC_READ` — право читать объект;
- `GENERIC_WRITE` — право записывать объект;
- `GENERIC_EXECUTE` — право исполнять объект.

Следующее родовое право включает все основные родовые права доступа: `GENERIC_ALL` — включает права `GENERIC_READ`, `GENERIC_WRITE` и `GENERIC_EXECUTE`.

Права доступа к списку SACL включают единственное право `ACCESS_SYSTEM_SECURITY` — право доступа к списку SACL.

Если структура типа EXPLICIT_ACCESS используется для доступа к элементам из списка DACL, то в поле grfAccessPermissions может быть установлена любая комбинация флагов, отмечающих специфические, стандартные и родовые права доступа. Если структура типа EXPLICIT_ACCESS используется для доступа к элементам из списка SACL, то в поле grfAccessPermissions имеет смысл устанавливать только флаг доступа ACCESS_SYSTEM_SECURITY.

Теперь перейдем к полю grfAccessMode структуры типа EXPLICIT_ACCESS. В этом поле может быть установлена одна из констант, которая принадлежит следующему перечислению:

```
typedef enum  _ACCESS_MODE {
  NOT_USED_ACCESS = 0,   // не используется
  GRANT_ACCESS,  // добавить права доступа к существующим правам
  SET_ACCESS,    // установить права доступа, удалив существующие права
  DENY_ACCESS,   // запретить права доступа
  REVOKE_ACCESS, // удалить все элементы из списка управления доступом
  SET_AUDIT_SUCCESS,// установить флаг генерации сообщения при
                    // успешном открытии доступа к охраняемому объекту
  SET_AUDIT_FAILURE // установить флаг генерации сообщения при
                    // неудачном открытии доступа к охраняемому объекту
} ACCESS_MODE;
```

Она указывает на режим использования прав доступа, установленных в поле grfAccessPermissions. То есть добавляет, запрещает или удаляет права доступа к данному охраняемому объекту для учетной записи, идентифицируемой полем Trustee этой же структуры. Отметим, что константы SET_AUDIT_SUCCESS и SET_AUDIT_FAILURE имеет смысл устанавливать только в структурах EXPLICIT_ACCESS, которые используются для доступа к элементам списка управления доступом SACL. Причем эти константы могут быть установлены одновременно, обращаясь с ними как с управляющими флагами.

В поле grfInheritance структуры EXPLICIT_ACCESS устанавливаются флаги, которые управляют наследованием элементов из списка управления доступом. Возможно установить любую комбинацию следующих флагов:

- CONTAINER_INHERIT_ACE — элемент наследуется только контейнерными дочерними объектами;
- OBJECT_INHERIT_ACE — элемент наследуется только не контейнерными дочерними объектами;
- SUB_CONTAINERS_ONLY_INHERIT — тоже, что и CONTAINER_INHERIT_ACE;
- SUB_OBJECTS_ONLY_INHERIT — тоже, что и OBJECT_INHERIT_ACE;
- SUB_CONTAINERS_AND_OBJECTS_INHERIT — элемент наследуется как контейнерными, так и неконтейнерными дочерними объектами;

- INHERIT_ONLY_ACE — отмечает, что элемент был унаследован от родительского объекта, но этот элемент не участвует в контроле доступа к данному объекту;
- NO_PROPAGATE_INHERIT_ACE — элемент наследуется, но флаги OBJECT_INHERIT_ACE и CONTAINER_INHERIT_ACE не устанавливаются в наследованных элементах.

Последнее поле — Trustee — имеет тип TRUSTEE и идентифицирует учетную запись пользователя, для которой создается данный элемент списка управления доступом. Структура типа TRUSTEE и ее инициализация были рассмотрены в *разд. 41.1*, *41.2*.

41.4. Инициализация структуры *EXPLICIT_ACCESS*

Для инициализации структуры типа EXPLICIT_ACCESS используется функция BuildExplicitAccessWithName, которая имеет следующий прототип:

```
VOID  BuildExplicitAccessWithName(
  PEXPLICIT_ACCESS  pExplicitAccess, // адрес структуры EXPLICIT_ACCESS
  LPSTR  pTrusteeName,          // имя учетной записи
  DWORD  AccessPermissions,     // флаги доступа
  ACCESS_MODE  AccessMode,      // режим доступа
  DWORD  Inheritance            // режим наследования
);
```

Эта функция не возвращает ничего. Параметры функции BuildExplicitAccessWithName имеют следующее назначение.

Параметр pExplicitAccess должен указывать на структуру типа EXPLICIT_ACCESS, которую будет инициализировать функция.

Параметр pTrusteeName должен указывать на имя учетной записи, которым будет инициализирована структура типа TRUSTEE, входящая в структуру типа EXPLICIT_ACCESS, заданную параметром pExplicitAccess. Функция инициализирует поля этой структуры TRUSTEE таким же образом, как это делает функция BuildTrusteeWithName, описанная в *разд. 41.2*.

Параметр AccessPermissions должен содержать маску установленных флагов, которая будет перенесена в поле grfAccessPermissions структуры EXPLICIT_ACCESS, на которую указывает параметр pExplicitAccess. Для установки флагов нужно использовать символьные константы, которые определяют права доступа к объекту и были рассмотрены в предыдущем разделе при описании поля grfAccessPermissions структуры EXPLICIT_ACCESS.

Параметр AccessMode должен содержать режим доступа к элементу списка управления доступом. В этом параметре должно быть установлено одно из значений перечислимой константы типа ACCESS_MODE, который был рассмотрен в предыдущем разделе при описании поля grfAccessMode структуры типа EXPLICIT_ACCESS.

Параметр Inheritance должен содержать режим наследования элемента списка управления доступом, для которого инициализируется структура типа EXPLICIT_ACCESS. В этом параметре можно установить любую комбинацию флагов, которые были рассмотрены в предыдущем разделе при описании поля grfInheritance структуры типа EXPLICIT_ACCESS.

Программа, в которой структура типа EXPLICIT_ACCESS инициализируется, используя функцию BuildExplicitAccessWithName, приведена в *разд. 41.5*.

41.5. Создание нового списка управления доступом

Как для создания новых списков управления доступом, так и для модификации существующих списков управления доступом предназначена функция SetEntriesInAcl, которая имеет следующий прототип:

```
DWORD  SetEntriesInAcl(
  ULONG  cCountOfExplicitEntries,  // количество элементов в массиве
  PEXPLICIT_ACCESS  pListOfExplicitEntries,  // массив структур
  PACL  OldAcl,    // адрес старого списка
  PACL  *NewAcl    // адрес указателя на новый список
);
```

В случае успешного завершения функция возвращает значение ERROR_SUCCESS, а в случае неудачи — код ошибки. Параметры функции имеют следующее назначение.

Параметр cCountOfExplicitEntries должен содержать количество элементов в массиве структур типа EXPLICIT_ACCESS, на который указывает параметр pListOfExplicitEntries.

Параметр pListOfExplicitEntries должен указывать на массив структур типа EXPLICIT_ACCESS, которые содержат информацию о добавляемых в список управления доступом элементах.

Параметр OldAcl должен указывать на старый список управления доступом. Если в этом параметре установлено значение NULL, то система создает новый список управления доступом.

Параметр *NewAcl должен содержать адрес указателя на список управления доступом. В случае успешного завершения в этот указатель функция запи-

шет адрес нового списка управления доступом, который создается системой путем присоединения элементов из массива `pListOfExplicitEntries` к старому списку управления доступом, на который указывает параметр `OldAcl`. Память под новый список управления доступом резервирует система. Поэтому после использования нового списка управления доступом эту память нужно освободить путем вызова функции `LocalFree`.

Сделаем важное замечание относительно порядка элементов в новом списке управления доступом, который создает функция `SetEntriesInAcl`. Элементы типа `ACCESS_DENIED_ACE`, т. е. запрещающие доступ к объекту, включаются в начало нового списка управления доступом. Элементы типа `ACCESS_ALLOWED_ACE`, т. е. запрещающие доступ к объекту, включаются в новый список управления доступом после всех запрещающих доступ элементов, но перед всеми разрешающими доступ элементами из старого списка управления доступом к охраняемому объекту.

В листинге 41.3 приведена программа, в которой создается новый список управления доступом DACL, используя функцию `SetEntriesInAcl`. После создания список DACL устанавливается в дескриптор безопасности объекта. Затем создается новый каталог, атрибуты защиты которого содержат адрес созданного дескриптора безопасности. В результате к созданному каталогу разрешается доступ только тому пользователю, для учетной записи которого был создан элемент в списке управления доступом DACL.

Листинг 41.3. Создание нового списка управления доступом DACL

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>
#include <lm.h>
#include <aclapi.h>
#include <accctrl.h>

int main()
{
  wchar_t wchDirName[248];   // имя каталога
  wchar_t wchAccountName[UNLEN + 255];   // имя учетной записи

  SECURITY_DESCRIPTOR sd;    // дескриптор безопасности каталога
```

```
SECURITY_ATTRIBUTES sa;     // атрибуты защиты каталога

EXPLICIT_ACCESS  ea;        // информация для элемента списка DACL
PACL lpNewDacl;             // указатель на список DACL

DWORD dwErrCode;            // код возврата

// инициализируем версию дескриптора безопасности
if (!InitializeSecurityDescriptor(
  &sd,
  SECURITY_DESCRIPTOR_REVISION))
{
  dwErrCode = GetLastError();
  printf("Initialize security descroptor failed.\n");
  printf("Error code: %d\n", dwErrCode);

  return dwErrCode;
}

// устанавливаем SID владельца объекта
if (!SetSecurityDescriptorOwner(
  &sd,        // адрес дескриптора безопасности
  NULL,       // не задаем владельца
  SE_OWNER_DEFAULTED))   // определить владельца по умолчанию
{
  dwErrCode = GetLastError();
  perror("Set security descriptor owner failed.\n");
  printf("The last error code: %u\n", dwErrCode);

  return dwErrCode;
}

// устанавливаем SID первичной группы владельца
if (!SetSecurityDescriptorGroup(
  &sd,        // адрес дескриптора безопасности
  NULL,       // не задаем первичную группу
  SE_GROUP_DEFAULTED))  // определить первичную группу по умолчанию
{
  dwErrCode = GetLastError();
```

```
  perror("Set security descriptor group failed.\n");
  printf("The last error code: %u\n", dwErrCode);

  return dwErrCode;
}

// вводим имя домена и пользователя
printf("The following user will have full access to directory.\n");
printf("Input domain account (domain_name\\user_name): ");
wscanf(L"%s", wchAccountName);       // вводим имя учетной записи

// строим структуру EXPLICIT_ACCESS по имени
BuildExplicitAccessWithName(
  &ea,              // адрес структуры ExplicitAccess
  wchAccountName,   // имя учетной записи
  GENERIC_ALL,      // полный доступ к объекту
  SET_ACCESS,       // установить доступ
  NO_INHERITANCE    // нет наследования
  );

// создаем список DACL
dwErrCode = SetEntriesInAcl(
  1,            // один элемент
  &ea,          // адрес структуры ExplicitAccess
  NULL,         // старого DACL нет
  &lpNewDacl); // адрес нового DACL
if (dwErrCode != ERROR_SUCCESS)
{
  perror("Set entries in DACL failed.\n");
  printf("The last error code: %u\n", dwErrCode);

  return dwErrCode;
}

// устанавливаем DACL в дескриптор безопасности
if (!SetSecurityDescriptorDacl(
  &sd,          // адрес дескриптора безопасности
  TRUE,         // DACL присутствует
  lpNewDacl,    // указатель на DACL
```

```
  FALSE))      // DACL не задан по умолчанию
{
  dwErrCode = GetLastError();
  perror("Set security descriptor group failed.\n");
  printf("The last error code: %u\n", dwErrCode);

  return dwErrCode;
}

// проверяем структуру дескриптора безопасности
if (!IsValidSecurityDescriptor(&sd))
{
  dwErrCode = GetLastError();
  perror("Security descriptor is invalid.\n");
  printf("The last error code: %u\n", dwErrCode);

  return dwErrCode;
}

// инициализируем атрибуты безопасности
sa.nLength = sizeof(sa);     // устанавливаем длину атрибутов защиты
sa.lpSecurityDescriptor = &sd;   // устанавливаем адрес дескриптора
                                 // безопасности
sa.bInheritHandle = FALSE;   // дескриптор каталога ненаследуемый

// читаем имя создаваемого каталога
printf("Input a directory name: ");
wscanf(L"%s", wchDirName);     // вводим имя каталога

// создаем каталог
if (!CreateDirectory(wchDirName, &sa))
{
  dwErrCode = GetLastError();
  perror("Security descriptor is invalid.\n");
  printf("The last error code: %u\n", dwErrCode);

  return dwErrCode;
}
```

```
  // освобождаем память, захваченную под DACL
  LocalFree(lpNewDacl);

  printf("The directory is created.\n");

  return 0;
}
```

В листинге 41.4 приведена программа, в которой показано, как можно создать новый список SACL, используя функцию `SetEntriesInAcl`. Фактически эта программа является расширением программы из листинга 41.3. Отличие заключается в том, что для пользователя, которому предоставляется полный доступ к каталогу, в список SACL включается элемент, который управляет аудитом доступа этого пользователя к создаваемому каталогу.

Сделаем следующие замечания. Для создания списка SACL требуется, чтобы пользователь имел действующую привилегию `SE_SECURITY_NAME`. Такую привилегию имеют администраторы системы, но по умолчанию она недействительна. Поэтому прежде чем работать со списком SACL, эта привилегия должна быть активизирована. Для этого используется функция `AdjustTokenPrivileges`. Подробнее работа с привилегиями рассмотрена в *гл. 43*.

Листинг 41.4. Создание новых списков управления доступом DACL и SACL

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>
#include <lm.h>
#include <aclapi.h>

int main()
{
  HANDLE hProcess;       // дескриптор процесса
  HANDLE hTokenHandle;   // дескриптор маркера доступа

  TOKEN_PRIVILEGES tp;   // привилегии маркера доступа

  wchar_t wchDirName[248];   // имя каталога
```

```
wchar_t wchAccountName[UNLEN + 255];   // имя учетной записи

SECURITY_DESCRIPTOR sd;     // дескриптор безопасности каталога
SECURITY_ATTRIBUTES sa;     // атрибуты защиты каталога

EXPLICIT_ACCESS  eaDacl;    // информация для элемента списка DACL
PACL lpNewDacl;             // указатель на список DACL

EXPLICIT_ACCESS  eaSacl;    // информация для элемента списка SACL
PACL lpNewSacl;             // указатель на список DACL

DWORD dwErrCode;      // код возврата

// получаем дескриптор процесса
hProcess = GetCurrentProcess();

// получаем маркер доступа процесса
if (!OpenProcessToken(
  hProcess,           // дескриптор процесса
  TOKEN_ALL_ACCESS,   // полный доступ к маркеру доступа
  &hTokenHandle))     // дескриптор маркера
{
  dwErrCode = GetLastError();
  printf( "Open process token failed: %u\n", dwErrCode);

  return dwErrCode;
}

// устанавливаем общее количество привилегий
tp.PrivilegeCount = 1;

// определяем идентификатор привилегии для установки аудита
if (!LookupPrivilegeValue(
  NULL,       // ищем идентификатор привилегии на локальном компьютере
  SE_SECURITY_NAME,  // привилегия для аудита
  &(tp.Privileges[0].Luid)))
{
  dwErrCode = GetLastError();
  printf("Lookup privilege value failed.\n");
```

```
  printf("Error code: %d\n", dwErrCode);

  return dwErrCode;
}

// разрешаем привилегию аудита
tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

// разрешаем привилегию для установки аудита
if (!AdjustTokenPrivileges(
  hTokenHandle,  // дескриптор маркера доступа процесса
  FALSE,   // не запрещаем все привилегии
  &tp,     // адрес привилегий
  0,       // длины буфера нет
  NULL,    // предыдущее состояние привилегий не нужно
  NULL))   // длина буфера не нужна
{
  dwErrCode = GetLastError();
  printf("Lookup privilege value failed.\n");
  printf("Error code: %d\n", dwErrCode);

  return dwErrCode;
}

// инициализируем версию дескриптора безопасности
if (!InitializeSecurityDescriptor(
  &sd,
  SECURITY_DESCRIPTOR_REVISION))
{
  dwErrCode = GetLastError();
  printf("Initialize security descroptor failed.\n");
  printf("Error code: %d\n", dwErrCode);

  return dwErrCode;
}

// устанавливаем SID владельца объекта
if (!SetSecurityDescriptorOwner(
  &sd,       // адрес дескриптора безопасности
```

```
  NULL,      // не задаем владельца
  SE_OWNER_DEFAULTED))   // определить владельца по умолчанию
{
  dwErrCode = GetLastError();
  perror("Set security descriptor owner failed.\n");
  printf("The last error code: %u\n", dwErrCode);

  return dwErrCode;
}

// устанавливаем SID первичной группы владельца
if (!SetSecurityDescriptorGroup(
  &sd,       // адрес дескриптора безопасности
  NULL,      // не задаем первичную группу
  SE_GROUP_DEFAULTED))   // определить первичную группу по умолчанию
{
  dwErrCode = GetLastError();
  perror("Set security descriptor group failed.\n");
  printf("The last error code: %u\n", dwErrCode);

  return dwErrCode;
}

// вводим имя домена и пользователя
printf("The following user will have full access to directory.\n");
printf("Input domain account (domain_name\\user_name): ");
wscanf(L"%s", wchAccountName);       // вводим имя учетной записи

// строим структуру EXPLICIT_ACCESS по имени для списка DACL
BuildExplicitAccessWithName(
  &eaDacl,         // адрес структуры ExplicitAccess
  wchAccountName,  // имя учетной записи
  GENERIC_ALL,     // полный доступ к объекту
  SET_ACCESS,      // установить доступ
  NO_INHERITANCE   // нет наследования
  );

// создаем список DACL
dwErrCode = SetEntriesInAcl(
```

```
  1,          // один элемент
  &eaDacl,    // адрес структуры ExplicitAccess
  NULL,       // старого DACL нет
  &lpNewDacl);   // адрес нового DACL
if (dwErrCode != ERROR_SUCCESS)
{
  perror("Set entries in DACL failed.\n");
  printf("The last error code: %u\n", dwErrCode);

  return dwErrCode;
}

// устанавливаем DACL в дескриптор безопасности
if (!SetSecurityDescriptorDacl(
  &sd,        // адрес дескриптора безопасности
  TRUE,       // DACL присутствует
  lpNewDacl, // указатель на DACL
  FALSE))     // DACL не задан по умолчанию
{
  dwErrCode = GetLastError();
  perror("Set security descriptor group failed.\n");
  printf("The last error code: %u\n", dwErrCode);

  return dwErrCode;
}

// строим структуру EXPLICIT_ACCESS по имени для списка SACL
BuildExplicitAccessWithName(
  &eaSacl,           // адрес структуры ExplicitAccess
  wchAccountName,    // имя учетной записи
  GENERIC_ALL,       // полный доступ к объекту
  SET_AUDIT_SUCCESS, // установить доступ
  NO_INHERITANCE     // нет наследования
  );

// создаем список SACL
dwErrCode = SetEntriesInAcl(
  1,          // один элемент
  &eaSacl,    // адрес структуры ExplicitAccess
```

```
    NULL,      // старого SACL нет
    &lpNewSacl);   // адрес нового SACL
  if (dwErrCode != ERROR_SUCCESS)
  {
    perror("Set entries in DACL failed.\n");
    printf("The last error code: %u\n", dwErrCode);

    return dwErrCode;
  }

  // устанавливаем SACL в дескриптор безопасности
  if (!SetSecurityDescriptorSacl(
    &sd,          // адрес дескриптора безопасности
    TRUE,         // SACL присутствует
    lpNewSacl,    // указатель на SACL
    FALSE))       // SACL не задан по умолчанию
  {
    dwErrCode = GetLastError();
    perror("Set security descriptor group failed.\n");
    printf("The last error code: %u\n", dwErrCode);

    return dwErrCode;
  }

  // проверяем структуру дескриптора безопасности
  if (!IsValidSecurityDescriptor(&sd))
  {
    dwErrCode = GetLastError();
    perror("Security descriptor is invalid.\n");
    printf("The last error code: %u\n", dwErrCode);

    return dwErrCode;
  }

  // инициализируем атрибуты безопасности
  sa.nLength = sizeof(sa);     // устанавливаем длину атрибутов защиты
  // устанавливаем адрес дескриптора безопасности
  sa.lpSecurityDescriptor = &sd;
```

```
// дескриптор каталога ненаследуемый
sa.bInheritHandle = FALSE;

// читаем имя создаваемого каталога
printf("Input a directory name: ");
wscanf(L"%s", wchDirName);     // вводим имя каталога

// создаем каталог
if (!CreateDirectory(wchDirName, &sa))
{
  dwErrCode = GetLastError();
  perror("Security descriptor is invalid.\n");
  printf("The last error code: %u\n", dwErrCode);

  return dwErrCode;
}

// запрещаем привилегию аудита
tp.Privileges[0].Attributes = 0;

// запрещаем привилегию для установки аудита
if (!AdjustTokenPrivileges(
  hTokenHandle,  // дескриптор маркера доступа процесса
  FALSE,     // не запрещаем все привилегии
  &tp,       // адрес привилегий
  0,         // длины буфера нет
  NULL,      // предыдущее состояние привилегий не нужно
  NULL))     // длина буфера не нужна
{
  dwErrCode = GetLastError();
  printf("Lookup privilege value failed.\n");
  printf("Error code: %d\n", dwErrCode);

  return dwErrCode;
}

// освобождаем память, захваченную под списки DACL и SACL
LocalFree(lpNewDacl);
```

```
  LocalFree(lpNewSacl);

  printf("The directory is created.\n");

  return 0;
}
```

41.6. Модификация списка управления доступом

Для модификации списков управления доступом, так же как и для создания нового списка, используется функция `SetEntriesInAcl`, прототип которой был рассмотрен в *разд. 41.5*. В листинге 41.5 приведена программа, в которой эта функция используется для добавления элемента в список управления доступом DACL существующего каталога. Каталог можно создать, используя программу из листинга 41.3.

Листинг 41.5. Добавление элемента в список управления доступом DACL

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>
#include <lm.h>
#include <aclapi.h>

int main()
{
  wchar_t wchDirName[248];     // имя каталога
  wchar_t wchAccountName[UNLEN + 255];   // имя учетной записи

  EXPLICIT_ACCESS  ea;     // информация для элемента списка DACL
  PACL lpOldDacl;          // указатель на старый список DACL
  PACL lpNewDacl;          // указатель на новый список DACL

  PSECURITY_DESCRIPTOR lpSD;   // указатель на дескриптор безопасности
```

```
DWORD dwErrCode;     // код возврата

// читаем имя созданного каталога
printf("Input a directory name: ");
wscanf(L"%s", wchDirName);       // вводим имя каталога

// получаем SD этого каталога
dwErrCode = GetNamedSecurityInfo(
  wchDirName,        // имя каталога
  SE_FILE_OBJECT,    // объект файл
  DACL_SECURITY_INFORMATION,   // получаем DACL
  NULL,          // адрес указателя на SID владельца
  NULL,          // адрес указателя на первичную группу
  &lpOldDacl,    // указатель на DACL
  NULL,          // указатель на SACL не нужен
  &lpSD);        // адрес указателя на дескриптор безопасности
if (dwErrCode != ERROR_SUCCESS)
{
  printf("Get named security info failed.\n");
  printf("Error code: %u\n", dwErrCode);

  return dwErrCode;
}

// вводим имя домена и пользователя для нового элемента списка DACL
printf("The following user will be added in DACL.\n");
printf("Input domain account (domain_name\\user_name): ");
wscanf(L"%s", wchAccountName);       // вводим имя учетной записи

// строим структуру EXPLICIT_ACCESS по имени
BuildExplicitAccessWithName(
  &ea,       // адрес структуры ExplicitAccess
  wchAccountName,    // имя учетной записи
  GENERIC_ALL,       // полный доступ
  SET_ACCESS,        // установить доступ
  NO_INHERITANCE     // нет наследования
  );

// создаем список DACL
```

```
  dwErrCode = SetEntriesInAcl(
    1,              // добавляем один элемент в список DACL
    &ea,            // адрес структуры ExplicitAccess
    lpOldDacl,      // адрес старого списка DACL
    &lpNewDacl);    // адрес указателя на новый список DACL
  if (dwErrCode != ERROR_SUCCESS)
  {
    perror("Set entries in DACL failed.\n");
    printf("The last error code: %u\n", dwErrCode);

    return dwErrCode;
  }

  // устанавливаем новый список DACL
  dwErrCode = SetNamedSecurityInfo(
    wchDirName,       // имя файла
    SE_FILE_OBJECT,   // объект файл
    DACL_SECURITY_INFORMATION,   // изменяем список DACL
    NULL,             // владельца не изменяем
    NULL,             // первичную группу не изменяем
    lpNewDacl,        // новый DACL
    NULL);            // SACL не изменяем
  if (dwErrCode != ERROR_SUCCESS)
  {
    printf("Set named security info failed.\n");
    printf("Error code: %u\n", dwErrCode);

    return dwErrCode;
  }

  // освобождаем память
  LocalFree(lpSD);
  LocalFree(lpNewDacl);

  printf("The DACL of directory is modified.\n");

  return 0;
}
```

Теперь, в листинге 41.6, приведем программу, в которой функция SetEntriesInAcl используется для добавления по одному элементу в списки управления доступом DACL и SACL существующего каталога. Каталог можно создать, используя программу из листинга 41.4. Относительно следующей программы, как и в случае с программой из листинга 41.4, заметим, что для модификации списка SACL требуется, чтобы пользователь имел действующую привилегию SE_SECURITY_NAME. Такую привилегию имеют администраторы системы, но по умолчанию она недействительна. Поэтому прежде чем модифицировать список SACL, эта привилегия должна быть активизирована. Для этого используется функция AdjustTokenPrivileges. Более подробно работа с привилегиями рассмотрена в *гл. 43*.

Листинг 41.6. Добавление элементов в списки управления доступом DACL и SACL

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>
#include <lm.h>
#include <aclapi.h>

int main()
{
  HANDLE hProcess;          // дескриптор процесса
  HANDLE hTokenHandle;      // дескриптор маркера доступа

  TOKEN_PRIVILEGES tp;      // привилегии маркера доступа

  wchar_t wchDirName[248];   // имя каталога
  wchar_t wchAccountName[UNLEN + 255];   // имя учетной записи

  EXPLICIT_ACCESS  eaSacl;   // информация для элемента списка DACL
  EXPLICIT_ACCESS  eaDacl;   // информация для элемента списка DACL

  PACL lpOldDacl;       // указатель на старый список DACL
  PACL lpNewDacl;       // указатель на новый список DACL
  PACL lpOldSacl;       // указатель на старый список SACL
  PACL lpNewSacl;       // указатель на новый список SACL
```

```
PSECURITY_DESCRIPTOR lpSD;   // указатель на дескриптор безопасности

DWORD dwErrCode;     // код возврата

// получаем дескриптор процесса
hProcess = GetCurrentProcess();

// получаем маркер доступа процесса
if (!OpenProcessToken(
  hProcess,          // дескриптор процесса
  TOKEN_ALL_ACCESS,  // полный доступ к маркеру доступа
  &hTokenHandle))    // дескриптор маркера
{
  dwErrCode = GetLastError();
  printf( "Open process token failed: %u\n", dwErrCode);

  return dwErrCode;
}

// устанавливаем общее количество привилегий
tp.PrivilegeCount = 1;

// определяем идентификатор привилегии для установки аудита
if (!LookupPrivilegeValue(
  NULL,      // ищем идентификатор привилегии на локальном компьютере
  SE_SECURITY_NAME,  // привилегия для аудита
  &(tp.Privileges[0].Luid)))
{
  dwErrCode = GetLastError();
  printf("Lookup privilege value failed.\n");
  printf("Error code: %d\n", dwErrCode);

  return dwErrCode;
}

// разрешаем привилегию аудита
tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

// разрешаем привилегию для установки аудита
```

```
if (!AdjustTokenPrivileges(
  hTokenHandle,  // дескриптор маркера доступа процесса
  FALSE,     // не запрещаем все привилегии
  &tp,       // адрес привилегий
  0,         // длины буфера нет
  NULL,      // предыдущее состояние привилегий не нужно
  NULL))     // длина буфера не нужна
{
  dwErrCode = GetLastError();
  printf("Lookup privilege value failed.\n");
  printf("Error code: %d\n", dwErrCode);

  return dwErrCode;
}

// читаем имя созданного каталога
printf("Input a directory name: ");
wscanf(L"%s", wchDirName);     // вводим имя каталога

// получаем SD этого каталога
dwErrCode = GetNamedSecurityInfo(
  wchDirName,      // имя каталога
  SE_FILE_OBJECT,  // объект файл
  DACL_SECURITY_INFORMATION |    // получаем DACL и SACL
  SACL_SECURITY_INFORMATION,
  NULL,            // адрес указателя на SID владельца
  NULL,            // адрес указателя на первичную группу
  &lpOldDacl,      // указатель на DACL
  &lpOldSacl,      // указатель на SACL
  &lpSD);          // адрес указателя на дескриптор безопасности
if (dwErrCode != ERROR_SUCCESS)
{
  printf("Get named security info failed.\n");
  printf("Error code: %u\n", dwErrCode);

  return dwErrCode;
}

// вводим имя домена и пользователя для нового элемента списка DACL
```

```
printf("The following user will be added in DACL and SACL.\n");
printf("Input domain account (domain_name\\user_name): ");
wscanf(L"%s", wchAccountName);     // вводим имя учетной записи

// строим структуру EXPLICIT_ACCESS для DACL по имени
BuildExplicitAccessWithName(
  &eaDacl,         // адрес структуры ExplicitAccess
  wchAccountName,  // имя учетной записи
  GENERIC_READ,    // только чтение
  SET_ACCESS,      // установить доступ
  NO_INHERITANCE   // нет наследования
  );

// создаем список DACL
dwErrCode = SetEntriesInAcl(
  1,             // добавляем один элемент в список DACL
  &eaDacl,       // адрес структуры ExplicitAccess
  lpOldDacl,     // адрес старого списка DACL
  &lpNewDacl);   // адрес указателя на новый список DACL
if (dwErrCode != ERROR_SUCCESS)
{
  perror("Set entries in DACL failed.\n");
  printf("The last error code: %u\n", dwErrCode);

  return dwErrCode;
}

// строим структуру EXPLICIT_ACCESS для SACL по имени
BuildExplicitAccessWithName(
  &eaSacl,         // адрес структуры ExplicitAccess
  wchAccountName,  // имя учетной записи
  GENERIC_READ,    // только чтение
  SET_AUDIT_SUCCESS,   // аудиторское сообщение при успешном доступе
  NO_INHERITANCE   // нет наследования
  );

// создаем список SACL
dwErrCode = SetEntriesInAcl(
  1,             // добавляем один элемент в список DACL
```

```
  &eaSacl,       // адрес структуры ExplicitAccess
  lpOldSacl,     // адрес старого списка DACL
  &lpNewSacl);   // адрес указателя на новый список DACL
if (dwErrCode != ERROR_SUCCESS)
{
  perror("Set entries in SACL failed.\n");
  printf("The last error code: %u\n", dwErrCode);

  return dwErrCode;
}

// устанавливаем новые списки DACL и SACL
dwErrCode = SetNamedSecurityInfo(
  wchDirName,      // имя файла
  SE_FILE_OBJECT,  // объект файл
  DACL_SECURITY_INFORMATION |  // изменяем списки DACL
  SACL_SECURITY_INFORMATION,   // и SACL
  NULL,            // владельца не изменяем
  NULL,            // первичную группу не изменяем
  lpNewDacl,       // новый DACL
  lpNewSacl);      // новый SACL
if (dwErrCode != ERROR_SUCCESS)
{
  printf("Set named security info failed.\n");
  printf("Error code: %u\n", dwErrCode);

  return dwErrCode;
}

// запрещаем привилегию аудита
tp.Privileges[0].Attributes = 0;

// разрешаем привилегию для установки аудита
if (!AdjustTokenPrivileges(
  hTokenHandle,    // дескриптор маркера доступа процесса
  FALSE,     // не запрещаем все привилегии
  &tp,       // адрес привилегий
  0,         // длины буфера нет
  NULL,      // предыдущее состояние привилегий не нужно
```

```
    NULL))      // длина буфера не нужна
  {
    dwErrCode = GetLastError();
    printf("Lookup privilege value failed.\n");
    printf("Error code: %d\n", dwErrCode);

    return dwErrCode;
  }

  // освобождаем память
  LocalFree(lpSD);
  LocalFree(lpNewDacl);
  LocalFree(lpNewSacl);

  printf("The DACL and SACL of the directory are modified.\n");

  return 0;
}
```

41.7. Получение элементов из списка управления доступом

Для получения информации, содержащейся в элементах списка управления доступом, используется функция `GetExplicitEntriesFromAcl`, которая имеет следующий прототип:

```
DWORD  GetExplicitEntriesFromAcl(
  PACL    pAcl,              // указатель на список управления доступом
  PULONG  pcCountOfExplicitEntries,    // количество элементов в списке
  PEXPLICIT_ACCESS  *pListOfExplicitEntries // адрес указателя на буфер
);
```

В случае успешного завершения функция возвращает значение `ERROR_SUCCESS`, а в случае неудачи — код ошибки. Параметры функции имеют следующее назначение.

Параметр `pAcl` должен указывать на список управления доступом, из которого извлекается информация, хранящаяся в элементах этого списка.

Параметр `pcCountOfExplicitEntries` должен указывать на переменную типа `ULONG`, в которую функция запишет количество элементов, находящихся в списке управления доступом.

Параметр pListOfExplicitEntries должен содержать адрес указателя типа PEXPLICIT_ACCESS. В случае успешного завершения функция запишет в этот указатель адрес массива структур типа EXPLICIT_ACCESS, в котором будет храниться информация из элементов списка управления доступом. Количество элементов в массиве равно числу, записанному функцией по адресу, заданному параметром pcCountOfExplicitEntries. Каждый элемент массива соответствует одному элементу из списка управления доступом. При этом заметим, что, начиная с операционной системы Windows 2000, функция не возвращает элементы из списка управления доступом, которые были наследованы от родительских объектов. Так как система сама распределяет память под массив структур типа EXPLICIT_ACCESS, то после использования эту память нужно освободить посредством вызова функции LocalFree.

Программа, в которой для получения элементов из списка управления доступом DACL используется функция GetExplicitEntriesFromAcl, приведена в листинге 41.7.

41.8. Получение информации из структуры *TRUSTEE*

Для получения формы и типа структуры типа TRUSTEE используются функции GetTrusteeForm и GetTrusteeType соответственно. Для получения из структуры TRUSTEE имени или идентификатора учетной записи используется функция GetTrusteeName. Прототипы этих функций описаны ниже.

Функция GetTrusteeForm имеет следующий прототип:

```
TRUSTEE_FORM  GetTrusteeForm(
  PTRUSTEE  pTrustee       // указатель на структуру TRUSTEE
);
```

Единственным параметром этой функции является указатель на структуру типа TRUSTEE, форму которой вернет функция. Функция возвращает одно из значений перечисления типа TRUSTEE_FORM, которое было подробно рассмотрено в *разд. 41.1.*

Функция GetTrusteeType имеет следующий прототип:

```
TRUSTEE_TYPE  GetTrusteeType(
  PTRUSTEE  pTrustee       // указатель на структуру TRUSTEE
);
```

Единственным параметром этой функции является указатель на структуру типа TRUSTEE, тип которой вернет функция. Функция возвращает одно из значений перечисления типа TRUSTEE_TYPE, которое было подробно рассмотрено в *разд. 41.1.*

Функция `GetTrusteeName` имеет следующий прототип:

```
LPSTR  GetTrusteeName(
  PTRUSTEE  pTrustee       // указатель на структуру TRUSTEE
);
```

Единственным параметром этой функции является указатель на структуру типа `TRUSTEE`, из которой функция вернет указатель на имя учетной записи. Это имя может быть как символьной строкой, так и идентификатором безопасности в зависимости от формы структуры `TRUSTEE`. Подробнее об имени структуры `TRUSTEE` рассказано в *разд. 41.1.*

В листинге 41.7 приведена программа, которая сначала получает элементы из списка управления доступом DACL, используя для этого функцию `GetExplicitEntriesFromAcl`. А затем получает информацию о структурах типа `TRUSTEE`, которые являются полями полученных структур типа `EXPLICIT_ACCESS`.

Листинг 41.7. Получение элементов из списка управления доступом DACL и получение информации из структуры типа `TRUSTEE`

```
#define _WIN32_WINNT 0x0500

#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>
#include <aclapi.h>
#include <sddl.h>

int main()
{
  wchar_t wchDirName[248];   // имя каталога

  PSECURITY_DESCRIPTOR lpSd; // указатель на дескриптор безопасности
  PACL lpDacl;               // указатель на список DACL
  PEXPLICIT_ACCESS lpEa;     // указатель на массив элементов типа
                             // EXPLICIT_ACCESS
  ULONG ulCount;     // количество элементов в массиве

  LPTSTR  lpStringSid = NULL;    // указатель на строку с SID
```

```
DWORD dwErrCode;   // код возврата

// читаем имя файла
printf("Input a file name: ");
wscanf(L"%s", wchDirName);     // вводим имя файла

// получаем SD этого файла
dwErrCode = GetNamedSecurityInfo(
  wchDirName,      // имя файла
  SE_FILE_OBJECT,  // объект файл
  DACL_SECURITY_INFORMATION,   // получаем DACL
  NULL,          // адрес указателя на SID владельца
  NULL,          // адрес указателя на первичную группу
  &lpDacl,       // указатель на DACL
  NULL,          // указатель на SACL не нужен
  &lpSd);        // адрес указателя на дескриптор безопасности
if (dwErrCode != ERROR_SUCCESS)
{
  printf("Get named security info failed.\n");
  printf("Error code: %u\n", dwErrCode);

  return dwErrCode;
}
// читаем элементы из списка DACL
dwErrCode = GetExplicitEntriesFromAcl(
  lpDacl,      // адрес списка DACL
  &ulCount,    // адрес для количества элементов
  &lpEa);      // адрес указателя на буфер
if (dwErrCode != ERROR_SUCCESS)
{
  printf("Get explicit entries from ACL failed.\n");
  printf("Error code: %u\n", dwErrCode);

  return dwErrCode;
}

printf("Number of elements in DACL = %u\n", ulCount);

// получаем информацию из структур типа TRUSTEE
```

```
  for (unsigned i = 0; i < ulCount; ++i)
  {
    if (GetTrusteeForm(&(lpEa->Trustee)) == TRUSTEE_IS_NAME)
      wprintf(L"Trustee name: %s", GetTrusteeName(&(lpEa->Trustee)));
    if (GetTrusteeForm(&(lpEa->Trustee)) == TRUSTEE_IS_SID)
    {
      // преобразуем SID в строку
      if (!ConvertSidToStringSid(lpEa->Trustee.ptstrName, &lpStringSid))
      {
        printf("Convert SID to string SID failed.");

        return GetLastError();
      }
      // распечатываем SID
      wprintf(L"Trustee SID: %s\n", lpStringSid);

      // освобождаем память, занятую строкой
      LocalFree(lpStringSid);
    }

    ++lpEa;    // продвигаемся по массиву
  }

  // освобождаем память
  LocalFree(lpSd);
  LocalFree(lpEa);

  return 0;
}
```

41.9. Получение прав доступа из списка управления доступом

Не все права доступа, установленные для субъекта в разрешающем элементе списка управления доступом DACL, который соответствует этому субъекту, могут быть субъектом реализованы. Некоторые из этих прав могут быть установлены также и в запрещающем элементе списка управления доступом, который соответствует этому же субъекту. Так как запрещающие элементы списка DACL имеют приоритет над разрешающими элементами этого списка,

то в последних могут быть установлены права, которые пользователь не может реализовать. Под *эффективными правами доступа* субъекта к объекту понимаются такие права, которые субъект действительно может реализовать при доступе к объекту.

Для получения эффективных прав доступа к объекту заданной учетной записи используется функция `GetEffectiveRightsFromAcl`, которая имеет следующий прототип:

```
DWORD  GetEffectiveRightsFromAcl(
  PACL      pAcl,         // указатель на список управления доступом
  PTRUSTEE  pTrustee,     // указатель на структуру TRUSTEE
  PACCESS_MASK  pAccessRights  // указатель на маску доступа
);
```

В случае успешного завершения функция возвращает значение `ERROR_SUCCESS`, а в случае неудачи — код ошибки.

Параметр `pAcl` должен указывать на список управления доступом, из которого извлекается информация об эффективных правах доступа учетной записи к объекту.

Параметр `pTrustee` должен указывать на структуру типа `TRUSTEE`, для которой определяются эффективные права доступа к объекту. Структура `TRUSTEE` может быть инициализирована именем или идентификатором безопасности учетной записи.

Параметр `pAccessRights` должен указывать на переменную типа `ACCESS_MASK`, в которую функция запишет эффективные права доступа для структуры `TRUSTEE`, заданной параметром `pTrustee`.

В листинге 41.8 приведена программа, в которой для определения эффективных прав доступа пользователя к файлу или каталогу используется функция `GetEffectiveRightsFromAcl`.

Листинг 41.8. Получение элементов из списка управления доступом DACL и получение информации из структуры типа `TRUSTEE`

```
#define _WIN32_WINNT 0x0500

#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>
```

```
#include <lm.h>
#include <aclapi.h>

int main()
{
  TRUSTEE  Trustee;            // информация об учетной записи
  wchar_t wchFileName[248];    // имя файла или каталога
  wchar_t wchAccountName[UNLEN + 255];   // имя учетной записи

  PSECURITY_DESCRIPTOR lpSd;   // указатель на дескриптор безопасности
  PACL lpDacl;                 // указатель на список DACL

  ACCESS_MASK dwAccessRights = 0;  // права доступа
  DWORD dwErrCode;      // код возврата

  // читаем имя файла
  printf("Input a file name: ");
  wscanf(L"%s", wchFileName);      // вводим имя файла или каталога

  // получаем SD этого файла
  dwErrCode = GetNamedSecurityInfo(
    wchFileName,     // имя файла
    SE_FILE_OBJECT,  // объект файл
    DACL_SECURITY_INFORMATION,   // получаем DACL
    NULL,         // адрес указателя на SID владельца
    NULL,         // адрес указателя на первичную группу
    &lpDacl,      // указатель на DACL
    NULL,         // указатель на SACL не нужен
    &lpSd);       // адрес указателя на дескриптор безопасности
  if (dwErrCode != ERROR_SUCCESS)
  {
    printf("Get named security info failed.\n");
    printf("Error code: %u\n", dwErrCode);

    return dwErrCode;
  }

  // вводим имя домена и пользователя
  printf("Input domain account (domain_name\\user_name): ");
  wscanf(L"%s", wchAccountName);       // вводим имя учетной записи
```

```
  // строим структуру TRUSTEE по имени
  BuildTrusteeWithName(&Trustee, wchAccountName);

  // получаем права доступа для заданного имени
  dwErrCode = GetEffectiveRightsFromAcl(
    lpDacl,      // указатель на список DACL
    &Trustee,    // адрес структуры TRUSTEE
    &dwAccessRights);  // адрес маски с флагами
  if (dwErrCode != ERROR_SUCCESS)
  {
    printf("Get effective rights from ACL failed.\n");
    printf("Error code: %u\n", dwErrCode);

    return dwErrCode;
  }

  // выводим на консоль права доступа
  printf("ACCESS_MASK: %x\n", dwAccessRights);

  // отображаем права доступа
  if ((dwAccessRights & SPECIFIC_RIGHTS_ALL) == SPECIFIC_RIGHTS_ALL)
    printf("SPECIFIC_RIGHTS_ALL is set.\n");
  else
    printf("SPECIFIC_RIGHTS_ALL is not set.\n");

  if ((dwAccessRights & STANDARD_RIGHTS_ALL) == STANDARD_RIGHTS_ALL)
    printf("STANDARD_RIGHTS_ALL is set.\n");
  else
    printf("STANDARD_RIGHTS_ALL is not set.\n");

  if ((dwAccessRights & GENERIC_ALL) == GENERIC_ALL)
    printf("GENERIC_ALL is set.\n");
  else
    printf("GENERIC_ALL is not set.\n");

  // освобождаем память
  LocalFree(lpSd);

  return 0;
}
```

41.10. Получение из списка управления доступом прав, которые подвергаются аудиту

Для получения из списка управления доступом SACL прав субъекта на доступ к объекту, которые подвергаются аудиту со стороны системы безопасности, используется функция GetAuditedPermissionsFromAcl, которая имеет следующий прототип:

```
DWORD  GetAuditedPermissionsFromAcl(
  PACL  pAcl,          // адрес списка SACL
  PTRUSTEE  pTrustee,  // адрес структуры TRUSTEE
  PACCESS_MASK  pSuccessfulAuditedRights,  // маска удачного доступа
  PACCESS_MASK  pFailedAuditRights         // маска неудачного доступа
);
```

В случае успешного завершения функция возвращает значение ERROR_SUCCESS, а в случае неудачи — код ошибки.

Параметр pAcl должен указывать на список управления доступом, из которого извлекается информация о правах доступа учетной записи к объекту, которые подвергаются аудиту со стороны системы безопасности.

Параметр pTrustee должен указывать на структуру типа TRUSTEE, для которой определяются права доступа к объекту, подверженные аудиту. Структура TRUSTEE может быть инициализирована именем или идентификатором безопасности учетной записи.

Параметр pSuccessfulAuditedRights должен указывать на переменную типа ACCESS_MASK, в которую функция запишет подвергаемые аудиту права доступа для структуры TRUSTEE, заданной параметром pTrustee. Причем для этих прав генерируется аудиторское сообщение только в случае успешного разрешения доступа к объекту.

Параметр pFailedAuditRights должен указывать на переменную типа ACCESS_MASK, в которую функция запишет подвергаемые аудиту права доступа для структуры TRUSTEE, заданной параметром pTrustee. Причем для этих прав генерируется аудиторское сообщение только в случае неудачной попытки получения доступа к объекту.

В листинге 41.9 приведена программа, определяющая права доступа пользователя к файлу или каталогу, которые подвергаются аудиту, используя для этого функцию GetAuditedPermissionsFromAcl. Как и для всех программ, работающих со списком управления доступом SACL, требуется, чтобы пользователь имел действующую привилегию SE_SECURITY_NAME. Такую привилегию имеют администраторы системы, но по умолчанию она недействительна. Поэтому, прежде чем получать информацию из списка SACL объекта, эта привилегия будет активизирована посредством вызова функции

AdjustTokenPrivileges. Более подробно работа с привилегиями рассмотрена в *гл. 43*.

Листинг 41.9. Получение информации о правах доступа пользователя к объекту, которые подвергаются аудиту

```
#ifndef UNICODE
#define UNICODE
#endif

#define _WIN32_WINNT 0x0500

#include <stdio.h>
#include <windows.h>
#include <lm.h>
#include <aclapi.h>
#include <sddl.h>

int main()
{
  HANDLE hProcess;       // дескриптор процесса
  HANDLE hTokenHandle;   // дескриптор маркера доступа

  TOKEN_PRIVILEGES tp;   // привилегии маркера доступа

  DWORD  dwLengthOfSID = 0;          // длина SID
  DWORD  dwLengthOfDomainName = 0;   // длина имени домена
  DWORD  dwLengthOfUserName = UNLEN; // длина имени учетной записи
  SID    *lpSid = NULL;              // указатель на SID

  LPTSTR  lpDomainName = NULL;   // указатель на имя домена
  SID_NAME_USE type_of_SID;      // тип учетной записи

  wchar_t wchFileName[248];      // имя каталога
  wchar_t wchUserName[UNLEN];    // имя учетной записи

  TRUSTEE  Trustee;      // информация об учетной записи
  PACL pSacl = NULL;     // указатель на список SACL
  PSECURITY_DESCRIPTOR pSd = NULL; // указатель на дескриптор безопасности
```

```
ACCESS_MASK amSuccess = 0;      // маска для аудита успешного доступа
ACCESS_MASK amFailed = 0;       // маска для аудита неудачного доступа

DWORD dwRetCode;        // код возврата

// получаем дескриптор процесса
hProcess = GetCurrentProcess();

// получаем маркер доступа процесса
if (!OpenProcessToken(
  hProcess,           // дескриптор процесса
  TOKEN_ALL_ACCESS,  // полный доступ к маркеру доступа
  &hTokenHandle))     // дескриптор маркера
{
  dwRetCode = GetLastError();
  printf( "Open process token failed: %u\n", dwRetCode);

  return dwRetCode;
}

// устанавливаем общее количество привилегий
tp.PrivilegeCount = 1;

// определяем идентификатор привилегии для установки аудита
if (!LookupPrivilegeValue(
  NULL,       // ищем идентификатор привилегии на локальном компьютере
  SE_SECURITY_NAME,  // привилегия для аудита
  &(tp.Privileges[0].Luid)))
{
  dwRetCode = GetLastError();
  printf("Lookup privilege value failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// разрешаем привилегию аудита
tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

// разрешаем привилегию для установки аудита
```

```
if (!AdjustTokenPrivileges(
  hTokenHandle,  // дескриптор маркера доступа процесса
  FALSE,   // не запрещаем все привилегии
  &tp,     // адрес привилегий
  0,       // длины буфера нет
  NULL,    // предыдущее состояние привилегий не нужно
  NULL))   // длина буфера не нужна
{
  dwRetCode = GetLastError();
  printf("Lookup privilege value failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// вводим имя файла или каталога, например: C:\\test.txt
printf("Input a full path to your file: ");
_getws(wchFileName);

// получаем дескриптор безопасности файла
dwRetCode = GetNamedSecurityInfo(
  wchFileName,     // имя файла
  SE_FILE_OBJECT,  // объект файл
  SACL_SECURITY_INFORMATION,   // получаем сведения об аудите
  NULL,      // SID владельца не нужен
  NULL,      // SID первичной группы не нужен
  NULL,      // DACL не нужен
  &pSacl,    // указатель на SACL
  &pSd);     // адрес указателя на дескриптор безопасности
if (dwRetCode != ERROR_SUCCESS)
{
  printf("Get named security info failed.\n");
  printf("Error code: %u\n", dwRetCode);

  return dwRetCode;
}

// вводим имя пользователя
printf("Input user_name: ");
```

```
_getws(wchUserName);        // вводим имя учетной записи

// получаем SID учетной записи
// определяем длину SID пользователя
LookupAccountName(
  NULL,             // ищем имя на локальном компьютере
  wchUserName,      // имя пользователя
  NULL,             // определяем длину SID
  &dwLengthOfSID,   // длина SID
  NULL,             // определяем имя домена
  &dwLengthOfDomainName,    // длина имени домена
  &type_of_SID);    // тип учетной записи

// проверяем, вернула ли функция длину SID
if (dwLengthOfSID == 0)
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// распределяем память для SID и имени домена
lpSid = (SID*) new char[dwLengthOfSID];
lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];

// определяем SID и имя домена пользователя
if(!LookupAccountName(
  NULL,             // ищем имя на локальном компьютере
  wchUserName,      // имя пользователя
  lpSid,            // указатель на SID
  &dwLengthOfSID,   // длина SID
  lpDomainName,     // указатель на имя домена
  &dwLengthOfDomainName,    // длина имени домена
  &type_of_SID))    // тип учетной записи
{
  dwRetCode = GetLastError();
```

```
  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// строим структуру TRUSTEE по идентификатору безопасности
BuildTrusteeWithSid(&Trustee, lpSid);

// получаем аудиторские права пользователя
dwRetCode = GetAuditedPermissionsFromAcl(
  pSacl,       // адрес списка SACL
  &Trustee,    // информация об учетной записи
  &amSuccess,  // адрес маски прав успешного доступа
  &amFailed);  // адрес маски прав неудачного доступа
if (dwRetCode != ERROR_SUCCESS)
{
  printf("Get audited permissions from ACL failed.\n");
  printf("Error code: %u\n", dwRetCode);

  return dwRetCode;
}

// распечатываем права доступа
printf("Success audit rights: %x\n", amSuccess);
printf("Failed audit rights: %x\n", amFailed);

// запрещаем привилегию аудита
tp.Privileges[0].Attributes = 0;

// разрешаем привилегию для установки аудита
if (!AdjustTokenPrivileges(
  hTokenHandle,    // дескриптор маркера доступа процесса
  FALSE,     // не запрещаем все привилегии
  &tp,       // адрес привилегий
  0,         // длины буфера нет
  NULL,      // предыдущее состояние привилегий не нужно
  NULL))     // длина буфера не нужна
{
```

```
    dwRetCode = GetLastError();
    printf("Lookup privilege value failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }

  // освобождаем память для дескриптора
  LocalFree(pSd);
  delete[] lpDomainName;
  delete[] lpSid;

  return 0;
}
```

В связи с работой функции `GetAuditedPermissionsFromAcl` сделаем следующее замечание. Похоже, что в операционной системе Windows 2000 функция возвращает права доступа, для которых генерируется аудиторское сообщение в случае успешного доступа к объекту в переменную, на которую указывает параметр `pFailedAuditRights`. И, наоборот, в переменную, на которую указывает параметр `pSuccessfulAuditedRights`, функция записывает права доступа, для которых генерируется аудиторское сообщение в случае неудачного доступа к объекту. То есть или при реализации функции `GetAuditedPermissionsFromAcl` перепутаны параметры `pSuccessfulAuditedRights` и `pFailedAuditRights`, или эти же параметры перепутаны при описании функции в документации.

Глава 42

Работа с привилегиями

42.1. Локальные идентификаторы привилегий

Привилегии, которыми могут обладать учетные записи, были рассмотрены в *разд. 36.8*, посвященном маркерам доступа. Назначаются привилегии при создании учетных записей. При создании маркера доступа он получает привилегии той учетной записи, которая его создала. Далее приведены символические имена для привилегий, которые используются в операционных системах Windows NT.

- `SE_ASSIGNPRIMARYTOKEN_NAME` — SeAssignPrimaryTokenPrivilege;
- `SE_AUDIT_NAME` — SeAuditPrivilege;
- `SE_BACKUP_NAME` — SeBackupPrivilege;
- `SE_CHANGE_NOTIFY_NAME` — SeChangeNotifyPrivilege;
- `SE_CREATE_PAGEFILE_NAME` — SeCreatePagefilePrivilege;
- `SE_CREATE_PERMANENT_NAME` — SeCreatePermanentPrivilege;
- `SE_CREATE_TOKEN_NAME` — SeCreateTokenPrivilege;
- `SE_DEBUG_NAME` — SeDebugPrivilege;
- `SE_INC_BASE_PRIORITY_NAME` — SeIncreaseBasePriorityPrivilege;
- `SE_INCREASE_QUOTA_NAME` — SeIncreaseQuotaPrivilege;
- `SE_LOAD_DRIVER_NAME` — SeLoadDriverPrivilege;
- `SE_LOCK_MEMORY_NAME` — SeLockMemoryPrivilege;
- `SE_MACHINE_ACCOUNT_NAME` — SeMachineAccountPrivilege;
- `SE_PROF_SINGLE_PROCESS_NAME` — SeProfileSingleProcessPrivilege;

- SE_REMOTE_SHUTDOWN_NAME — SeRemoteShutdownPrivilege;
- SE_RESTORE_NAME — SeRestorePrivilege;
- SE_SECURITY_NAME — SeSecurityPrivilege;
- SE_SHUTDOWN_NAME — SeShutdownPrivilege;
- SE_SYSTEM_ENVIRONMENT_NAME — SeSystemEnvironmentPrivilege;
- SE_SYSTEM_PROFILE_NAME — SeSystemProfilePrivilege;
- SE_SYSTEMTIME_NAME — SeSystemtimePrivilege;
- SE_TAKE_OWNERSHIP_NAME — SeTakeOwnershipPrivilege;
- SE_TCB_NAME — SeTcbPrivilege;
- SE_UNSOLICITED_INPUT_NAME — SeUnsolicitedInputPrivilege.

При работе с привилегиями используются специальные числовые значения, которые называются локальными идентификаторами. Локальные идентификаторы действительны только на локальной машине от момента запуска системы и до момента ее выключения. При последующих запусках системы локальные идентификаторы, используемые системой, генерируются повторно. Так как локальные идентификаторы генерируются локально, то на каждой машине для идентификации одинаковых объектов используются различные локальные идентификаторы. Например, для обозначения одной и той же привилегии на разных машинах могут использоваться различные локальные идентификаторы.

Локальные идентификаторы определяются структурами следующего типа:

```
typedef struct  _LUID {
  DWORD  LowPart;    // младшая часть
  LONG   HighPart;   // старшая часть
}  LUID, *PLUID;
```

При работе с локальными идентификаторами обычно используются структуры типа:

```
typedef struct  _LUID_AND_ATTRIBUTES {
  LUID   Luid;          // локальный идентификатор
  DWORD  Attributes;    // атрибуты локального идентификатора
}  LUID_AND_ATTRIBUTES, * PLUID_AND_ATTRIBUTES;
```

Здесь поле `Attributes` используется для установки различных флагов, характеризующих локальный идентификатор, а, следовательно, и объект, который этот локальный идентификатор представляет.

В последующих разделах будут рассмотрены функции для работы с локальными идентификаторами, которые представляют привилегии. Но в общем случае локальные идентификаторы могут использоваться и для других целей.

42.2. Инициализация локального идентификатора

Для инициализации локального идентификатора используется функция `AllocateLocallyUniqueId`, которая имеет следующий прототип:

```
BOOL  AllocateLocallyUniqueId(
  PLUID  Luid      // указатель на локальный идентификатор
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`. Единственный параметр функции должен указывать на структуру типа `LUID`, описанную в *разд. 42.1*, в которую функция запишет сгенерированное значение локального идентификатора.

В листинге 42.1 приведена программа, которая создает новый локальный идентификатор посредством вызова функции `AllocateLocallyUniqueId`.

Листинг 42.1. Инициализация локального идентификатора

```
#include <windows.h>
#include <stdio.h>

int main()
{
  LUID  luId;         // локальный идентификатор

  DWORD dwRetCode;    // код возврата

  // распределяем локальный уникальный идентификатор
  if (!AllocateLocallyUniqueId(
    &luId))       // адрес локального идентификатора
  {
    dwRetCode = GetLastError();
    perror("Lookup privilege value failed.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  // распечатываем LUID
```

```
  printf("Locally unique identifier.\n");
  printf("\tHigh part: %x, Low part: %x\n", luId.HighPart, luId.LowPart);

  return 0;
}
```

42.3. Получение локального идентификатора привилегии

Для получения локального идентификатора привилегии по имени привилегии используется функция `LookupPrivilegeValue`, которая имеет следующий прототип:

```
BOOL  LookupPrivilegeValue(
  LPCWSTR  lpSystemName,     // имя системы
  LPCWSTR  lpName,           // имя привилегии
  PLUID    lpLuid            // адрес локального идентификатора
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`.

Параметр `lpSystemName` должен указывать на имя компьютера, на котором ищется локальный идентификатор привилегии. Если в этом параметре установлено значение `NULL`, то локальный идентификатор привилегии ищется на локальном компьютере.

Параметр `lpName` должен указывать на имя привилегии, для которой ищется ее локальный идентификатор.

Параметр `lpLuid` должен указывать на структуру типа `LUID`, в которую функция поместит локальный идентификатор привилегии.

Программа, в которой локальный идентификатор привилегии определяется посредством вызова функции `LookupPrivilegeValue`, приведена в листинге 42.2.

42.4. Получение имени привилегии

Для определения имени привилегии по ее локальному идентификатору используется функция `LookupPrivilegeName`, которая имеет следующий прототип:

```
BOOL  LookupPrivilegeName(
  LPCWSTR  lpSystemName,     // имя системы
  PLUID    lpLuid,           // адрес локального идентификатора
```

```
    LPWSTR   lpName,          // имя привилегии
    LPDWORD  cbName           // длина имени привилегии
  );
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError.

Параметры функции имеют следующее назначение.

Параметр lpSystemName должен указывать на имя компьютера, на котором ищется имя привилегии, соответствующей локальному идентификатору, на который указывает параметр lpLuid. Если в этом параметре установлено значение NULL, то имя привилегии ищется на локальном компьютере.

Параметр lpLuid должен указывать на структуру типа LUID, которая содержит локальный идентификатор привилегии, имя которой определяется.

Параметр lpName должен указывать на буфер памяти, в который функция запишет имя привилегии.

Параметр cbName должен указывать на переменную типа DWORD, которая содержит длину буфера для имени привилегии. Если эта длина меньше необходимой, то функция закончится неудачей и в эту переменную будет записана требуемая длина буфера для имени привилегии.

В листинге 42.2 приведена программа, которая сначала определяет локальный идентификатор привилегии, используя для этого функцию LookupPrivilegeValue, а затем определяет имя привилегии по ее локальному идентификатору, используя для этого функцию LookupPrivilegeName.

Листинг 42.2. Определение локального идентификатора и имени привилегии

```
#include <windows.h>
#include <stdio.h>

int main()
{
  LUID  luId;                 // локальный идентификатор для привилегии
  DWORD dwLength = 0;         // длина имени привилегии
  LPTSTR lpPrivName = NULL;   // адрес имени привилегии

  DWORD dwRetCode;            // код возврата

  // определяем локальный идентификатор привилегии
  if (!LookupPrivilegeValue(
```

```
  NULL,             // ищем привилегии на локальной машине
  SE_DEBUG_NAME,    // привилегия выполнять отладку процесса
  &luId))           // адрес локального идентификатора привилегии
{
  dwRetCode = GetLastError();
  perror("Lookup privilege value failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// распечатываем LUID
printf("Luid of SE_DEBUG_NAME privilege. ");
printf("High part: %x, Low part: %x\n", luId.HighPart, luId.LowPart);

// определяем длину имени привилегии
if (!LookupPrivilegeName(
  NULL,           // локальная машина
  &luId,          // адрес локального уникального идентификатора
  lpPrivName,     // адрес имени привилегии
  &dwLength))     // адрес длины буфера
{
  dwRetCode = GetLastError();
  if (dwRetCode != ERROR_INSUFFICIENT_BUFFER)
  {
    perror("Lookup privilege name failed.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  // захватываем память для имени привилегии
  lpPrivName = new CHAR[dwLength + 1];
}

// определяем имя привилегии
if (!LookupPrivilegeName(
  NULL,           // локальная машина
  &luId,          // адрес локального уникального идентификатора
```

```
    lpPrivName,     // адрес имени привилегии
    &dwLength))     // адрес длины буфера
  {
    dwRetCode = GetLastError();
    perror("Lookup privilege name failed.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  // распечатываем имя привилегии
  printf("Name of the Luid: %s\n", lpPrivName);

  delete[] lpPrivName;

  return 0;
}
```

42.5. Получение имени привилегии для отображения

Для определения имени привилегии, которое имеет вид, пригодный для чтения непрофессиональным пользователем, используется функция `LookupPrivilegeDisplayName`, которая имеет следующий прототип:

```
BOOL  LookupPrivilegeDisplayName(
  LPCSTR   lpSystemName,     // имя системы
  LPCSTR   lpName,           // имя привилегии
  LPSTR    lpDisplayName,    // имя привилегии для отображения
  LPDWORD  cbDisplayName,    // длина имени привилегии для отображения
  LPDWORD  lpLanguageId      // идентификатор языка
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`.

Параметр `lpSystemName` должен указывать на имя компьютера, на котором ищется имя привилегии для отображения, соответствующее имени привилегии, на которое указывает параметр `lpName`. Если в этом параметре установлено значение `NULL`, то имя привилегии ищется на локальном компьютере.

Параметр `lpName` должен указывать имя привилегии, для которого определяется удобное для отображения имя.

Параметр `lpDisplayName` должен указывать на буфер, в который функция запишет удобное для отображения имя привилегии.

Параметр `cbDisplayName` должен указывать на переменную типа `DWORD`, которая содержит длину буфера для удобного для отображения имени привилегии. Если эта длина меньше необходимой, то функция закончится неудачей и в эту переменную будет записана требуемая длина буфера для удобного для отображения имени привилегии.

Параметр `lpLanguageId` должен указывать на переменную типа `DWORD`, в которую функция запишет идентификатор языка, используемого при написании имени привилегии, удобного для отображения.

В листинге 42.3 приведена программа, которая определяет удобное для отображения имя привилегии по ее внутреннему имени, используя для этого функцию `LookupPrivilegeDisplayName`.

Листинг 42.3. Определение удобного для отображения имени привилегии

```
#include <windows.h>
#include <stdio.h>

int main()
{
  LPTSTR lpPrivDisplayName = NULL;    // адрес имени привилегии
                                      // для отображения

  DWORD dwLength = 0;  // длина имени привилегии
  DWORD dwLangId;      // идентификатор языка

  DWORD dwRetCode;     // код возврата


  // определяем длину имени привилегии для отображения
  if (!LookupPrivilegeDisplayName (
    NULL,              // локальная машина
    SE_SHUTDOWN_NAME,  // имя привилегии
    lpPrivDisplayName,     // адрес для имени привилегии
    &dwLength,         // адрес длины буфера
    &dwLangId))        // адрес идентификатора языка
  {
    dwRetCode = GetLastError();
    if (dwRetCode != ERROR_INSUFFICIENT_BUFFER)
```

```
    {
      perror("Lookup privilege display name failed.\n");
      printf("The last error code: %u\n", dwRetCode);

      return dwRetCode;
    }

    // захватываем память для имени привилегии
    lpPrivDisplayName = new char[dwLength + 1];
  }

  // определяем имя привилегии для отображения
  if (!LookupPrivilegeDisplayName (
    NULL,              // локальная машина
    SE_SHUTDOWN_NAME,  // имя привилегии
    lpPrivDisplayName,     // адрес для имени привилегии
    &dwLength,         // адрес длины буфера
    &dwLangId))        // адрес идентификатора языка
  {
    dwRetCode = GetLastError();
    perror("Lookup privilege display name failed.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  // распечатываем имя привилегии и идентификатор языка
  printf("Display name of the privilege SE_SHUTDOWN_NAME:\n\t%s\n",
lpPrivDisplayName);
  printf("Language identifier: %x\n", dwLangId);

  delete[] lpPrivDisplayName;

  return 0;
}
```

Глава 43

Работа с маркерами доступа

43.1. Открытие маркера доступа процесса

Прежде чем с маркером доступа можно будет производить какие-либо действия, его нужно открыть. Для открытия маркера доступа процесса используется функция `OpenProcessToken`, которая имеет следующий прототип:

```
BOOL  OpenProcessToken(
  HANDLE  ProcessHandle,   // дескриптор процесса
  DWORD   DesiredAccess,   // режимы доступа к маркеру доступа процесса
  PHANDLE TokenHandle      // адрес маркера доступа процесса
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`.

Параметр `ProcessHandle` должен содержать дескриптор процесса, маркер доступа которого открывается.

В параметре `DesiredAccess` должны быть установлены флаги, отмечающие затребованные права доступа к маркеру доступа процесса. Ниже перечислены флаги, которые можно установить в этом поле маркера доступа.

Специфические для маркера доступа права доступа устанавливаются следующим флагами:

- `TOKEN_ASSIGN_PRIMARY` — право присоединить первичный маркер доступа к процессу;
- `TOKEN_DUPLICATE` — право дублировать маркер доступа;
- `TOKEN_IMPERSONATE` — право замещать маркер доступа процесса;
- `TOKEN_QUERY` — право получать информацию из маркера доступа;

- `TOKEN_QUERY_SOURCE` — право получить информацию об источнике маркера доступа;
- `TOKEN_ADJUST_PRIVILEGES` — право настраивать привилегии маркера доступа;
- `TOKEN_ADJUST_GROUPS` — право настраивать свойства групп в маркере доступа;
- `TOKEN_ADJUST_DEFAULT` — право изменять информацию для установки в дескриптор объекта по умолчанию;
- `TOKEN_ADJUST_SESSIONID` — право настраивать идентификатор сессии в маркере доступа.

Для стандартных прав доступа можно установить следующие флаги:

- `READ_CONTROL` — право читать информацию из маркера доступа, за исключением информации из списка SACL;
- `WRITE_DAC` — право модифицировать список DACL в маркере доступа;
- `WRITE_OWNER` — право изменить владельца маркера доступа;
- `DELETE` — право удалять маркер доступа.

Кроме того, можно использовать следующие символические константы, применяемые для обозначения комбинации нескольких различных флагов доступа:

- `TOKEN_ALL_ACCESS` — включает все права доступа к маркеру доступа;
- `TOKEN_READ` — включает права доступа `STANDARD_RIGHTS_READ` и `TOKEN_QUERY`;
- `TOKEN_WRITE` — включает права `STANDARD_RIGHTS_WRITE`, `TOKEN_ADJUST_PRIVILEGES`, `TOKEN_ADJUST_GROUPS` и `TOKEN_ADJUST_DEFAULT`;
- `TOKEN_EXECUTE` — включат право `STANDARD_RIGHTS_EXECUTE`.

Для доступа к списку управления доступом SACL маркера доступа нужно установить флаг `ACCESS_SYSTEM_SECURITY` — доступ к списку управления доступом SACL.

Теперь перейдем к последнему параметру функции `OpenProcessToken` — `TokenHandle`. Этот параметр должен указывать на переменную типа `HANDLE`, в которую функция запишет дескриптор маркера доступа процесса. Так как этот дескриптор является обычным дескриптором, описывающим объект ядра операционной системы, то после завершения работы с маркером доступа этот дескриптор нужно закрыть при помощи функции `CloseHandle`.

Примеры использования функции `OpenProcessToken` будут приведены в следующих параграфах, посвященных работе с маркерами доступа.

43.2. Открытие маркера доступа потока

Прежде чем работать с маркером доступа потока его необходимо открыть при помощи функции `OpenThreadToken`, которая имеет следующий прототип:

```
BOOL  OpenThreadToken(
  HANDLE  ThreadHandle,    // дескриптор потока
  DWORD   DesiredAccess,   // режимы доступа к маркеру доступа потока
  BOOL    OpenAsSelf,      // выбор контекста безопасности при открытии
  PHANDLE TokenHandle      // адрес маркера доступа потока
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`.

Параметр `ThreadHandle` должен содержать дескриптор потока, маркер доступа которого открывается.

В параметре `DesiredAccess` должны быть установлены флаги, отмечающие затребованные права доступа к маркеру доступа потока. Флаги, которые можно установить в этом поле маркера доступа, перечислены в предыдущем разделе при рассмотрении соответствующего параметра функции `OpenProcessToken`.

В параметре `OpenAsSelf` должно быть установлено одно из значений `FALSE` или `TRUE`. Если в параметре установлено значение `FALSE`, то функция работает в контексте безопасности потока, вызвавшего функцию. Это дает возможность исполнять функцию в контексте процесса клиента, если данный поток запущен на сервере от имени клиента. Если же в параметре установлено значение `TRUE`, то функция работает в контексте безопасности процесса, вызвавшего функцию.

Параметр `TokenHandle` должен указывать на переменную типа `HANDLE`, в которую функция запишет дескриптор маркера доступа потока. Так как этот дескриптор является обычным дескриптором, описывающим объект ядра операционной системы, то после завершения работы с маркером доступа этот дескриптор нужно закрыть при помощи функции `CloseHandle`.

Пример использования функции `OpenThreadToken` будет приведен в *разд. 43.11.*

43.3. Структуры, используемые для работы с маркером доступа

Программы не имеют непосредственного доступа к внутренней структуре маркера доступа. Поэтому при работе с маркерами доступа функции используют специальные структуры или перечисления, которые позволяют

получать информацию из маркера доступа или изменять требуемую информацию в маркере доступа. Далее приведены типы структур и перечислений, которые используются при работе с маркерами доступа.

Для получения информации о пользователе, связанном с заданным маркером доступа, используется структура типа:

```
typedef struct  _TOKEN_USER {
  SID_AND_ATTRIBUTES  User;     // SID пользователя и атрибуты
}  TOKEN_USER, *PTOKEN_USER;
```

Структура типа `SID_AND_ATTRIBUTES` определена следующим образом:

```
typedef struct  _SID_AND_ATTRIBUTES {
  PSID   Sid;             // указатель на SID пользователя
  DWORD  Attributes;      // атрибуты SID
}  SID_AND_ATTRIBUTES, * PSID_AND_ATTRIBUTES;
```

Атрибуты идентификатора безопасности задаются флагами, каждый из которых определяет некоторое свойство идентификатора безопасности. В настоящее время для идентификатора безопасности пользователя не определены атрибуты, поэтому это поле не используется.

Для получения информации о группах, связанных с заданным маркером доступа, используется структура типа:

```
typedef struct  _TOKEN_GROUPS {
  DWORD  GroupCount;            // количество групп
  SID_AND_ATTRIBUTES  Groups[ANYSIZE_ARRAY];   // SID и атрибуты групп
}  TOKEN_GROUPS, *PTOKEN_GROUPS;
```

В поле `Attributes` структуры типа `SID_AND_ATTRIBUTES` могут быть установлены следующие флаги:

- `SE_GROUP_MANDATORY` — нельзя сбросить флаг `SE_GROUP_ENABLED`;
- `SE_GROUP_ENABLED_BY_DEFAULT` — флаг проверки доступа установлен по умолчанию;
- `SE_GROUP_ENABLED` — группа проверяется при доступе к объекту;
- `SE_GROUP_OWNER` — пользователь, связанный с маркером доступа, является владельцем группы; или группа может быть владельцем объекта;
- `SE_GROUP_USE_FOR_DENY_ONLY` — группа используется только при проверке запрещения доступа к объекту;
- `SE_GROUP_LOGON_ID` — `SID` идентифицирует сессию, выполнившую вход в систему для пользователя, связанного с группой;
- `SE_GROUP_RESOURCE` — локальная группа домена.

Напомним, что структура SID_AND_ATTRIBUTES была описана выше, при описании структуры типа TOKEN_USER.

Для получения информации о привилегиях, которыми обладает маркер доступа, используется структура типа:

```
typedef struct  _TOKEN_PRIVILEGES {
  DWORD  PrivilegeCount;
  LUID_AND_ATTRIBUTES  Privileges[ANYSIZE_ARRAY];
}  TOKEN_PRIVILEGES, *PTOKEN_PRIVILEGES;
```

В поле Attributes структуры типа LUID_AND_ATTRIBUTES могут быть установлены следующие флаги:

- SE_PRIVILEGE_ENABLED_BY_DEFAULT — привилегия установлена по умолчанию;
- SE_PRIVILEGE_ENABLED — привилегия установлена;
- SE_PRIVILEGE_USED_FOR_ACCESS — привилегия использовалась для доступа к объекту или сервису.

Напомним, что структура LUID_AND_ATTRIBUTES была описана в *разд. 42.1.*

Для получения информации об идентификаторе безопасности, который будет использоваться системой в алгоритме установки по умолчанию владельца нового объекта, создаваемого процессом с данным маркером доступа, используется структура типа:

```
typedef struct  _TOKEN_OWNER {
  PSID  Owner;        // SID владельца объекта по умолчанию
}  TOKEN_OWNER, *PTOKEN_OWNER;
```

Отметим, что в этом случае идентификатор безопасности должен быть идентификатором безопасности одного из пользователей или групп, связанных с данным маркером доступа.

Для получения информации об идентификаторе безопасности, который будет использоваться системой в алгоритме установки по умолчанию первичной группы нового объекта, создаваемого процессом с данным маркером доступа, используется структура типа:

```
typedef struct  _TOKEN_PRIMARY_GROUP {
  PSID  PrimaryGroup;       // SID первичной группы по умолчанию
}  TOKEN_PRIMARY_GROUP, *PTOKEN_PRIMARY_GROUP;
```

Отметим, что в этом случае идентификатор безопасности должен быть идентификатором безопасности одной из групп, связанных с данным маркером доступа.

Для получения информации о списке управления доступом DACL, который будет использоваться в алгоритме установки по умолчанию списка DACL

нового объекта, создаваемого процессом с данным маркером доступа, используется структура типа:

```
typedef struct  _TOKEN_DEFAULT_DACL {
  PACL  DefaultDacl;     // указатель на список DACL по умолчанию
} TOKEN_DEFAULT_DACL, *PTOKEN_DEFAULT_DACL;
```

Для получения информации об источнике, вызвавшем создание данного маркера доступа, используется структура типа:

```
typedef struct  _TOKEN_SOURCE {
  CHAR  SourceName[TOKEN_SOURCE_LENGTH];   // имя источника
  LUID  SourceIdentifier;      // локальный идентификатор источника
}  TOKEN_SOURCE, *PTOKEN_SOURCE;
```

Заметим, что поле `SourceName` является простым символьным массивом, содержащим имя источника маркера доступа.

Информация о типе маркера доступа представляется перечислимой константой следующего типа:

```
typedef enum  _TOKEN_TYPE {
  TokenPrimary = 1,      // первичный маркер доступа
  TokenImpersonation     // замещающий маркер доступа
}  TOKEN_TYPE;
```

Информация об уровне замещения первичного маркера доступа представляется перечислимой константой следующего типа:

```
typedef enum  _SECURITY_IMPERSONATION_LEVEL {
  SecurityAnonymous,         // анонимный уровень замещения
  SecurityIdentification,    // идентифицирующий уровень замещения
  SecurityImpersonation,     // уровень подмены контекста безопасности
  SecurityDelegation         // уровень делегирования полномочий
}  SECURITY_IMPERSONATION_LEVEL, * PSECURITY_IMPERSONATION_LEVEL;
```

Более подробно об уровнях замещения маркеров доступа рассказано в *разд. 36.8*.

Для получения статистических данных о маркере доступа используется структура следующего типа:

```
typedef struct  _TOKEN_STATISTICS {
  LUID  TokenId;           // локальный идентификатор маркера доступа
  LUID  AuthenticationId;  // локальный идентификатор сессии,
                           // которую представляет маркер доступа
  LARGE_INTEGER  ExpirationTime;   // не поддерживается
  TOKEN_TYPE  TokenType;           // тип маркера доступа
  SECURITY_IMPERSONATION_LEVEL  ImpersonationLevel;// уровень замещения
```

```
    DWORD  DynamicCharged;   // объем буфера, используемого для хранения
                             // информации о защите по умолчанию и идентификаторе
                             // безопасности первичной группы маркера доступа
    DWORD  DynamicAvailable;   // объем свободной памяти в буфере
    DWORD  GroupCount;  // количество групп, связанных с маркером доступа
    DWORD  PrivilegeCount;      // количество привилегий, доступных в
                                // маркере доступа
    LUID   ModifiedId;  // локальный идентификатор, который изменяется
                        // каждый раз при модификации маркера доступа
} TOKEN_STATISTICS, *PTOKEN_STATISTICS;
```

Для получения информации об ограничивающих идентификаторах безопасности используется структура типа `TOKEN_GROUPS`, которая была описана ранее.

Для получения информации об идентификаторе сессии, в которой был создан маркер доступа, используется переменная типа `DWORD`. Этот идентификатор используется только в том случае, если маркер доступа был создан для терминальной сессии. В остальных случаях значение этого идентификатора равно 0.

43.4. Получение информации из маркера доступа

Для получения информации из маркера доступа используется функция `GetTokenInformation`, которая имеет следующий прототип:

```
BOOL  GetTokenInformation(
  HANDLE  TokenHandle,       // дескриптор маркера доступа
  TOKEN_INFORMATION_CLASS  TokenInformationClass,  // тип информации
  LPVOID  TokenInformation,  // указатель на буфер для информации
  DWORD   TokenInformationLength,    // длина буфера
  PDWORD  ReturnLength       // требуемая длина буфера
);
```

В случае удачного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного завершения функции можно получить посредством вызова функции `GetLastError`. Параметры функции `GetTokenInformation` имеют следующее назначение.

Параметр `TokenHandle` должен содержать дескриптор маркера доступа, из которого извлекается информация. Причем этот дескриптор должен быть открыт в режиме `TOKEN_QUERY_SOURCE`, если извлекается информация об источнике маркера доступа и в режиме `TOKEN_QUERY` для извлечения информации другого типа.

Параметр `TokenInformationClass` должен содержать одно из значений перечисления типа `TOKEN_INFORMATION_CLASS`, которое указывает, какую информацию из маркера доступа должна вернуть функция. Перечисление типа `TOKEN_INFORMATION_CLASS` определено следующим образом:

```
typedef enum  _TOKEN_INFORMATION_CLASS {
  TokenUser = 1,            // информация о пользователе
  TokenGroups,   // информация о группах, связанных с маркером доступа
  TokenPrivileges,          // информация о привилегиях
  TokenOwner,         // информация о владельце объекта по умолчанию
  TokenPrimaryGroup,        // информация о первичной группе владельца
                            // объекта по умолчанию
  TokenDefaultDacl,  // информация о списке DACL объекта по умолчанию
  TokenSource,              // источник маркера доступа
  TokenType,                // тип маркера доступа
  TokenImpersonationLevel, // уровень замещения маркера доступа
  TokenStatistics,          // статистика
  TokenRestrictedSids,      // список ограничивающих идентификаторов
                            // безопасности
  TokenSessionId            // идентификатор сессии
}  TOKEN_INFORMATION_CLASS, *PTOKEN_INFORMATION_CLASS;
```

Параметр `TokenInformation` должен указывать на буфер, в который функция поместит информацию, извлекаемую из маркера доступа. В зависимости от значения, заданного параметром `TokenInformationClass`, информация о маркере доступа записана в структурном виде или имеет значение некоторого перечислимого типа. Ниже приведено соответствие значений перечислимой константы типа `TOKEN_INFORMATION_CLASS` и типов, используемых для представления соответствующей информации о маркере доступа:

- TokenUser — `TOKEN_USER`;
- TokenGroups — `TOKEN_GROUPS`;
- TokenPrivileges — `TOKEN_PRIVILEGES`;
- TokenOwner — `TOKEN_OWNER`;
- TokenPrimaryGroup — `TOKEN_PRIMARY_GROUP`;
- TokenDefaultDacl — `TOKEN_DEFAULT_DACL`;
- TokenSource — `TOKEN_SOURCE`;
- TokenType — `TOKEN_TYPE`;
- TokenImpersonationLevel — `SECURITY_IMPERSONATION_LEVEL`;
- TokenStatistics — `TOKEN_STATISTICS`;

- TokenRestrictedSids — TOKEN_GROUPS;
- TokenSessionId — DWORD.

В параметре TokenInformation может быть также установлено значение NULL. В этом случае функция запишет по адресу, заданному последним параметром, необходимую длину буфера.

Параметр TokenInformationLength должен содержать длину буфера, на который указывает параметр TokenInformation. Если в параметре TokenInformation установлено значение NULL, то в параметре TokenInformationLength должно быть установлено значение 0.

В параметре ReturnLength должен быть установлен адрес двойного слова, в которое функция вернет требуемую длину буфера, если в параметре TokenInformation установлено значение NULL или заданная длина буфера недостаточна для записи требуемой информации.

В листинге 43.1 приведена программа, в которой функция GetTokenInformation используется для определения источника маркера доступа.

Листинг 43.1. Определение источника маркера доступа

```
#include <windows.h>
#include <stdio.h>

int main()
{
  HANDLE hProcess;       // дескриптор процесса
  HANDLE hTokenHandle;   // дескриптор маркера доступа

  TOKEN_SOURCE ts;       // источник маркера доступа

  DWORD dwRetLength;     // возвращаемая длина буфера
  DWORD dwRetCode;       // код возврата

  // получаем дескриптор процесса
  hProcess = GetCurrentProcess();

  // открываем маркер доступа процесса
  if (!OpenProcessToken(
    hProcess,            // дескриптор процесса
    TOKEN_QUERY_SOURCE,  // доступ к источнику маркера
```

```
    &hTokenHandle))      // дескриптор маркера
  {
    dwRetCode = GetLastError();
    printf( "Open process token failed: %u\n", dwRetCode);

    return dwRetCode;
  }

  // получаем источник маркера доступа
  if (!GetTokenInformation(
    hTokenHandle,    // дескриптор маркера доступа
    TokenSource,     // получаем источник маркера доступа
    &ts,             // адрес буфера для источника
    sizeof(ts),      // размер буфера
    &dwRetLength))   // требуемый размер буфера в случае неудачи
  {
    dwRetCode = GetLastError();
    printf( "Get token information failed: %u\n", dwRetCode);

    return dwRetCode;
  }

  // распечатываем источник маркера доступа
  printf("Source name: ");
  for (int i = 0; i < TOKEN_SOURCE_LENGTH; ++i)
    putchar(ts.SourceName[i]);
  putchar('\n');
  printf("Source identifier: %X %X\n", ts.SourceIdentifier.HighPart,
    ts.SourceIdentifier.LowPart);

  CloseHandle(hTokenHandle);

  return 0;
}
```

Теперь приведем программу, в которой функция `GetTokenInformation` используется для определения всех привилегий маркера доступа. Эта программа приведена в листинге 43.2.

Листинг 43.2. Определение привилегий маркера доступа

```
#include <windows.h>
#include <stdio.h>

int main()
{
  HANDLE hProcessToken;      // дескриптор маркера доступа
  PTOKEN_PRIVILEGES lpTokenPrivileges = NULL;

  LPTSTR lpPrivName = NULL;          // адрес имени привилегии
  LPTSTR lpPrivDisplayName = NULL;   // адрес имени привилегии для
                                     // отображения

  DWORD dwLangId;
  DWORD dwLength;

  DWORD dwRetCode;      // код возврата

  if (!OpenProcessToken(
    GetCurrentProcess(),     // дескриптор процесса
    TOKEN_QUERY,             // чтение информации из маркера доступа
    &hProcessToken ))        // дескриптор маркера доступа
  {
    dwRetCode = GetLastError();
    perror("Set security descriptor owner failed.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  // получаем длину буфера для привилегий
  if (!GetTokenInformation(
    hProcessToken,       // дескриптор маркера доступа
    TokenPrivileges,     // получаем привилегии
    lpTokenPrivileges,   // адрес буфера
    0,                   // длина буфера
    &dwLength))          // требуемая длина
  {
```

```
  dwRetCode = GetLastError();
  if (dwRetCode != ERROR_INSUFFICIENT_BUFFER)
  {
    perror("Get token information for length failed.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  // захватываем память для имени привилегии
  lpTokenPrivileges = (TOKEN_PRIVILEGES*)new char[dwLength];
}

// получаем привилегии маркера доступа
if (!GetTokenInformation(
  hProcessToken,       // дескриптор маркера доступа
  TokenPrivileges,     // получаем привилегии
  lpTokenPrivileges,   // адрес буфера
  dwLength,            // длина буфера
  &dwLength))          // требуемая длина
{
  dwRetCode = GetLastError();
  perror("Get token information failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// распечатываем привилегии
printf( "User privileges: \n" );
for( unsigned i = 0; i < lpTokenPrivileges->PrivilegeCount; ++i)
{
  // печатаем состояние привилегии
  if ((lpTokenPrivileges->Privileges[i].Attributes &
      SE_PRIVILEGE_ENABLED) == SE_PRIVILEGE_ENABLED)
    printf("SE_PRIVILEGE_ENABLED:\n");
  if ((lpTokenPrivileges->Privileges[i].Attributes &
      SE_PRIVILEGE_ENABLED_BY_DEFAULT) ==
      SE_PRIVILEGE_ENABLED_BY_DEFAULT)
```

```
  printf("SE_PRIVILEGE_ENABLED_BY_DEFAULT:\n");
if ((lpTokenPrivileges->Privileges[i].Attributes &
    SE_PRIVILEGE_USED_FOR_ACCESS) == SE_PRIVILEGE_USED_FOR_ACCESS)
  printf("SE_PRIVILEGE_USED_FOR_ACCESS:\n");
if (!lpTokenPrivileges->Privileges[i].Attributes)
  printf("The privilege is disabled:\n");

// определяем длину имени привилегии
dwLength = 0;
if (!LookupPrivilegeName(
  NULL,        // локальная машина
  &(lpTokenPrivileges->Privileges[i].Luid),  // адрес LUID
  lpPrivName,  // адрес имени привилегии
  &dwLength))  // адрес длины буфера
{
  dwRetCode = GetLastError();
  if (dwRetCode != ERROR_INSUFFICIENT_BUFFER)
  {
    perror("Lookup privilege name for length failed.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  // захватываем память для имени привилегии
  lpPrivName = new char[dwLength + 1];
}

// определяем имя привилегии
if (!LookupPrivilegeName(
  NULL,        // локальная машина
  &(lpTokenPrivileges->Privileges[i].Luid),  // адрес LUID
  lpPrivName,  // адрес имени привилегии
  &dwLength))  // адрес длины буфера
{
  dwRetCode = GetLastError();
  perror("Lookup privilege name failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
```

```
}
// определяем длину имени привилегии для отображения
dwLength = 0;
if (!LookupPrivilegeDisplayName (
  NULL,           // локальная машина
  lpPrivName,     // имя привилегии
  lpPrivDisplayName,   // адрес для имени привилегии
  &dwLength,      // адрес длины буфера
  &dwLangId))     // адрес идентификатора языка
{
  dwRetCode = GetLastError();
  if (dwRetCode != ERROR_INSUFFICIENT_BUFFER)
  {
    perror("Lookup privilege display name for length failed.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  // захватываем память для имени привилегии
  lpPrivDisplayName = new char[dwLength + 1];
}

// определяем имя привилегии для отображения
if (!LookupPrivilegeDisplayName (
  NULL,           // локальная машина
  lpPrivName,     // имя привилегии
  lpPrivDisplayName,   // адрес для имени привилегии
  &dwLength,      // адрес длины буфера
  &dwLangId))     // адрес идентификатора языка
{
  dwRetCode = GetLastError();
  perror("Lookup privilege display name failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}
// распечатываем имя привилегии и идентификатор языка
printf("\t%s\n", lpPrivDisplayName);
```

```
    // освобождаем память
    delete[] lpPrivName;
    delete[] lpPrivDisplayName;
    lpPrivName = NULL;
    lpPrivDisplayName = NULL;
  }
  CloseHandle(hProcessToken);
  return 0;
}
```

43.5. Изменение информации в маркере доступа

Для изменения в маркере доступа информации, используемой в алгоритме для задания по умолчанию атрибутов безопасности нового объекта, применяется функция `SetTokenInformation`, которая имеет следующий прототип:

```
BOOL  SetTokenInformation(
  HANDLE  TokenHandle,             // дескриптор маркера доступа
  TOKEN_INFORMATION_CLASS  TokenInformationClass,  // тип информации
  LPVOID  TokenInformation,        // указатель на буфер с информацией
  DWORD   TokenInformationLength   // длина буфера с информацией
);
```

В случае удачного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного завершения функции можно получить посредством вызова функции `GetLastError`. Параметры функции `SetTokenInformation` имеют следующее назначение.

Параметр `TokenHandle` должен содержать дескриптор маркера доступа, из которого извлекается информация. Причем этот маркер доступа должен быть открыт в режиме `TOKEN_ADJUST_DEFAULT`.

Параметр `TokenInformationClass` должен содержать одно из значений перечисления типа `TOKEN_INFORMATION_CLASS`, которое указывает, какую информацию в маркере доступа изменяет функция. Все константы, принадлежащие перечислению типа `TOKEN_INFORMATION_CLASS`, были приведены в предыдущем параграфе. При использовании функции `SetTokenInformation` в этом параметре могут быть установлены только следующие константы:

- `TokenOwner` — информация о владельце объекта по умолчанию;
- `TokenPrimaryGroup` — информация о первичной группе владельца объекта по умолчанию;
- `TokenDefaultDacl` — информация о списке DACL объекта по умолчанию.

Параметр `TokenInformation` должен указывать на буфер с информацией, которая будет установлена в маркер доступа. В зависимости от значения, заданного параметром `TokenInformationClass`, информация в буфере должна храниться в структуре одного из следующих типов:

- `TokenPrimaryGroup` — `TOKEN_PRIMARY_GROUP`;
- `TokenDefaultDacl` — `TOKEN_DEFAULT_DACL`;
- `TokenSource` — `TOKEN_SOURCE`.

Параметр `TokenInformationLength` должен содержать длину буфера, на который указывает параметр `TokenInformation`.

В листинге 43.3 приведена программа, в которой функция `SetTokenInformation` изменяет идентификатор безопасности владельца объекта, который используется в алгоритме установки владельца нового объекта по умолчанию.

Листинг 43.3. Создание списка DACL объекта по умолчанию

```
#define _WIN32_WINNT 0x0500

#include <windows.h>
#include <stdio.h>
#include <lm.h>
#include <sddl.h>

int main()
{
  HANDLE hMutex;          // дескриптор мьютекса
  HANDLE hProcess;        // дескриптор процесса
  HANDLE hTokenHandle;    // дескриптор маркера доступа

  ACL *lpDacl;            // указатель на список доступа
  void *lpAce = NULL;     // указатель на элемент списка

  BOOL bDaclPresent;      // признак присутствия списка DACL
  BOOL bDaclDefaulted;    // признак списка DACL по умолчанию

  char chUserName[UNLEN];  // имя пользователя

  DWORD dwLengthOfDomainName = 0;  // длина имени домена
  DWORD dwLengthOfSid = 0;    // длина разрешающего SID
```

```
DWORD dwDaclLength = 0;     // длина списка доступа
DWORD dwRetLength = 0;      // возвращаемая длина буфера

SID *lpSid = NULL;          // указатель на разрешающий SID
LPTSTR lpDomainName = NULL;     // указатель на имя домена
SID_NAME_USE typeOfSid;     // тип учетной записи

PSECURITY_DESCRIPTOR lpSd = NULL;  // указатель на SD
DWORD dwLengthOfSd = 0;     // длина дескриптора безопасности

LPTSTR StringSid;     // указатель на строку SID

DWORD dwRetCode;      // код возврата

// получаем дескриптор процесса
hProcess = GetCurrentProcess();

// открываем маркер доступа процесса
if (!OpenProcessToken(
  hProcess,         // дескриптор процесса
  TOKEN_QUERY |     // получаем информацию из маркера доступа
  TOKEN_ADJUST_DEFAULT,  // и измененяем список DACL маркера доступа
  &hTokenHandle))  // дескриптор маркера
{
  dwRetCode = GetLastError();
  printf( "Open process token failed: %u\n", dwRetCode);

  return dwRetCode;
}

// вводим имя пользователя, которому разрешим доступ к каталогу
printf("Input a user name: ");
gets(chUserName);

// определяем длину SID пользователя
if (!LookupAccountName(
  NULL,             // ищем имя на локальном компьютере
  chUserName,       // имя пользователя
  NULL,             // определяем длину SID
```

```
    &dwLengthOfSid,  // длина SID
    NULL,            // определяем имя домена
    &dwLengthOfDomainName,   // длина имени домена
    &typeOfSid))     // тип учетной записи
  {
    dwRetCode = GetLastError();

    if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
    {
      // распределяем память для SID и имени домена
      lpSid = (SID*) new char[dwLengthOfSid];
      lpDomainName = (LPTSTR) new char[dwLengthOfDomainName];
    }
    else
    {
      // выходим из программы
      printf("Lookup account name failed.\n");
      printf("Error code: %d\n", dwRetCode);

      return dwRetCode;
    }
  }

  // определяем SID и имя домена пользователя
  if(!LookupAccountName(
    NULL,            // ищем имя на локальном компьютере
    chUserName,      // имя пользователя
    lpSid,           // указатель на SID
    &dwLengthOfSid,  // длина SID
    lpDomainName,    // указатель на имя домена
    &dwLengthOfDomainName,   // длина имени домена
    &typeOfSid))     // тип учетной записи
  {
    dwRetCode = GetLastError();

    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
```

```
}

// преобразуем SID в строку
if (!ConvertSidToStringSid(
  lpSid,
  &StringSid))
{
  dwRetCode = GetLastError();
  printf("Convert sid to string sid failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}
// печатем SID
printf("%s\n",StringSid);
LocalFree(StringSid);

// определяем длину списка DACL
dwDaclLength = sizeof(ACL)
  + sizeof(ACCESS_ALLOWED_ACE)
  - sizeof(DWORD) + dwLengthOfSid;

// распределяем память под DACL
lpDacl = (ACL*)new char[dwDaclLength];

// инициализируем список DACL
if (!InitializeAcl(
  lpDacl,          // адрес DACL
  dwDaclLength,    // длина DACL
  ACL_REVISION))   // версия DACL
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// добавляем разрешающий элемент в список DACL
```

```
if (!AddAccessAllowedAce(
  lpDacl,        // адрес DACL
  ACL_REVISION,  // версия DACL
  GENERIC_ALL,   // разрешаем все родовые права доступа
  lpSid))        // адрес SID
{
  dwRetCode = GetLastError();
  perror("Add access allowed ace failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// устанавливаем новый DACL по умолчанию в маркер доступа
if (!SetTokenInformation(
  hTokenHandle,      // дескриптор маркера доступа
  TokenDefaultDacl,  // устанавливаем список DACL
  &lpDacl,           // адрес буфера с новым списком DACL
  dwDaclLength))     // размер буфера
{
  dwRetCode = GetLastError();
  printf( "Set token information failed: %u\n", dwRetCode);

  return dwRetCode;
}

// создаем мьютекс
hMutex = CreateMutex(NULL, FALSE, "DemoMutex");
if (hMutex == NULL)
{
  dwRetCode = GetLastError();
  perror("Create mutex failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}
printf("The mutex is created.\n");

// освобождаем память
```

```
delete[] lpSid;
delete[] lpDomainName;
delete[] lpDacl;

// теперь проверяем список DACL мьютекса

// определяем требуемую длину буфера
if (!GetKernelObjectSecurity(
  hMutex,          // дескриптор мьютекса
  DACL_SECURITY_INFORMATION,   // получаем список DACL мьютекса
  lpSd,            // адрес SD
  0,               // определяем длину буфера
  &dwLengthOfSd))  // требуемая длина буфера
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
    // распределяем память для буфера
    lpSd = (SECURITY_DESCRIPTOR*) new char[dwLengthOfSd];
  else
  {
    // выходим из программы
    printf("Get kernel object security failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// читаем идентификатор безопасности мьютекса
if (!GetKernelObjectSecurity(
  hMutex,          // дескриптор мьютекса
  DACL_SECURITY_INFORMATION,   // получаем список DACL мьютекса
  lpSd,            // адрес SD
  dwLengthOfSd,    // длина буфера
  &dwLengthOfSd))  // требуемая длина буфера
{
  dwRetCode = GetLastError();
  printf("Get kernel object security failed.\n");
```

```
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// получаем список DACL из дескриптора безопасности
if (!GetSecurityDescriptorDacl(
  lpSd,              // адрес дескриптора безопасности
  &bDaclPresent,     // признак присутствия списка DACL
  &lpDacl,           // адрес указателя на DACL
  &bDaclDefaulted))  // признак списка DACL по умолчанию
{
  dwRetCode = GetLastError();
  printf("Get security descriptor DACL failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// проверяем, есть ли DACL
if (!bDaclPresent)
{
  printf("Dacl is not present.\n");
  return 0;
}
else
  printf("Dacl is present.\n");

// проверяем, установлен ли DACL пользователем
if (bDaclDefaulted == SE_DACL_DEFAULTED)
  printf("Dacl is defaulted.\n");
else
  printf("Dacl is not defaulted.\n");


// печатаем количество элементов
printf("Ace count: %u\n", lpDacl->AceCount);

// получаем элементы списка DACL
for (unsigned i = 0; i < lpDacl->AceCount; ++i)
```

```
  {
    if (!GetAce(
      lpDacl,    // адрес DACL
      i,         // индекс элемента
      &lpAce))   // указатель на элемент списка
    {
      dwRetCode = GetLastError();
      printf("Get ace failed.\n");
      printf("Error code: %d\n", dwRetCode);

      return dwRetCode;
    }
    // выводим на печать тип элемента и SID
    if (((ACE_HEADER*)lpAce)->AceType == ACCESS_ALLOWED_ACE_TYPE)
    {
      printf("ACCESS_ALLOWED_ACE_TYPE\n");
      // преобразуем SID в строку
      if (!ConvertSidToStringSid(
        &((ACCESS_ALLOWED_ACE*)lpAce)->SidStart,
        &StringSid))
      {
        dwRetCode = GetLastError();
        printf("Convert sid to string sid failed.\n");
        printf("Error code: %d\n", dwRetCode);

        return dwRetCode;
      }
      // печатаем SID
      printf("%s\n",StringSid);
      LocalFree(StringSid);
    }
  }
  // освобождаем память
  delete[] lpSd;

  CloseHandle(hMutex);
  CloseHandle(hTokenHandle);

  return 0;
}
```

43.6. Настройка привилегий

Теперь рассмотрим, как настраивать привилегии в маркере доступа. Вообще привилегии даются пользователю при регистрации этого пользователя в системе. Каждый пользователь имеет свой набор привилегий, некоторые из которых могут быть установлены или действительны, а некоторые — сброшены или отменены. Маркер доступа получает свой набор привилегий из учетной записи пользователя, которая явилась источником создания данного маркера доступа. Добавить новые или удалить существующие привилегии в маркер доступа невозможно. Можно только сделать привилегию действительной или отмененной. Или, другими словами, установить или сбросить привилегию соответственно. Назовем такую работу с привилегиями *настройкой привилегий в маркере доступа*.

Для настройки привилегий, доступных в маркере доступа, используется функция `AdjustTokenPrivileges`, которая имеет следующий прототип:

```
BOOL  AdjustTokenPrivileges(
  HANDLE  TokenHandle,               // дескриптор маркера доступа
  BOOL    DisableAllPrivileges,      // флаг сброса всех привилегий
  PTOKEN_PRIVILEGES  NewState,       // новое состояние привилегий
  DWORD   BufferLength,              // длина буфера PreviousState
  PTOKEN_PRIVILEGES  PreviousState,  // старое состояние привилегий
  PDWORD  ReturnLength       // требуемая длина буфера PreviousState
);
```

В случае удачного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного завершения функции можно получить посредством вызова функции `GetLastError`. Параметры функции `AdjustTokenPrivileges` имеют следующее назначение.

Параметр `TokenHandle` должен содержать дескриптор маркера доступа, в котором настраиваются привилегии. Причем маркер доступа должен быть открыт в режиме `TOKEN_ADJUST_PRIVILEGES`.

В параметре `DisableAllPrivileges` должно быть установлено одно из значений: `FALSE` или `TRUE`. Если установлено значение `FALSE`, то функция настраивает привилегии в соответствии с информацией, заданной параметром `NewState`. Если же в этом параметре установлено значение `TRUE`, то функция отменяет все привилегии в заданном маркере доступа.

Параметр `NewState` должен указывать на структуру типа `TOKEN_PRIVILEGES`, которая содержит привилегии и новые состояния для каждой из перечисленных привилегий. Если для привилегии установлен флаг `SE_PRIVILEGE_ENABLED`, то привилегия становится действительной, в противном случае — привилегия отменяется. Подробно структура типа `TOKEN_PRIVILEGES` была рассмотрена в *разд. 43.3*.

Параметр BufferLength должен содержать размер буфера памяти, на который указывает параметр PreviousState. Если в PreviousState установлено значение NULL, то в этом параметре может быть установлено значение 0.

Параметр PreviousState должен указывать на буфер, в который функция запишет структуру типа TOKEN_PRIVILEGES. Эта структура будет содержать предыдущее состояние привилегий маркера доступа. В этом параметре может быть установлено значение NULL. В этом случае функция не вернет предыдущее состояние привилегий маркера доступа. Если длина буфера недостаточна для записи предыдущего состояния привилегий, то функция закончится неудачей, а требуемая длина буфера будет записана в переменную, на которую указывает параметр ReturnLength. При этом заметим, что предыдущее состояние привилегий маркера доступа используется для восстановления привилегий в первоначальное состояние.

Параметр ReturnLength должен указывать на переменную типа PDWORD, в которую функция запишет требуемый размер буфера для предыдущего состояния привилегий маркера доступа. Если размер, заданный параметром BufferLength, меньше необходимого. Если в параметре PreviousState установлено значение NULL, то в этом параметре также может быть установлено значение NULL.

Примеры использования функции AdjustTokenPrivileges для настройки в маркере доступа состояния привилегии SE_SECURITY_NAME, которая разрешает доступ к списку DACL, содержатся в программах из листингов 41.4 и 41.6, которые приведены в *разд. 41.5* и *41.6* соответственно.

43.7. Настройка групп

Под настройкой групп в маркере доступа понимаем установку или сброс флага SE_GROUP_ENABLED для каждой группы, связанной с заданным маркером доступа. При этом отметим, что этот флаг не может быть установлен для группы с установленным флагом SE_GROUP_USE_FOR_DENY_ONLY. Кроме того, заметим, что флаг SE_GROUP_ENABLED не может быть сброшен для группы с установленным флагом SE_GROUP_MANDATORY. Подробно флаги, задающие атрибуты группы, рассмотрены в *разд. 43.3*, при описании возможных значений поля Attributes структуры типа TOKEN_GROUPS.

Для настройки групп используется функция AdjustTokenGroups, которая имеет следующий прототип:

```
BOOL  AdjustTokenGroups(
  HANDLE  TokenHandle,       // дескриптор маркера доступа
  BOOL    ResetToDefault,    // сбросить в состояние по умолчанию
  PTOKEN_GROUPS  NewState,   // новое состояние групп
```

```
    DWORD   BufferLength,       // длина буфера PreviousState
    PTOKEN_GROUPS  PreviousState,    // предыдущее состояние групп
    PDWORD  ReturnLength        // требуемая длина буфера PreviousState
);
```

В случае удачного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного завершения функции можно получить посредством вызова функции GetLastError. Параметры функции AdjustTokenPrivileges имеют следующее назначение.

Параметр TokenHandle должен содержать дескриптор маркера доступа, в котором настраиваются группы. Причем маркер доступа должен быть открыт в режиме TOKEN_ADJUST_GROUPS.

В параметре ResetToDefault должно быть установлено одно из значений: FALSE или TRUE. Если установлено значение FALSE, то функция настраивает группы в соответствии с информацией, заданной параметром NewState. Если же в этом параметре установлено значение TRUE, то функция переводит все группы в состояния, которые были заданы по умолчанию.

Параметр NewState должен указывать на структуру типа TOKEN_GROUPS, которая содержит группы и новые флага состояния SE_GROUP_ENABLED для каждой из перечисленных групп. Если для группы установлен флаг SE_GROUP_ENABLED, то этот флаг будет также установлен и для соответствующей группы в маркере доступа, в противном случае этот флаг будет сброшен для соответствующей группы в маркере доступа. Подробно структура типа TOKEN_GROUPS была рассмотрена в *разд. 43.3*.

Параметр BufferLength должен содержать размер буфера памяти, на который указывает параметр PreviousState. Если в параметре PreviousState установлено значение NULL, то в этом параметре может быть установлено значение 0.

Параметр PreviousState должен указывать на буфер, в который функция запишет структуру типа TOKEN_GROUPS. Эта структура будет содержать предыдущее состояние групп, связанных с маркером доступа. В этом параметре может быть установлено значение NULL. В этом случае функция не вернет предыдущее состояние групп из маркера доступа. Если длина буфера недостаточна для записи предыдущего состояния групп, то функция закончится неудачей, а требуемая длина буфера будет записана в переменную, на которую указывает параметр ReturnLength. При этом заметим, что предыдущее состояние групп используется для восстановления состояния групп в начальное состояние.

Параметр ReturnLength должен указывать на переменную типа PDWORD, в которую функция запишет требуемую длину буфера для предыдущего состояния групп маркера доступа, если длина заданная параметром BufferLength меньше необходимой. Если в параметре PreviousState уста-

новлено значение NULL, то в этом параметре также может быть установлено значение NULL.

43.8. Создание маркера ограниченного доступа

Маркером ограниченного доступа называется маркер доступа, который создается из существующего маркера доступа, но может иметь по сравнению с оригиналом следующие ограничения:

- некоторые идентификаторы безопасности, которые в исходном маркере доступа использовались для проверки разрешения доступа к охраняемому объекту, в ограничивающем маркере доступа будут использоваться для проверки запрещения доступа к объекту. То есть идентификатор безопасности, имеющий атрибут SE_GROUP_ENABLED в исходном маркере доступа, будет иметь атрибут SE_GROUP_USE_FOR_DENY_ONLY в маркере ограниченного доступа;
- некоторые привилегии могут быть отменены в маркере ограниченного доступа, хотя они были действительны в исходном маркере доступа;
- маркер ограниченного доступа может иметь дополнительные идентификаторы безопасности, которые используются для проверки разрешения доступа субъекта, который представляется этим маркером доступа, к охраняемому объекту; такие идентификаторы безопасности называются *ограничивающими идентификаторами безопасности*.

То есть маркер ограниченного доступа имеет два множества, связанных с ним идентификаторов безопасности. Одно множество идентификаторов безопасности наследуется от исходного маркера доступа, а второе — включает ограничивающие идентификаторы безопасности, которые добавлены в маркер ограниченного доступа при его создании. Ограничение доступа к охраняемому объекту при помощи маркера ограниченного доступа выполняется посредством того, что система управления безопасностью контролирует доступ, используя как идентификаторы безопасности, наследуемые маркером ограниченного доступа от исходного маркера, так и ограничивающие маркеры, которые были добавлены в маркер ограниченного доступа.

По сути, маркер ограниченного доступа и определяется наличием в нем списка ограничивающих идентификаторов безопасности. Если этот список пуст, то маркер доступа не считается ограниченным в доступе к охраняемым объектам.

Маркеры ограниченного доступа используются, во-первых, для создания нового процесса посредством вызова функции CreateProcessAsUser, который имеет ограниченный контекст безопасности и может иметь свой рабочий стол; во-вторых, для ограничения контекста безопасности текущего процесса посредством вызова функции ImpersonateLoggedOnUser.

Для создания маркера ограниченного доступа используется функция `CreateRestrictedToken`, которая имеет следующий прототип:

```
BOOL  CreateRestrictedToken(
  HANDLE   ExistingTokenHandle, // дескриптор исходного маркера доступа
  DWORD    Flags,               // флаг сброса привилегий
  DWORD    DisableSidCount,     // количество запрещающих SID
  PSID_AND_ATTRIBUTES  SidsToDisable,       // массив запрещающих SID
  DWORD    DeletePrivilegeCount, // количество отменяемых привилегий
  PLUID_AND_ATTRIBUTES  PrivilegesToDelete,  // массив отменяемых
                                             // привилегий
  DWORD  RestrictedSidCount,    // количество ограничивающих SID
  PSID_AND_ATTRIBUTES  SidsToRestrict, // массив ограничивающих SID
  PHANDLE  NewTokenHandle       // указатель на дескриптор нового
                                // маркера доступа
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного завершения функции можно получить посредством вызова функции `GetLastError`. Параметры функции `CreateRestrictedToken` имеют следующее назначение.

Параметр `ExistingTokenHandle` должен содержать дескриптор исходного маркера доступа, который может быть первичным или замещающим маркером доступа, а также маркером ограниченного доступа. Данный маркер должен быть открыт в режиме `TOKEN_DUPLICATE`.

В параметре `Flags` может быть установлен флаг `DISABLE_MAX_PRIVILEGE`, который отменяет все привилегии в маркере ограниченного доступа. Если этот флаг установлен, то функция игнорирует параметры `DeletePrivilegeCount` и `PrivilegesToDelete`.

Параметр `DisableSidCount` должен содержать количество запрещающих идентификаторов безопасности в массиве, на который указывает следующий параметр `SidsToDisable`.

Параметр `SidsToDisable` должен указывать на массив структур типа `SID_AND_ATTRIBUTES`, каждая из которых содержит запрещающий идентификатор безопасности, для которого будет установлен режим `SE_GROUP_USE_FOR_DENY_ONLY` в маркере ограниченного доступа. Отметим, что эти запрещающие идентификаторы должны быть связаны с исходным маркером доступа. Функция игнорирует идентификаторы безопасности, которые не связаны с исходным маркером доступа. Функция также игнорирует атрибуты идентификатора безопасности, которые установлены в поле `Attributes` структуры `SID_AND_ATTRIBUTES`. Если функция не изменяет атрибуты дескрипторов безопасности, которые связаны с исходным маркером доступа,

то в этом параметре можно установить значение NULL. Подробное описание структуры типа SID_AND_ATTRIBUTES, которая описывает элементы массива SidsToDisable, было приведено в *разд. 43.3.*

Параметр DeletePrivilegeCount должен содержать количество отменяемых привилегий в массиве, на который указывает следующий параметр PrivilegesToDelete.

Параметр PrivilegesToDelete должен указывать на массив структур типа LUID_AND_ATTRIBUTES, каждая из которых содержит локальный идентификатор привилегии, которая должна быть отменена в маркере ограниченного доступа. Функция игнорирует все привилегии, которые не входят в исходный маркер доступа. Функция также игнорирует атрибуты привилегий, которые установлены в поле Attributes структуры типа LUID_AND_ATTRIBUTES. Если функция не отменяет привилегии исходного маркера доступа, то в этом параметре можно установить значение NULL. Подробное описание структуры типа LUID_AND_ATTRIBUTES было приведено в *разд. 42.1.*

Параметр RestrictedSidCount должен содержать количество ограничивающих идентификаторов в массиве, на который указывает следующий параметр SidsToRestrict.

Параметр SidsToRestrict должен указывать на массив структур типа SID_AND_ATTRIBUTES, каждая из которых содержит ограничивающий идентификатор безопасности, который будет связан с новым маркером ограниченного доступа. Ограничивающие идентификаторы безопасности не связаны с исходным маркером доступа и используются только для проверки разрешения доступа маркера ограниченного доступа к охраняемому объекту. Поле Attributes структуры SID_AND_ATTRIBUTES должно содержать нулевое значение для каждого ограничивающего маркера доступа. Если функция не связывает с маркером ограниченного доступа ограничивающих идентификаторов безопасности, то в этом параметре можно установить значение NULL. Если исходным маркером доступа является маркер ограниченного доступа, то с новым маркером ограниченного доступа будут связаны только те ограничивающие идентификаторы безопасности, которые одновременно связаны с исходным маркером ограниченного доступа и находятся в массиве, на который указывает параметр SidsToRestrict. Подробное описание структуры типа SID_AND_ATTRIBUTES, которая описывает элементы массива SidsToRestrict, было приведено в *разд. 43.3.*

Параметр NewTokenHandle должен содержать адрес переменной типа HANDLE, в которую функция запишет дескриптор созданного маркера ограниченного доступа. Созданный маркер доступа имеет тот же тип, что и исходный маркер доступа, т. е. может быть первичным или замещающим маркером доступа. А также для нового маркера доступа установлены те же режимы доступа, что и для исходного маркера доступа.

Для проверки является ли заданный маркер доступа маркером ограниченного доступа используется функция `IsTokenRestricted`, которая имеет следующий прототип:

```
BOOL  IsTokenRestricted(
  HANDLE  TokenHandle    // дескриптор маркера доступа
);
```

Единственный параметр этой функции должен содержать дескриптор маркера доступа, который подвергается проверке на ограничения по доступу к охраняемым объектам. Если маркер доступа, который содержится в параметре `TokenHandle`, содержит список ограничивающих идентификаторов безопасности, то функция вернет ненулевое значение, в противном случае — `FALSE`. Код ошибки в случае неудачного завершения функции можно получить посредством вызова функции `GetLastError`.

В листинге 43.4 приведена программа, в которой при помощи функции `CreateRestrictedToken` создается маркер ограниченного доступа. После этого этот маркер проверяется функцией `IsTokenRestricted` на ограниченность.

Листинг 43.4. Создание и проверка маркера ограниченного доступа

```
#include <windows.h>
#include <stdio.h>
#include <lm.h>

int main()
{
  HANDLE hProcess;     // дескриптор процесса
  HANDLE hToken;       // дескриптор маркера доступа
  HANDLE hRestrict;    // дескриптор ограничивающего маркера доступа

  char chUserName[UNLEN];  // имя пользователя

  DWORD  dwLengthOfUserName = UNLEN;  // длина имени учетной записи

  DWORD  dwLengthOfSID = 0;     // длина SID
  DWORD  dwLengthOfDomainName = 0;   // длина имени домена

  SID    *lpSid = NULL;          // указатель на SID
  LPTSTR  lpDomainName = NULL;   // указатель на имя домена
```

```
SID_NAME_USE type_of_SID;      // тип учетной записи

SID_AND_ATTRIBUTES RestrictingSid;   // ограничивающий SID

DWORD dwRetCode;       // код возврата

// получаем дескриптор процесса
hProcess = GetCurrentProcess();

// получаем маркер доступа процесса
if (!OpenProcessToken(
  hProcess,          // дескриптор процесса
  TOKEN_ALL_ACCESS,  // полный доступ к маркеру доступа
  &hToken))          // дескриптор маркера
{
  dwRetCode = GetLastError();
  printf( "Open process token failed: %u\n", dwRetCode);

  return dwRetCode;
}
printf("Input a user name: ");
gets(chUserName);        // вводим имя пользователя

// определяем длину SID пользователя
if (!LookupAccountName(
  NULL,            // ищем имя на локальном компьютере
  chUserName,      // имя пользователя
  NULL,            // определяем длину SID
  &dwLengthOfSID,  // длина SID
  NULL,            // определяем имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &type_of_SID))   // тип учетной записи
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
  {
    // распределяем память для SID и имени домена
    lpSid = (SID*) new char[dwLengthOfSID];
```

```
    lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];
  }
  else
  {
    // выходим из программы
    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// определяем SID и имя домена пользователя
if(!LookupAccountName(
  NULL,             // ищем имя на локальном компьютере
  chUserName,       // имя пользователя
  lpSid,            // указатель на SID
  &dwLengthOfSID,   // длина SID
  lpDomainName,     // указатель на имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &type_of_SID))    // тип учетной записи
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// инициализируем ограничивающий SID
RestrictingSid.Sid = lpSid;
RestrictingSid.Attributes = 0;

// создаем ограничивающий маркер доступа
if (!CreateRestrictedToken(
  hToken,           // дескриптор исходного маркера доступа
  0,                // привилегии не изменяем
  0, NULL,          // флаг SE_GROUP_USE_FOR_DENY_ONLY не устанавливаем
```

```
  0, NULL,          // привилегии не удаляем
  1,                // один ограничивающий SID
  &RestrictingSid, // адрес ограничивающего SID
  &hRestrict))      // адрес ограничивающего маркера доступа
{
  dwRetCode = GetLastError();

  printf("Create restricted token failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// проверяем замещение маркера доступа
if (IsTokenRestricted(hRestrict))
  printf("The restricted token is created.\n");
else
{
  dwRetCode = GetLastError();

  printf("Is Token Restricted failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}


// замещаем контекст безопасности потока
if (!ImpersonateLoggedOnUser(hRestrict))
{
  dwRetCode = GetLastError();

  printf("Impersonate logged on user failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// здесь делаем какую-то работу
```

```
  printf("Some work is done.\n");

  // возвращаем исходный контекст
  if (!RevertToSelf())
  {
    dwRetCode = GetLastError();

    printf("Revert to self failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }

  CloseHandle(hToken);

  return 0;
}
```

43.9. Дублирование маркеров доступа

Под *дублированием маркера доступа* понимается создание нового маркера доступа, который может отличаться от исходного маркера доступа установленными режимами доступа, дескриптором безопасности и может быть как первичным, так и замещающим маркером доступа. В дублированном маркере доступа остается без изменения список идентификаторов безопасности, связанных с исходным маркером доступа.

Для дублирования маркеров доступа используется функция `DuplicateToken`, которая имеет следующий прототип:

```
BOOL  DuplicateToken(
  HANDLE  ExistingTokenHandle,  // дескриптор исходного маркера доступа
  SECURITY_IMPERSONATION_LEVEL  ImpersonationLevel,// уровень замещения
  PHANDLE DuplicateTokenHandle  // дескриптор нового маркера доступа
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного завершения функции можно получить посредством вызова функции `GetLastError`. Параметры функции `DuplicateToken` имеют следующее назначение.

Параметр `ExistingTokenHandle` должен содержать дескриптор исходного маркера доступа, который будет продублирован. Исходный маркер доступа должен быть открыт в режиме доступа `TOKEN_DUPLICATE`.

В параметре ImpersonationLevel должно быть установлено одно из значений перечисления типа SECURITY_IMPERSONATION_LEVEL, которое определяет уровень замещения дублированным маркером доступа исходного маркера доступа. Перечисление типа SECURITY_IMPERSONATION_LEVEL было определено в *разд. 43.3*. Подробнее об уровнях замещения маркеров доступа было рассказано в *разд. 36.8*.

Параметр DuplicateTokenHandle должен указывать на переменную типа HANDLE, в которую функция запишет дескриптор нового маркера доступа, полученного дублированием исходного маркера доступа. Для нового маркера доступа будут установлены режимы доступа TOKEN_IMPERSONATE и TOKEN_QUERY.

Программа, в которой создается дубликат маркера доступа, используя для этого функцию DuplicateToken, приведена в листинге 43.5.

Для дублирования маркера доступа с возможностью изменения в новом маркере доступа свойств, установленных в исходном маркере доступа, используется функция DuplicateTokenEx, которая имеет следующий прототип:

```
BOOL  DuplicateTokenEx(
  HANDLE   hExistingToken,   // дескриптор исходного маркера доступа
  DWORD    dwDesiredAccess,  // права доступа
  LPSECURITY_ATTRIBUTES  lpTokenAttributes,    // атрибуты безопасности
  SECURITY_IMPERSONATION_LEVEL  ImpersonationLevel,// уровень замещения
  TOKEN_TYPE  TokenType,     // тип маркера доступа
  PHANDLE  phNewToken        // дескриптор нового маркера доступа
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного завершения функции можно получить посредством вызова функции GetLastError. Параметры функции DuplicateTokenEx имеют следующее назначение.

Параметр hExistingToken должен содержать дескриптор исходного маркера доступа, который будет продублирован. Исходный маркер доступа должен быть открыт в режиме доступа TOKEN_DUPLICATE.

Параметр dwDesiredAccess должен содержать режимы доступа к новому маркеру доступа. При установке новых режимов доступа функция DuplicateTokenEx проверяет список DACL исходного маркера доступа, чтобы определить разрешены или запрещены эти режимы для учетной записи, от имени которой выполняется функция DuplicateTokenEx. Если режимы доступа существующего маркера доступа нужно продублировать в новом маркере доступа, то в этом параметре должно быть установлено значение 0. Если в новом маркере доступа нужно установить максимально возможные режимы доступа, то в этом параметре должно быть установлено значение MAXIMUM_ALLOWED.

Параметр lpTokenAttributes должен указывать на атрибуты безопасности нового маркера доступа. Если в этом параметре установлено NULL, то в новом маркере доступа устанавливается дескриптор безопасности, определяемый по умолчанию.

В параметре ImpersonationLevel должно быть установлено одно из значений перечисления типа SECURITY_IMPERSONATION_LEVEL, которое определяет уровень замещения дублированным маркером доступа исходного маркера доступа. Определение перечисления типа SECURITY_IMPERSONATION_LEVEL было приведено в *разд. 43.3*. Подробнее об уровнях замещения маркеров доступа было рассказано в *разд. 36.8*.

В параметре TokenType должно быть установлено одно из значений перечисления TOKEN_TYPE, которое определяет, является новый маркер доступа первичным или замещает первичный маркер доступа. Определение перечисления типа TOKEN_TYPE было приведено в *разд. 43.3*. Подробнее о замещении маркеров доступа было рассказано в *разд. 36.8*.

Параметр DuplicateTokenHandle должен указывать на переменную типа HANDLE, в которую функция запишет дескриптор нового маркера доступа, полученного дублированием исходного маркера доступа.

43.10. Замещение маркеров доступа потока

Замещением маркера доступа будем называть подмену маркера доступа одного потока на маркер доступа другого потока. При этом маркер доступа, который замещает исходный маркер доступа, должен иметь тип TokenImpersonation. Напомним, что значение TokenImpersonation является перечислимой константой, которая принадлежит перечислению типа TOKEN_TYPE. Определение этого перечисления было дано в *разд. 43.3*.

Для замещения маркера доступа потока используется функция SetThreadToken, которая имеет следующий прототип:

```
BOOL  SetThreadToken(
  PHANDLE  Thread,     // указатель на дескриптор потока
  HANDLE   Token       // дескриптор замещающего маркера доступа
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного завершения функции можно получить посредством вызова функции GetLastError. Параметры функции SetThreadToken имеют следующее назначение.

Параметр Thread должен содержать адрес дескриптора потока, для которого будет замещен маркер доступа. Если в этом параметре установлено значение

NULL, то маркер доступа будет замещен для потока, вызвавшего функцию SetThreadToken.

Параметр Token должен содержать дескриптор маркера доступа, который заместит маркер доступа в потоке, дескриптор которого задан первым параметром. Если в параметр Token установлено значение NULL, то функция завершит поток, используя замещающий маркер доступа.

В листинге 43.5 приведена программа, в которой создается дубликат маркера доступа, используя для этого функцию DuplicateToken. После этого при помощи функции SetThreadToken созданный дубликат замещает маркер доступа потока.

Листинг 43.5. Дублирование и замещение маркера доступа

```
#include <windows.h>
#include <stdio.h>

volatile UINT count;
volatile BOOL flag = TRUE;

void thread()
{
  while(flag)
  {
    count++;
    printf("count = %u\n", count);
    Sleep(500);
  }
}

int main()
{
  HANDLE hThread;
  DWORD IDThread;

  HANDLE hProcess;     // дескриптор процесса
  HANDLE hToken;       // дескриптор маркера доступа
  HANDLE hDuplicate;   // дескриптор дубликата маркера доступа

  DWORD dwRetCode;     // код возврата
```

```
// запускаем поток
hThread = CreateThread(
  NULL,          // защита по умолчанию
  0,             // размер стека по умолчанию
  (LPTHREAD_START_ROUTINE)thread,  // адрес программы потока
  NULL,          // параметров нет
  CREATE_SUSPENDED,    // подвешенное состояние потока
  &IDThread);    // идентификатор потока
if (hThread == NULL)
{
  printf("Create thread failed.\n");
  return GetLastError();
}

// получаем дескриптор процесса
hProcess = GetCurrentProcess();

// получаем маркер доступа потока
if (!OpenProcessToken(
  hProcess,          // дескриптор процесса
  TOKEN_DUPLICATE,   // дублирование маркера доступа
  &hToken))          // дескриптор маркера
{
  dwRetCode = GetLastError();
  printf( "Open process token failed: %u\n", dwRetCode);

  return dwRetCode;
}

// дублируем маркер доступа
if (!DuplicateToken(
  hToken,                  // маркер доступа
  SecurityImpersonation,   // уровень замещения
  &hDuplicate))            // адрес дубликата маркера доступа
{
  dwRetCode = GetLastError();
  printf( "Duplicate token failed: %u\n", dwRetCode);

  return dwRetCode;
```

```
  }

  // устанавливаем замещающий маркер доступа для потока
  if (!SetThreadToken(
    &hThread,      // адрес дескриптора потока
    hDuplicate))   // дубликат маркера доступа
  {
    dwRetCode = GetLastError();
    printf( "Set thread token failed: %u\n", dwRetCode);

    return dwRetCode;
  }

  printf("Press any key to exit.\n");

  // возобновляем поток
  ResumeThread(hThread);

  // ждем команду на завершение потока
  getchar();

  flag = FALSE;    // завершить поток

  CloseHandle(hThread);
  CloseHandle(hToken);

  return 0;
}
```

43.11. Проверка идентификатора безопасности на принадлежность маркеру доступа

Действительными идентификаторами безопасности в маркере доступа называются такие идентификаторы, для которых установлен флаг `SE_GROUP_ENABLED`.

Для проверки, является ли заданный идентификатор действительным в маркере доступа, который замещает первичный маркер доступа, используется функция `CheckTokenMembership`, которая имеет следующий прототип:

```
BOOL  CheckTokenMembership(
  HANDLE  TokenHandle,   // дескриптор маркера доступа
```

```
    PSID    SidToCheck,    // указатель на идентификатор безопасности
    PBOOL   IsMember       // результат проверки
  );
```

В случае удачного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного завершения функции можно получить посредством вызова функции GetLastError. Параметры функции CheckTokenMembership имеют следующее назначение.

Параметр TokenHandle должен содержать дескриптор маркера доступа, который замещает первичный маркер доступа. Маркер доступа должен быть открыт в режиме TOKEN_QUERY. Функция проверяет связанные с этим маркером доступа идентификаторы безопасности, чтобы определить, является ли заданный идентификатор безопасности действительным. Если в этом параметре установлено значение NULL, то функция проверяет идентификаторы безопасности в замещающем маркере доступа потока, вызвавшего эту функцию. Если этот маркер доступа не является замещающим, то функция дублирует его, чтобы создать замещающий маркер доступа, а затем проводит проверку.

Параметр SidToCheck должен указывать на идентификатор безопасности, для которого функция выполняет проверку на действительность.

Параметр IsMember должен указывать на переменную типа BOOL, в которую функция запишет результат проверки. Если идентификатор безопасности, на который указывает параметр SidToCheck, связан с маркером доступа и для этого идентификатора установлен флаг SE_GROUP_ENABLED, то функция запишет по адресу IsMember значение TRUE, иначе — FALSE.

В листинге 43.6 приведена программа, в которой замещается маркер доступа потока thread, а потом этот поток проверяет — установлен ли атрибут SE_GROUP_ENABLED для заданной группы, имя которой вводится с консоли.

Листинг 43.6. Проверка состояния атрибута SE_GROUP_ENABLED

```
#include <windows.h>
#include <stdio.h>
#include <lm.h>

extern "C" BOOL WINAPI CheckTokenMembership(HANDLE, PSID, PBOOL);

DWORD WINAPI thread(LPVOID lpSid)
{
  HANDLE hImperson;  // дескриптор замещенного потока
  BOOL bIsMember;    // признак присутствия SID
```

```
DWORD dwRetCode;   // код возврата

// получаем маркер доступа потока
if (!OpenThreadToken(
  GetCurrentThread(),
  TOKEN_QUERY,
  FALSE,
  &hImperson))
{
  dwRetCode = GetLastError();

  printf("Open Thread Token failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// проверяем принадлежность маркера доступа
if (!CheckTokenMembership(
  hImperson,
  lpSid,
  &bIsMember))
{
  dwRetCode = GetLastError();

  printf("Check token membership failed.\n");
  printf("Error code: %d\n", dwRetCode);

  CloseHandle(hImperson);

  return dwRetCode;
}

if (bIsMember)
  printf("The group is enabled.\n");
else
  printf("The group is disabled or not present.\n");

CloseHandle(hImperson);
```

```
  return 0;
}

int main()
{
  HANDLE hThread;
  DWORD IDThread;

  HANDLE hToken;       // дескриптор маркера доступа
  HANDLE hDuplicate;   // дескриптор дубликата маркера доступа

  char chGroupName[GNLEN];     // имя пользователя

  DWORD dwLengthOfUserName = GNLEN;  // длина имени учетной записи

  DWORD dwLengthOfSID = 0;     // длина SID
  DWORD dwLengthOfDomainName = 0;    // длина имени домена

  PSID lpSid = NULL;           // указатель на проверяемый SID
  LPTSTR lpDomainName = NULL;  // указатель на имя домена

  SID_NAME_USE typeOfSid;      // тип учетной записи

  DWORD dwRetCode;             // код возврата

  printf("Input a group name: ");
  gets(chGroupName);           // вводим имя пользователя

  // определяем длину SID пользователя
  if (!LookupAccountName(
    NULL,             // ищем имя на локальном компьютере
    chGroupName,      // имя группы
    NULL,             // определяем длину SID
    &dwLengthOfSID,   // длина SID
    NULL,             // определяем имя домена
    &dwLengthOfDomainName,   // длина имени домена
    &typeOfSid))      // тип учетной записи
  {
    dwRetCode = GetLastError();
```

```
  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
  {
    // распределяем память для SID и имени домена
    lpSid = (SID*) new char[dwLengthOfSID];
    lpDomainName = (LPTSTR) new char[dwLengthOfDomainName];
  }
  else
  {
    // выходим из программы
    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// определяем SID и имя домена пользователя
if(!LookupAccountName(
  NULL,             // ищем имя на локальном компьютере
  chGroupName,      // имя пользователя
  lpSid,            // указатель на SID
  &dwLengthOfSID,   // длина SID
  lpDomainName,     // указатель на имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &typeOfSid))      // тип учетной записи
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// проверяем тип идентификатора безопасности
if (typeOfSid != SidTypeAlias)
{
  printf("This is not an alias.\n");
  return 0;
```

```
}

// получаем маркер доступа процесса
if (!OpenProcessToken(
  GetCurrentProcess(),   // дескриптор процесса
  TOKEN_DUPLICATE,       // дублирование маркера доступа
  &hToken))              // дескриптор маркера
{
  dwRetCode = GetLastError();
  printf( "Open process token failed: %u\n", dwRetCode);

  return dwRetCode;
}

// дублируем маркер доступа
if (!DuplicateToken(
  hToken,          // маркер доступа
  SecurityImpersonation,   // уровень замещения
  &hDuplicate))    // адрес дубликата маркера доступа
{
  dwRetCode = GetLastError();
  printf( "Duplicate token failed: %u\n", dwRetCode);

  return dwRetCode;
}

// запускаем поток
hThread = CreateThread(
  NULL,            // защита по умолчанию
  0,               // размер стека по умолчанию
  thread,          // адрес программы потока
  (LPVOID) lpSid,  // адрес проверяемого SID
  CREATE_SUSPENDED,  // подвешенное состояние потока
  &IDThread);      // идентификатор потока
if (hThread == NULL)
{
  printf("Create thread failed.\n");
  return GetLastError();
}
```

```
  // устанавливаем замещающий маркер доступа для потока
  if (!SetThreadToken(
    &hThread,      // адрес дескриптора потока
    hDuplicate))   // дубликат маркера доступа
  {
    dwRetCode = GetLastError();
    printf( "Set thread token failed: %u\n", dwRetCode);

    return dwRetCode;
  }
  // возобновляем поток
  ResumeThread(hThread);
  // ждем завершение потока
  WaitForSingleObject(hThread, INFINITE);
  CloseHandle(hToken);
  CloseHandle(hThread);

  return 0;
}
```

Глава 44

Работа со списками управления доступом на низком уровне

В этой главе будут рассмотрены функции для работы со списками управления доступом на низком уровне. То есть программа сама должна инициализировать список управления доступом, элементы списка управления доступом и самостоятельно определять порядок включения элементов в список управления доступом. Работа со списками управления доступом на высоком уровне была рассмотрена в *гл. 41.*

44.1. Структура списка управления доступом

При работе со списком управления доступом на низком уровне сам список хранится в непрерывной области памяти, которая структурирована следующим образом. Начинается список управления доступом с заголовка, за которым следуют элементы, контролирующие доступ к объекту. Заголовок списка управления доступом имеет следующую структуру:

```
typedef struct  _ACL {
  BYTE  AclRevision;   // версия списка управления доступом
  BYTE  Sbz1;          // равен 0
  WORD  AclSize;       // длина списка управления доступом в байтах
  WORD  AceCount;      // количество элементов в списке
  WORD  Sbz2;          // равен 0
}  ACL;
```

Также определяется тип:

```
typedef ACL *PACL;
```

В настоящее время версия списка управления доступом определяется символической константой `ACL_REVISION`, которая и должна быть установлена в поле `AclRevision`.

Структура элементов списка управления доступом будет описана в следующем разделе.

44.2. Структура элемента списка управления доступом

Система безопасности операционной системы Windows 2000 поддерживает следующие три типа элементов, которые могут храниться в списках управления доступом любого охраняемого объекта:

- ACCESS_ALLOWED_ACE — разрешает доступ к объекту, хранится в списке DACL;
- ACCESS_DENIED_ACE — запрещает доступ к объекту, хранится в списке DACL;
- SYSTEM_AUDIT_ACE — обеспечивает генерацию аудиторской записи при доступе к объекту, хранится в списке SACL

Каждому из этих типов соответствует определенный тип структуры. Определения типов этих структур следующие:

```
typedef struct _ACCESS_ALLOWED_ACE {
  ACE_HEADER   Header;    // заголовок элемента
  ACCESS_MASK  Mask;      // маска доступа
  DWORD        SidStart;  // начало SID
} ACCESS_ALLOWED_ACE;

typedef struct _ACCESS_DENIED_ACE {
  ACE_HEADER   Header;    // заголовок элемента
  ACCESS_MASK  Mask;      // маска доступа
  DWORD        SidStart;  // начало SID
} ACCESS_DENIED_ACE;

typedef struct _SYSTEM_AUDIT_ACE {
  ACE_HEADER   Header;    // заголовок элемента
  ACCESS_MASK  Mask;      // маска доступа
  DWORD        SidStart;  // начало SID
} SYSTEM_AUDIT_ACE;
```

Как видим, хотя все эти структуры имеют разные типы, но их внутренняя структура одинакова. Опишем кратко назначение каждого поля этих структур.

В поле Header элемента списка управления хранятся тип, управляющие флаги и размер элемента списка управления доступом. Это поле имеет тип структуры ACE_HEADER, которая будет подробно рассмотрена далее.

В поле Mask элемента списка управления доступом устанавливаются права доступа к охраняемому объекту, которые контролирует данный элемент. Само поле имеет тип:

```
typedef DWORD ACCESS_MASK;
```

То есть является типом DWORD, в котором установлены управляющие флаги, определяющие права доступа. Все права доступа к объекту делятся на три категории: специфические права доступа, стандартные права доступа и родовые права доступа. В этом поле может быть установлена любая комбинация флагов, рассмотренных далее.

Для установки специфических прав доступа используется флаг SPECIFIC_RIGHTS_ALL, который включает все специфические права доступа.

Остальные флаги, используемые для специфических прав доступа, определяются конкретно для каждого типа объекта.

Стандартные права доступа задаются следующими символьными константами:

- DELETE — право удаления объекта;
- READ_CONTROL — право чтения управляющей информации из дескриптора безопасности, исключая содержимое списка DACL;
- SYNCHRONIZE — право использования объекта для синхронизации;
- WRITE_DAC — право записи в список DACL объекта;
- WRITE_OWNER — право записи владельца объекта.

Кроме того, для стандартных прав доступа определены символьные константы, которые включают несколько основных прав доступа:

- STANDARD_RIGHTS_ALL — включает все права доступа;
- STANDARD_RIGHTS_REQUIRED — включает права DELETE, READ_CONTROL, WRITE_DAC, WRITE_OWNER;
- STANDARD_RIGHTS_READ — эквивалентно READ_CONTROL;
- STANDARD_RIGHTS_WRITE — эквивалентно READ_CONTROL;
- STANDARD_RIGHTS_EXECUTE — эквивалентно READ_CONTROL.

Для установки родовых прав доступа используются следующие символьные константы:

- GENERIC_READ — право читать объект;
- GENERIC_WRITE — право записывать объект;
- GENERIC_EXECUTE — право исполнять объект.

Следующее родовое право включает все основные родовые права доступа:

- GENERIC_ALL — включает права GENERIC_READ, GENERIC_WRITE и GENERIC_EXECUTE.

Права доступа к списку SACL включают единственное право ACCESS_SYSTEM_SECURITY — право доступа к списку SACL.

Поле SidStart представляет собой переменную типа DWORD, которая определяет начало идентификатора безопасности, доступ которого к объекту контролирует данный элемент списка управления доступом. То есть это поле фактически содержит первые 32 бита идентификатора безопасности из элемента списка управления доступом.

Теперь подробно опишем структуру типа ACE_HEADER, которая служит для описания заголовка элемента списка управления доступом. Тип этой структуры определяется следующим образом:

```
typedef struct _ACE_HEADER {
  BYTE  AceType;     // тип элемента
  BYTE  AceFlags;    // флаги наследования и аудита
  WORD  AceSize;     // длина элемента
} ACE_HEADER;
```

Кратко опишем назначение каждого поля этой структуры.

В поле AceType должен быть установлен тип элемента списка управления доступом. Этот тип задается одной из следующих констант:

- ACCESS_ALLOWED_ACE_TYPE — элемент типа ACCESS_ALLOWED_ACE;
- ACCESS_DENIED_ACE_TYPE — элемент типа ACCESS_DENIED_ACE;
- SYSTEM_AUDIT_ACE_TYPE — элемент типа SYSTEM_AUDIT_ACE.

В поле AceFlags устанавливаются флаги, управляющие наследованием элемента списка управления доступом при создании дочернего охраняемого объекта и аудитом доступа к охраняемому объекту. В этом поле может быть установлена комбинация из следующих флагов, управляющих наследованием элементов списка управления доступом:

- OBJECT_INHERIT_ACE — элемент наследуется не контейнерным дочерним объектом; если не установлен флаг NO_PROPAGATE_INHERIT_ACE, то элемент также наследуется и контейнерным дочерним объектом, но при этом в дочернем объекте устанавливается флаг INHERIT_ONLY_ACE;
- CONTAINER_INHERIT_ACE — элемент наследуется контейнерным дочерним объектом;
- NO_PROPAGATE_INHERIT_ACE — элемент не наследуется;
- INHERIT_ONLY_ACE — отмечает, что элемент был унаследован от родительского объекта; этот элемент не участвует в контроле доступа к объекту;
- INHERITED_ACE — отмечает, что элемент был унаследован от родительского объекта.

И флагов управляющих аудитом доступа к охраняемому объекту:

- SUCCESSFUL_ACCESS_ACE_FLAG — отмечает, что нужно генерировать аудиторское сообщение при успешном доступе к объекту;
- FAILED_ACCESS_ACE_FLAG — отмечает, что нужно генерировать аудиторское сообщение при неудачной попытке доступа к объекту.

Отметим, что два последних флага используются только в элементах, принадлежащих спискам управления доступом SACL.

В поле AceSize хранится длина (в байтах) самого элемента из списка управления доступом.

44.3. Инициализация списка управления доступом

Инициализация списка управления доступом заключается в задании размера буфера памяти, которую может занять список управления доступом, и установки версии списка управления доступом. Фактически инициализация списка управления доступом заключается в заполнении данными заголовка этого списка.

Для инициализации списка управления доступом используется функция InitializeAcl, которая имеет следующий прототип:

```
BOOL  InitializeAcl(
  PACL   pAcl,              // адрес буфера для ACL
  DWORD  nAclLength,        // длина буфера
  DWORD  dwAclRevision      // версия ACL
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError. Параметры функции имеют следующее назначение.

Параметр pAcl должен указывать на область памяти, в которой будет храниться список управления доступом. В начало этой памяти будет записан заголовок списка управления доступом.

Параметр nAclLength должен содержать длину памяти, выделенной для хранения списка управления доступом.

Параметр dwAclRevision должен содержать версию списка управления доступом. В настоящее время эта версия задается символьной константой ACL_REVISION.

В заключение этого раздела сделаем несколько замечаний относительно вычисления длины памяти, необходимой для списка управления доступом.

Так как список управления доступом содержит заголовок и элементы, то его длина соответственно равна сумме длин заголовка и всех элементов. Но при этом надо учитывать, что длина каждого элемента зависит от длины идентификатора безопасности, который хранится в этом элементе. В свою очередь длина идентификатора безопасности должна быть уменьшена на длину типа `DWORD`, т. е. длину поля `SidStart`, которое определяет в элементе списка управления доступом начала идентификатора безопасности. Например, длина списка управления доступом, который содержит два элемента (один из которых контролирует разрешение доступа, а второй — запрещение доступа), может вычисляться следующим образом:

```
dwAclLength = sizeof(ACL)
  + sizeof(ACCESS_ALLOWED_ACE) - sizeof(DWORD) + dwLengthOfSidAllow
  + sizeof(ACCESS_DENIED_ACE) - sizeof(DWORD) + dwLengthOfSidDeny;
```

Программа, в которой для инициализации списка управления доступом используется функция `InitializeAcl`, приведена в *разд. 44.5*, посвященном добавлению элементов в список управления доступом.

44.4. Проверка достоверности списка управления доступом

Под проверкой достоверности списка управления доступом будем понимать проверку достоверности версии и длины этого списка. Длина списка управления доступом должна соответствовать количеству включенных в этот список элементов.

Для проверки достоверности списка управления доступом используется функция `IsValidAcl`, которая имеет следующий прототип:

```
BOOL  IsValidAcl(
  PACL  pAcl     // адрес списка DACL
);
```

Если проверка списка на достоверность прошла успешно, то функция вернет ненулевое значение, в противном случае — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`. Единственный параметр `pAcl` этой функции должен указывать на список управления доступом, достоверность которого будет проверяться.

Программа, в которой посредством функции `IsValidAcl` проверяется достоверность списка управления доступом, приведена в *разд. 44.5*, посвященном добавлению элементов в список управления доступом.

44.5. Добавление элементов в список управления доступом

В интерфейсе Win32 API существует семь функций, предназначенных для добавления элементов в список управления доступом на низком уровне. Уровень этих функций называется низким потому, что система не контролирует порядок элементов в списке управления доступом при включении нового элемента, используя одну из этих функций. Как правило, новый элемент включается в начало или конец списка управления доступом. Поэтому программа сама должна контролировать порядок, в котором элементы включаются в список управления доступом. Прежде чем перейти к описанию функций, просто перечислим их, попутно определяя их функциональное назначение. Затем подробно опишем каждую из функций.

Для добавления в список управления доступом DACL элементов, разрешающих доступ, используются функции `AddAccessAllowedAce` и `AddAccessAllowedAceEx`.

Для добавления в список управления доступом DACL элементов, запрещающих доступ, используются функции `AddAccessDeniedAce` и `AddAccessDeniedAceEx`.

Для добавления в список управления доступом SACL элементов, контролирующих доступ, используются функции `AddAuditAccessAce` и `AddAuditAccessAceEx`.

Для копирования элементов из одного списка управления доступом в другой список управления доступом используется функция `AddAce`.

Теперь перейдем к описанию каждой из перечисленных функций и примерам их использования.

Для добавления в конец списка управления доступом элемента, разрешающего доступ к объекту, используется функция `AddAccessAllowedAce`, которая имеет следующий прототип:

```
BOOL  AddAccessAllowedAce(
  PACL   pAcl,           // адрес списка управления доступом
  DWORD  dwAceRevision,  // версия элемента списка управления доступом
  DWORD  AccessMask,     // маска доступа
  PSID   pSid            // адрес идентификатора безопасности
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`. Возможны следующие коды ошибок, возвращаемые функцией `GetLastError`, после завершения работы функции `AddAccessAllowedAce`:

- `ERROR_ALLOTTED_SPACE_EXCEEDED` — недостаточная длина буфера для ACL;
- `ERROR_INVALID_ACL` — неправильный ACL;

- ERROR_INVALID_SID — неправильный SID;
- ERROR_REVISION_MISMATCH — неправильная версия ACE;
- ERROR_SUCCESS — удачное завершение функции.

Параметр pAcl должен указывать на список управления доступом, в который добавляется элемент, контролирующий разрешение доступа субъекта к объекту

Параметр dwAclRevision должен содержать версию элемента, включаемого в список управления доступом. В настоящее время эта версия задается символьной константой ACL_REVISION.

Параметр AccessMask должен содержать разрешенные для учетной записи права доступа к охраняемому объекту, с которым связан данный список управления доступом. Идентификатор безопасности учетной записи задается параметром pSid. Подробно возможные значения флагов для стандартных и родовых прав доступа описаны в *разд. 36.7*, *41.3*. Специфические права доступа к объектам описаны в соответствующих разделах *гл. 45*, посвященных защите этих объектов.

Параметр pSid должен указывать на идентификатор безопасности учетной записи, которая наделяется этими правами.

Пример использования функции AddAccessAllowedAce показан в программе из листинга 44.1. В этой программе создается новый каталог со списком управления доступом DACL. Для добавления в список элемента, контролирующего разрешение доступа к каталогу, используется функция AddAccessAllowedAce.

Для добавления в конец списка управления доступом элемента, запрещающего доступ к объекту, используется функция AddAccessDeniedAce, которая имеет следующий прототип:

```
BOOL AddAccessDeniedAce(
  PACL   pAcl,           // адрес списка управления доступом
  DWORD  dwAceRevision,  // версия элемента списка управления доступом
  DWORD  AccessMask,     // маска доступа
  PSID   pSid            // адрес идентификатора безопасности
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError. В этом случае функция GetLastError возвращает такие же коды, как и после завершения работы функции AddAccessAllowedAce.

Параметры функции AddAccessDeniedAce имеют такое же назначение, как и соответствующие параметры функции AddAccessAllowedAce. Но при этом

флаги, установленные в параметре AccessMask, отмечают запрещенные для учетной записи права доступа к охраняемому объекту.

В листинге 44.1 приведена программа, демонстрирующая работу функций AddAccessAllowedAce и AddAccessDeniedAce. Для этого программа создает каталог, владелец которого устанавливается по умолчанию. Для каталога создается список управления доступом DACL, который включает два элемента: один контролирующий разрешение, а второй — запрещение доступа к каталогу для учетной записи пользователя, имя которого вводится с консоли. Этот пользователь наделяется полными родовыми правами доступа к каталогу, но ему запрещается изменять владельца каталога.

Листинг 44.1. Добавление элементов в список DACL

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>
#include <lm.h>

int main()
{
  ACL *lpDacl;           // указатель на список доступа
  DWORD dwDaclLength;    // длина списка доступа

  wchar_t wchUserName[UNLEN];    // имя пользователя
  wchar_t wchDirName[248];       // имя каталога

  DWORD dwLengthOfDomainName = 0;  // длина имени домена

  DWORD dwLengthOfSid = 0;       // длина SID
  SID *lpSid = NULL;             // указатель на SID
  LPTSTR lpDomainName = NULL;    // указатель на имя домена

  SID_NAME_USE typeOfSid;    // тип учетной записи

  SECURITY_DESCRIPTOR sd;    // дескриптор безопасности каталога
  SECURITY_ATTRIBUTES sa;    // атрибуты защиты каталога
```

```
DWORD dwRetCode;  // код возврата

// вводим имя пользователя, которому разрешим доступ к каталогу
printf("Input a user name: ");
_getws(wchUserName);

// определяем длину SID пользователя
if (!LookupAccountName(
  NULL,                  // ищем имя на локальном компьютере
  wchUserName,           // имя пользователя
  NULL,                  // определяем длину SID
  &dwLengthOfSid,        // длина SID
  NULL,                  // определяем имя домена
  &dwLengthOfDomainName, // длина имени домена
  &typeOfSid))           // тип учетной записи
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
  {
    // распределяем память для SID и имени домена
    lpSid = (SID*) new char[dwLengthOfSid];
    lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];
  }
  else
  {
    // выходим из программы
    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// определяем SID и имя домена пользователя
if(!LookupAccountName(
  NULL,                  // ищем имя на локальном компьютере
  wchUserName,           // имя пользователя
  lpSid,                 // указатель на SID
```

```
  &dwLengthOfSid,        // длина SID
  lpDomainName,          // указатель на имя домена
  &dwLengthOfDomainName, // длина имени домена
  &typeOfSid))           // тип учетной записи
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// определем длину списка DACL
dwDaclLength = sizeof(ACL)
  + sizeof(ACCESS_ALLOWED_ACE) - sizeof(DWORD) + dwLengthOfSid
  + sizeof(ACCESS_DENIED_ACE) - sizeof(DWORD) + dwLengthOfSid;

// распределяем память под DACL
lpDacl = (ACL*)new char[dwDaclLength];

// инициализируем список DACL
if (!InitializeAcl(
  lpDacl,          // адрес DACL
  dwDaclLength,    // длина DACL
  ACL_REVISION))   // версия DACL
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// добавляем запрещающий элемент в список DACL
if (!AddAccessDeniedAce(
  lpDacl,        // адрес DACL
  ACL_REVISION,  // версия DACL
```

```
  WRITE_OWNER,    // запрещаем изменять владельца объекта
  lpSid))         // адрес SID
{
  dwRetCode = GetLastError();
  perror("Add access denied ace failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// добавляем разрешающий элемент в список DACL
if (!AddAccessAllowedAce(
  lpDacl,         // адрес DACL
  ACL_REVISION,   // версия DACL
  GENERIC_ALL,    // разрешаем все родовые права доступа
  lpSid))         // адрес SID
{
  dwRetCode = GetLastError();
  perror("Add access allowed ace failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// инициализируем версию дескриптора безопасности
if (!InitializeSecurityDescriptor(
  &sd,
  SECURITY_DESCRIPTOR_REVISION))
{
  dwRetCode = GetLastError();
  printf("Initialize security descroptor failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// устанавливаем SID владельца объекта
if (!SetSecurityDescriptorOwner(
  &sd,            // адрес дескриптора безопасности
```

```
  NULL,          // не задаем владельца
  SE_OWNER_DEFAULTED))   // определить владельца по умолчанию
{
  dwRetCode = GetLastError();
  perror("Set security descriptor owner failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// устанавливаем SID первичной группы владельца
if (!SetSecurityDescriptorGroup(
  &sd,           // адрес дескриптора безопасности
  NULL,          // не задаем первичную группу
  SE_GROUP_DEFAULTED))   // определить первичную группу по умолчанию
{
  dwRetCode = GetLastError();
  perror("Set security descriptor group failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}
// устанавливаем DACL в дескриптор безопасности
if (!SetSecurityDescriptorDacl(
  &sd,         // адрес дескриптора безопасности
  TRUE,        // DACL присутствует
  lpDacl,      // указатель на DACL
  FALSE))      // DACL не задан по умолчанию
{
  dwRetCode = GetLastError();
  perror("Set security descriptor group failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// проверяем структуру дескриптора безопасности
if (!IsValidSecurityDescriptor(&sd))
{
  dwRetCode = GetLastError();
```

```
    perror("Security descriptor is invalid.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  // инициализируем атрибуты безопасности
  sa.nLength = sizeof(sa);       // устанавливаем длину атрибутов защиты
  sa.lpSecurityDescriptor = &sd;   // устанавливаем адрес SD
  sa.bInheritHandle = FALSE;     // дескриптор каталога ненаследуемый

  // читаем имя создаваемого каталога
  printf("Input a directory name: ");
  _getws(wchDirName);       // вводим имя каталога

  // создаем каталог
  if (!CreateDirectory(wchDirName, &sa))
  {
    dwRetCode = GetLastError();
    perror("Security descriptor is invalid.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  printf("The directory is created.\n");

  // освобождаем память
  delete[] lpSid;
  delete[] lpDomainName;
  delete[] lpDacl;

  return 0;
}
```

Для добавления в конец списка управления доступом элемента, разрешающего доступ к объекту, может также использоваться функция `AddAccessAllowedAceEx`, которая имеет следующий прототип:

```
BOOL  AddAccessAllowedAceEx(
  PACL   pAcl,           // адрес списка управления доступом
```

```
    DWORD  dwAceRevision,  // версия элемента списка управления доступом
    DWORD  AceFlags,       // флаги наследования элемента
    DWORD  AccessMask,     // маска доступа
    PSID   pSid            // адрес идентификатора безопасности
  );
```

Эта функция отличается от функции `AddAccessAllowedAce` только тем, что позволяет также устанавливать флаги, управляющие наследованием включаемого в список управления доступом элемента. Флаги управления наследованием элемента устанавливаются в параметре `AceFlags`. Символьные константы, используемые для флагов, управляющих наследованием, были подробно рассмотрены в *разд. 36.7*, *41.3*.

Функция `AddAccessAllowedAceEx` работает точно также же, как и функция `AddAccessAllowedAce`, но в случае неудачного завершения она может вернуть дополнительный код ошибки `ERROR_INVALID_FLAGS` — неправильные флаги в параметре `AceFlags`.

Кроме того, заметим, что эта функция работает только в операционных системах Windows, начиная с версии Windows 2000.

Пример использования функции `AddAccessAllowedAceEx` показан в программе из листинга 44.2. В этой программе создается новый каталог со списком управления доступом DACL, в который включается один элемент, контролирующий разрешение на доступ к каталогу. Для включения этого элемента в список используется функция `AddAccessAllowedAceEx`.

Для добавления в конец списка управления доступом элемента, запрещающего доступ к объекту, может также использоваться функция `AddAccessDeniedAceEx`, которая имеет следующий прототип:

```
BOOL AddAccessDeniedAceEx(
  PACL   pAcl,           // адрес списка управления доступом
  DWORD  dwAceRevision,  // версия элемента списка управления доступом
  DWORD  AceFlags,       // флаги наследования элемента
  DWORD  AccessMask,     // маска доступа
  PSID   pSid            // адрес идентификатора безопасности
);
```

Эта функция отличается от функции `AddAccessDeniedAce` только тем, что позволяет также устанавливать флаги, управляющие наследованием включаемого в список управления доступом элемента. Флаги управления наследованием элемента устанавливаются в параметре `AceFlags`. Символьные константы, используемые для флагов, управляющих наследованием, были подробно рассмотрены в *разд. 36.7*, *41.3*.

Функция `AddAccessDeniedAceEx` работает точно также же, как и функция `AddAccessDeniedAce`, но в случае неудачного завершения она может вернуть

дополнительный код ошибки ERROR_INVALID_FLAGS — неправильные флаги в параметре AceFlags.

Кроме того, заметим, что эта функция работает только в операционных системах Windows, начиная с версии Windows 2000.

В листинге 44.2 приведена программа, демонстрирующая работу функций AddAccessAllowedAceEx и AddAccessDeniedAceEx. Функционально эта программа ничем не отличается от программы из листинга 44.1, за исключением того, что для включения в список управления доступом DACL вместо функций AddAccessAllowedAce и AddAccessDeniedAce используются функции AddAccessAllowedAceEx и AddAccessDeniedAceEx соответственно.

Листинг 44.2. Добавление элементов в список DACL

```
#define _WIN32_WINNT 0x0500

#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>
#include <lm.h>

int main()
{
  ACL *lpDacl;           // указатель на список доступа
  DWORD dwDaclLength;    // длина списка доступа

  wchar_t wchUserName[UNLEN];    // имя пользователя
  wchar_t wchDirName[248];       // имя каталога

  DWORD dwLengthOfDomainName = 0;  // длина имени домена

  DWORD dwLengthOfSid = 0;   // длина SID
  SID *lpSid = NULL;         // указатель на SID
  LPTSTR lpDomainName = NULL;    // указатель на имя домена

  SID_NAME_USE typeOfSid;    // тип учетной записи

  SECURITY_DESCRIPTOR sd;    // дескриптор безопасности каталога
```

```
SECURITY_ATTRIBUTES sa;     // атрибуты защиты каталога

DWORD dwRetCode;            // код возврата

// вводим имя пользователя, которому разрешим доступ к каталогу
printf("Input a user name: ");
_getws(wchUserName);

// определяем длину SID пользователя
if (!LookupAccountName(
  NULL,             // ищем имя на локальном компьютере
  wchUserName,      // имя пользователя
  NULL,             // определяем длину SID
  &dwLengthOfSid,   // длина SID
  NULL,             // определяем имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &typeOfSid))      // тип учетной записи
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
  {
    // распределяем память для SID и имени домена
    lpSid = (SID*) new char[dwLengthOfSid];
    lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];
  }
  else
  {
    // выходим из программы
    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// определяем SID и имя домена пользователя
if(!LookupAccountName(
  NULL,             // ищем имя на локальном компьютере
```

```
  wchUserName,      // имя пользователя
  lpSid,            // указатель на SID
  &dwLengthOfSid,   // длина SID
  lpDomainName,     // указатель на имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &typeOfSid))      // тип учетной записи
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// определем длину списка DACL
dwDaclLength = sizeof(ACL)
  + sizeof(ACCESS_ALLOWED_ACE) - sizeof(DWORD) + dwLengthOfSid
  + sizeof(ACCESS_DENIED_ACE) - sizeof(DWORD) + dwLengthOfSid;

// распределяем память под DACL
lpDacl = (ACL*)new char[dwDaclLength];

// инициализируем список DACL
if (!InitializeAcl(
  lpDacl,           // адрес DACL
  dwDaclLength,     // длина DACL
  ACL_REVISION))    // версия DACL
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// добавляем запрещающий элемент в список DACL
if (!AddAccessDeniedAceEx(
```

```
  lpDacl,          // адрес DACL
  ACL_REVISION,    // версия DACL
  CONTAINER_INHERIT_ACE,   // контейнерные объекты наследуют элемент
  WRITE_OWNER,     // запрещаем изменять владельца объекта
  lpSid))          // адрес SID
{
  dwRetCode = GetLastError();
  perror("Add access denied ace failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// добавляем разрешающий элемент в список DACL
if (!AddAccessAllowedAceEx(
  lpDacl,          // адрес DACL
  ACL_REVISION,    // версия DACL
  CONTAINER_INHERIT_ACE,   // контейнерные объекты наследуют элемент
  GENERIC_ALL,     // разрешаем все родовые права доступа
  lpSid))          // адрес SID
{
  dwRetCode = GetLastError();
  perror("Add access allowed ace failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// инициализируем версию дескриптора безопасности
if (!InitializeSecurityDescriptor(
  &sd,
  SECURITY_DESCRIPTOR_REVISION))
{
  dwRetCode = GetLastError();
  printf("Initialize security descroptor failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}
```

```
// устанавливаем SID владельца объекта
if (!SetSecurityDescriptorOwner(
  &sd,        // адрес дескриптора безопасности
  NULL,       // не задаем владельца
  SE_OWNER_DEFAULTED))   // определить владельца по умолчанию
{
  dwRetCode = GetLastError();
  perror("Set security descriptor owner failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// устанавливаем SID первичной группы владельца
if (!SetSecurityDescriptorGroup(
  &sd,          // адрес дескриптора безопасности
  NULL,         // не задаем первичную группу
  SE_GROUP_DEFAULTED))   // определить первичную группу по умолчанию
{
  dwRetCode = GetLastError();
  perror("Set security descriptor group failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}
// устанавливаем DACL в дескриптор безопасности
if (!SetSecurityDescriptorDacl(
  &sd,        // адрес дескриптора безопасности
  TRUE,       // DACL присутствует
  lpDacl,     // указатель на DACL
  FALSE))     // DACL не задан по умолчанию
{
  dwRetCode = GetLastError();
  perror("Set security descriptor group failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// проверяем структуру дескриптора безопасности
```

```
  if (!IsValidSecurityDescriptor(&sd))
  {
    dwRetCode = GetLastError();
    perror("Security descriptor is invalid.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  // инициализируем атрибуты безопасности
  sa.nLength = sizeof(sa);     // устанавливаем длину атрибутов защиты
  sa.lpSecurityDescriptor = &sd;   // устанавливаем адрес SD
  sa.bInheritHandle = FALSE;   // дескриптор каталога ненаследуемый

  // читаем имя создаваемого каталога
  printf("Input a directory name: ");
  _getws(wchDirName);       // вводим имя каталога

  // создаем каталог
  if (!CreateDirectory(wchDirName, &sa))
  {
    dwRetCode = GetLastError();
    perror("Security descriptor is invalid.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  printf("The directory is created.\n");

  // освобождаем память
  delete[] lpSid;
  delete[] lpDomainName;
  delete[] lpDacl;

  return 0;
}
```

Для добавления в список управления доступом элемента, управляющего аудитом доступа к объекту, используется функция AddAuditAccessAce, которая имеет следующий прототип:

```
BOOL  AddAuditAccessAce(
  PACL   pAcl,           // адрес списка управления доступом
  DWORD  dwAceRevision,  // версия элемента списка управления доступом
  DWORD  AccessMask,     // маска доступа
  PSID   pSid            // адрес идентификатора безопасности
  BOOL   bAuditSuccess,  // признак аудита успешного доступа
  BOOL   bAuditFailure   // признак аудита неудачного доступа
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError. Возможны следующие коды ошибок, возвращаемые функцией GetLastError, после завершения работы функции AddAccessAllowedAce:

- ERROR_ALLOTTED_SPACE_EXCEEDED — недостаточная длина буфера для ACL;
- ERROR_INVALID_ACL — неправильный ACL;
- ERROR_INVALID_SID — неправильный SID;
- ERROR_REVISION_MISMATCH — неправильная версия ACE;
- ERROR_SUCCESS — удачное завершение функции.

Параметры функции имеют следующее назначение.

Параметр pAcl должен указывать на список управления доступом, в который добавляется элемент, контролирующий разрешение доступа субъекта к объекту.

Параметр dwAclRevision должен содержать версию элемента, включаемого в список управления доступом. В настоящее время эта версия задается символьной константой ACL_REVISION.

В параметре AccessMask должны быть установлены права доступа к охраняемому объекту, которые подвергаются аудиту в случае доступа учетной записи к объекту. Идентификатор безопасности учетной записи задается параметром pSid. Подробно возможные значения флагов для стандартных и родовых прав доступа описаны в *разд. 36.7*, *41.3*. Специфические права доступа к объектам описаны в соответствующих разделах *гл. 45*, посвященных защите этих объектов.

Параметр pSid должен указывать на идентификатор безопасности учетной записи, доступ которой к охраняемому объекту подвергается аудиту.

В параметре bAuditSuccess должно быть установлено одно из булевых значений TRUE или FALSE. Если установлено значение TRUE, то элемент контро-

лирует успешный доступ учетной записи к объекту, с которым связан список SACL. В противном случае аудит успешного доступа к объекту не проводится.

В параметре `bAuditFailure` также должно быть установлено одно из булевых значений `TRUE` или `FALSE`. Если установлено значение `TRUE`, то элемент контролирует неудачный доступ учетной записи к объекту, с которым связан список SACL. В противном случае аудит неудачного доступа к объекту не проводится.

В листинге 44.3 приведена программа, в которой создается новый список управления аудитом SACL. Для включения элемента в список управления аудитом используется функция `AddAccessAllowedAce`. Фактически эта программа является расширением предыдущей программы. Отличие заключается в том, что для пользователя, которому запрещается устанавливать нового владельца каталога, в список SACL включается элемент, который управляет аудитом неудачных попыток изменения владельца каталога этим пользователем. В связи с этой программой сделаем следующие замечания. Для создания списка SACL требуется, чтобы пользователь имел действующую привилегию `SE_SECURITY_NAME`. Такую привилегию имеют администраторы системы, но по умолчанию она недействительна. Поэтому прежде чем работать со списком SACL эта привилегия должна быть активизирована. Для этого используется функция `AdjustTokenPrivileges`. Более подробно работа с привилегиями рассмотрена в *гл. 43*.

Листинг 44.3. Добавление элементов в списки DACL и SACL

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>
#include <lm.h>

int main()
{
  HANDLE hProcess;       // дескриптор процесса
  HANDLE hTokenHandle;   // дескриптор маркера доступа

  TOKEN_PRIVILEGES tp;   // привилегии маркера доступа

  ACL *lpDacl;           // указатель на список доступа DACL
```

```
DWORD dwDaclLength;    // длина списка доступа DACL

ACL *lpSacl;           // указатель на список доступа SACL
DWORD dwSaclLength;    // длина списка доступа SACL

wchar_t wchUserName[UNLEN];    // имя пользователя
wchar_t wchDirName[248];       // имя каталога

DWORD dwLengthOfDomainName = 0;  // длина имени домена

DWORD dwLengthOfSid = 0;   // длина SID
SID *lpSid = NULL;         // указатель на SID
LPTSTR lpDomainName = NULL;    // указатель на имя домена

SID_NAME_USE typeOfSid;    // тип учетной записи

SECURITY_DESCRIPTOR sd;    // дескриптор безопасности каталога
SECURITY_ATTRIBUTES sa;    // атрибуты защиты каталога

DWORD dwRetCode;           // код возврата

// получаем дескриптор процесса
hProcess = GetCurrentProcess();

// получаем маркер доступа процесса
if (!OpenProcessToken(
  hProcess,            // дескриптор процесса
  TOKEN_ALL_ACCESS,    // полный доступ к маркеру доступа
  &hTokenHandle))      // дескриптор маркера доступа
{
  dwRetCode = GetLastError();
  printf( "Open process token failed: %u\n", dwRetCode);

  return dwRetCode;
}

// устанавливаем общее количество привилегий
tp.PrivilegeCount = 1;

// определяем идентификатор привилегии для установки аудита
```

```
if (!LookupPrivilegeValue(
  NULL,      // ищем идентификатор привилегии на локальном компьютере
  SE_SECURITY_NAME,  // привилегия для аудита
  &(tp.Privileges[0].Luid)))
{
  dwRetCode = GetLastError();
  printf("Lookup privilege value failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// разрешаем привилегию аудита
tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

// разрешаем привилегию для установки аудита
if (!AdjustTokenPrivileges(
  hTokenHandle,    // дескриптор маркера доступа процесса
  FALSE,     // не запрещаем все привилегии
  &tp,       // адрес привилегий
  0,         // длины буфера нет
  NULL,      // предыдущее состояние привилегий не нужно
  NULL))     // длина буфера не нужна
{
  dwRetCode = GetLastError();
  printf("Lookup privilege value failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// вводим имя пользователя, которому разрешим доступ к каталогу
printf("Input a user name: ");
_getws(wchUserName);

// определяем длину SID пользователя
if (!LookupAccountName(
  NULL,            // ищем имя на локальном компьютере
  wchUserName,     // имя пользователя
```

```
  NULL,            // определяем длину SID
  &dwLengthOfSid,  // длина SID
  NULL,            // определяем имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &typeOfSid))     // тип учетной записи
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
  {
    // распределяем память для SID и имени домена
    lpSid = (SID*) new char[dwLengthOfSid];
    lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];
  }
  else
  {
    // выходим из программы
    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// определяем SID и имя домена пользователя
if(!LookupAccountName(
  NULL,              // ищем имя на локальном компьютере
  wchUserName,       // имя пользователя
  lpSid,             // указатель на SID
  &dwLengthOfSid,    // длина SID
  lpDomainName,      // указатель на имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &typeOfSid))       // тип учетной записи
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
```

```
}

// определяем длину списка DACL
dwDaclLength = sizeof(ACL)
  + sizeof(ACCESS_ALLOWED_ACE) - sizeof(DWORD) + dwLengthOfSid
  + sizeof(ACCESS_DENIED_ACE) - sizeof(DWORD) + dwLengthOfSid;

// распределяем память под DACL
lpDacl = (ACL*)new char[dwDaclLength];

// инициализируем список DACL
if (!InitializeAcl(
  lpDacl,          // адрес DACL
  dwDaclLength,    // длина DACL
  ACL_REVISION))   // версия DACL
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// добавляем запрещающий элемент в список DACL
if (!AddAccessDeniedAce(
  lpDacl,        // адрес DACL
  ACL_REVISION,  // версия DACL
  WRITE_OWNER,   // запрещаем изменять владельца объекта
  lpSid))        // адрес SID
{
  dwRetCode = GetLastError();
  perror("Add access denied ace failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// добавляем разрешающий элемент в список DACL
```

```
if (!AddAccessAllowedAce(
  lpDacl,        // адрес DACL
  ACL_REVISION,  // версия DACL
  GENERIC_ALL,   // разрешаем все родовые права доступа
  lpSid))        // адрес SID
{
  dwRetCode = GetLastError();
  perror("Add access allowed ace failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// определяем длину списка SACL
dwSaclLength = sizeof(ACL)
  + sizeof(SYSTEM_AUDIT_ACE ) - sizeof(DWORD) + dwLengthOfSid;

// распределяем память под SACL
lpSacl = (ACL*)new char[dwSaclLength];

// инициализируем список SACL
if (!InitializeAcl(
  lpSacl,          // адрес SACL
  dwSaclLength,    // длина SACL
  ACL_REVISION))   // версия SACL
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// добавляем элемент в список SACL
if (!AddAuditAccessAce(
  lpSacl,        // адрес DACL
  ACL_REVISION,  // версия DACL
  WRITE_OWNER,   // запрещаем изменять владельца объекта
```

```
  lpSid,          // адрес SID
  FALSE,          // не нужен аудит удачного доступа
  TRUE))          // нужен аудит неудачного доступа
{
  dwRetCode = GetLastError();
  perror("Add audit access ace failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// инициализируем версию дескриптора безопасности
if (!InitializeSecurityDescriptor(
  &sd,
  SECURITY_DESCRIPTOR_REVISION))
{
  dwRetCode = GetLastError();
  printf("Initialize security descroptor failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// устанавливаем SID владельца объекта
if (!SetSecurityDescriptorOwner(
  &sd,          // адрес дескриптора безопасности
  NULL,         // не задаем владельца
  SE_OWNER_DEFAULTED))   // определить владельца по умолчанию
{
  dwRetCode = GetLastError();
  perror("Set security descriptor owner failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// устанавливаем SID первичной группы владельца
if (!SetSecurityDescriptorGroup(
  &sd,          // адрес дескриптора безопасности
```

```
  NULL,        // не задаем первичную группу
  SE_GROUP_DEFAULTED))   // определить первичную группу по умолчанию
{
  dwRetCode = GetLastError();
  perror("Set security descriptor group failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}
// устанавливаем DACL в дескриптор безопасности
if (!SetSecurityDescriptorDacl(
  &sd,       // адрес дескриптора безопасности
  TRUE,      // DACL присутствует
  lpDacl,    // указатель на DACL
  FALSE))    // DACL не задан по умолчанию
{
  dwRetCode = GetLastError();
  perror("Set security descriptor group failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}
// устанавливаем SACL в дескриптор безопасности
if (!SetSecurityDescriptorSacl(
  &sd,       // адрес дескриптора безопасности
  TRUE,      // SACL присутствует
  lpSacl,    // указатель на SACL
  FALSE))    // SACL не задан по умолчанию
{
  dwRetCode = GetLastError();
  perror("Set security descriptor group failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// проверяем структуру дескриптора безопасности
if (!IsValidSecurityDescriptor(&sd))
{
```

```
  dwRetCode = GetLastError();
  perror("Security descriptor is invalid.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// инициализируем атрибуты безопасности
sa.nLength = sizeof(sa);     // устанавливаем длину атрибутов защиты
sa.lpSecurityDescriptor = &sd;     // устанавливаем адрес дескриптора
                                   // безопасности
sa.bInheritHandle = FALSE;   // дескриптор каталога не наследуемый

// читаем имя создаваемого каталога
printf("Input a directory name: ");
_getws(wchDirName);          // вводим имя каталога

// создаем каталог
if (!CreateDirectory(wchDirName, &sa))
{
  dwRetCode = GetLastError();
  perror("Security descriptor is invalid.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

printf("The directory is created.\n");

// запрещаем привилегию аудита
tp.Privileges[0].Attributes = 0;

// разрешаем привилегию для установки аудита
if (!AdjustTokenPrivileges(
  hTokenHandle,    // дескриптор маркера доступа процесса
  FALSE,           // не запрещаем все привилегии
  &tp,             // адрес привилегий
  0,               // длины буфера нет
  NULL,            // предыдущее состояние привилегий не нужно
```

```
    NULL))            // длина буфера не нужна
  {
    dwRetCode = GetLastError();
    printf("Lookup privilege value failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }

  // освобождаем память
  delete[] lpSid;
  delete[] lpDomainName;
  delete[] lpDacl;

  return 0;
}
```

Для добавления в список управления доступом элемента, управляющего аудитом доступа к объекту, может также использоваться функция `AddAuditAccessAceEx`, которая имеет следующий прототип:

```
BOOL AddAuditAccessAceEx(
  PACL   pAcl,           // адрес списка управления доступом
  DWORD  dwAceRevision,  // версия элемента списка управления доступом
  DWORD  AceFlags,       // флаги наследования и аудита
  DWORD  AccessMask,     // маска доступа
  PSID   pSid            // адрес идентификатора безопасности
  BOOL   bAuditSuccess,  // признак аудита успешного доступа
  BOOL   bAuditFailure   // признак аудита неудачного доступа
);
```

Эта функция отличается от функции `AddAuditAccessAce` только тем, что позволяет также устанавливать флаги, управляющие наследованием включаемого в список управления доступом элемента. Флаги управления наследованием элемента устанавливаются в параметре `AceFlags`. Символьные константы, используемые для флагов, управляющих наследованием, были подробно рассмотрены в *разд. 36.7*, *41.3*. В этом параметре также могут быть установлены флаги, управляющие аудитом. Таким образом, флаги аудита успешного или неудачного доступа к объекту могут быть установлены как при помощи параметров `bAuditSuccess` и `bAuditFailure`, так и параметра `AceFlags`.

Функция AddAuditAccessAceEx работает точно также же, как и функция AddAuditAccessAce, но в случае неудачного завершения она может вернуть дополнительный код ошибки:

- ERROR_INVALID_FLAGS — неправильные флаги в параметре AceFlags.

Кроме того, отметим, что эта функция работает только в операционных системах Windows, начиная с версии Windows 2000.

Для копирования элементов из одного списка управления доступом в другой список управления доступом используется функция AddAce, которая имеет следующий прототип:

```
BOOL  AddAce(
  PACL   pAcl,                // адрес списка управления доступом
  DWORD  dwAceRevision,       // версия списка управления доступом
  DWORD  dwStartingAceIndex,  // индекс первого включаемого элемента
  LPVOID pAceList,            // список включаемых элементов
  DWORD  nAceListLength       // длина списка включаемых элементов
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError. Возможны следующие коды ошибок, возвращаемые функцией GetLastError, после завершения работы функции AddAce:

- ERROR_ALLOTTED_SPACE_EXCEEDED — недостаточная длина буфера для ACL;
- ERROR_INVALID_ACL — неправильный ACL;
- ERROR_INVALID_SID — неправильный SID;
- ERROR_REVISION_MISMATCH — неправильная версия ACE;
- ERROR_SUCCESS — удачное завершение функции.

Параметр pAcl должен указывать на список управления доступом, в который добавляются элементы, контролирующие доступ субъекта к охраняемому объекту

Параметр dwAclRevision должен содержать версию списка управления доступом, в который добавляются элементы. В настоящее время эта версия задается символьной константой ACL_REVISION.

Параметр dwStartingAceIndex должен содержать индекс, с которого начинают добавляться элементы в список управления доступом. Если в этом параметре установлено значение 0, то элементы включаются в начало списка. Если в этом параметре установлено значение MAXDWORD, то элементы добавляются в конец списка.

Параметр pAceList должен указывать на список элементов, которые включаются в список управления доступом. Список элементов должен храниться

в последовательной области памяти. Отметим, что параметр pAceList должен указывать не на структуру типа ACL, а на элемент в списке управления доступом. Этот элемент может быть произвольным элементом списка управления доступом.

Параметр nAceListLength должен содержать длину списка элементов, на который указывает параметр pAceList.

В *разд. 45.5* приведена программа, в которой функция AddAce используется для копирования элементов одного списка управления доступом в другой список.

44.6. Получение элементов из списка управления доступом

Для получения адреса элемента списка управления доступом используется функция GetAce, которая имеет следующий прототип:

```
BOOL  GetAce(
  PACL   pAcl,          // адрес списка управления доступом
  DWORD  dwAceIndex,    // индекс элемента
  LPVOID *pAce          // указатель на переменную для адреса элемента
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError. Параметры функции имеют следующее назначение.

Параметр pAcl должен указывать на область памяти, в которой хранится список управления доступом.

Параметр dwAceIndex должен содержать индекс элемента из списка управления доступом, адрес которого определяет функция.

Параметр pAce должен содержать адрес указателя, в который функция запишет адрес элемента с индексом dwAceIndex из списка управления доступом.

Напомним, что количество элементов, находящихся в списке управления доступом, хранится в поле AceCount структуры типа ACL, которая была рассмотрена в *разд. 44.1*. Для определения количества элементов, включенных в список управления доступом, может также использоваться функция GetAclInformation, которая описана в *разд. 44.8*.

В листинге 44.4 приведена программа, которая перечисляет элементы списка управления доступом DACL, используя для этого функцию GetAce.

Листинг 44.4. Перечисление элементов списка управления доступом DACL

```
#define _WIN32_WINNT 0x0500

#include <windows.h>
#include <stdio.h>
#include <sddl.h>

int main()
{
  char chDirName[248];   // имя файла

  PSECURITY_DESCRIPTOR lpSd = NULL;  // указатель на SD

  PACL lpDacl = NULL;    // указатель на список управления доступом
  BOOL bDaclPresent;     // признак присутствия списка DACL
  BOOL bDaclDefaulted;   // признак списка DACL по умолчанию

  void *lpAce = NULL;    // указатель на элемент списка
  LPTSTR StringSid;      // указатель на строку SID

  DWORD dwLength;        // длина дескриптора безопасности
  DWORD dwRetCode;       // код возврата

  // вводим имя файла
  printf("Input a directory or file name: ");
  gets(chDirName);

  // получаем длину дескриптора безопасности
  if (!GetFileSecurity(
    chDirName,     // имя файла
    DACL_SECURITY_INFORMATION,   // получаем DACL
    lpSd,          // адрес дескриптора безопасности
    0,             // определяем длину буфера
    &dwLength))    // адрес для требуемой длины
  {
    dwRetCode = GetLastError();

    if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
```

```
    // распределяем память для буфера
    lpSd = (SECURITY_DESCRIPTOR*) new char[dwLength];
  else
  {
    // выходим из программы
    printf("Get file security failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// распределяем память для дескриптора безопасности
lpSd = (PSECURITY_DESCRIPTOR) new char[dwLength];

// читаем дескриптор безопасности
if (!GetFileSecurity(
  chDirName,     // имя файла
  DACL_SECURITY_INFORMATION,   // получаем DACL
  lpSd,          // адрес дескриптора безопасности
  dwLength,      // длину буфера
  &dwLength))    // адрес для требуемой длины
{
  dwRetCode = GetLastError();
  printf("Get file security failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// получаем список DACL из дескриптора безопасности
if (!GetSecurityDescriptorDacl(
  lpSd,              // адрес дескриптора безопасности
  &bDaclPresent,     // признак присутствия списка DACL
  &lpDacl,           // адрес указателя на DACL
  &bDaclDefaulted))  // признак списка DACL по умолчанию
{
```

```
    dwRetCode = GetLastError();
    printf("Get security descriptor DACL failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }

  // проверяем, есть ли DACL
  if (!bDaclPresent)
  {
    printf("Dacl is not present.");

    return 0;
  }

  // печатаем количество элементов
  printf("Ace count: %u\n", lpDacl->AceCount);

  // получаем элементы списка DACL
  for (unsigned i = 0; i < lpDacl->AceCount; ++i)
  {
    if (!GetAce(
      lpDacl,  // адрес DACL
      i,       // индекс элемента
      &lpAce)) // указатель на элемент списка
    {
      dwRetCode = GetLastError();
      printf("Get ace failed.\n");
      printf("Error code: %d\n", dwRetCode);

      return dwRetCode;
    }
    // выводим на печать тип элемента и SID
    if (((ACE_HEADER*)lpAce)->AceType == ACCESS_ALLOWED_ACE_TYPE)
    {
      printf("ACCESS_ALLOWED_ACE_TYPE\n");
      // преобразуем SID в строку
```

```
      if (!ConvertSidToStringSid(
        &((ACCESS_ALLOWED_ACE*)lpAce)->SidStart,
        &StringSid))
      {
        dwRetCode = GetLastError();
        printf("Convert sid to string sid failed.\n");
        printf("Error code: %d\n", dwRetCode);

        return dwRetCode;
      }
      printf("%s\n", StringSid);
      LocalFree(StringSid);
    }
    if (((ACE_HEADER*)lpAce)->AceType == ACCESS_DENIED_ACE_TYPE)
    {
      printf("ACCESS_DENIED_ACE_TYPE\n");
      // преобразуем SID в строку
      if (!ConvertSidToStringSid(
        &((ACCESS_DENIED_ACE*)lpAce)->SidStart,
        &StringSid))
      {
        dwRetCode = GetLastError();
        printf("Convert sid to string sid failed.\n");
        printf("Error code: %d\n", dwRetCode);

        return dwRetCode;
      }

      printf("%s\n", StringSid);
      LocalFree(StringSid);
    }
  }

  // освобождаем память
  delete[] lpSd;

  return 0;
}
```

44.7. Удаление элементов из списка управления доступом

Для удаления элемента из списка управления доступом используется функция `DeleteAce`, которая имеет следующий прототип:

```
BOOL  DeleteAce(
  PACL   pAcl,          // адрес списка управления доступом
  DWORD  dwAceIndex,    // индекс элемента
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`. Параметры функции имеют следующее назначение.

Параметр `pAcl` должен указывать на область памяти, в которой хранится список управления доступом.

Параметр `dwAceIndex` должен содержать индекс элемента, который удаляется из списка управления доступом.

Напомним, что количество элементов, находящихся в списке управления доступом, хранится в поле `AceCount` структуры типа ACL, которая была рассмотрена в *разд. 44.1*. Для определения количества элементов, включенных в список управления доступом, может также использоваться функция `GetAclInformation`, которая описана в следующем разделе.

В листинге 44.5 приведена программа, которая удаляет элемент из списка управления доступом DACL, используя для этого функцию `DeleteAce`.

Листинг 44.5. Удаление элементов из списка управления доступом DACL

```
#include <windows.h>
#include <stdio.h>

int main()
{
  char chDirName[248];  // имя файла

  PSECURITY_DESCRIPTOR lpSd = NULL;  // указатель на SD

  PACL lpDacl = NULL;     // указатель на список управления доступом
  BOOL bDaclPresent;      // признак присутствия списка DACL
  BOOL bDaclDefaulted;    // признак списка DACL по умолчанию
```

```
void *lpAce = NULL;     // указатель на элемент списка

DWORD dwLength;         // длина дескриптора безопасности
DWORD dwRetCode;        // код возврата

// вводим имя файла
printf("Input a directory or file name: ");
gets(chDirName);

// получаем длину дескриптора безопасности
if (!GetFileSecurity(
  chDirName,          // имя файла
  DACL_SECURITY_INFORMATION,   // получаем DACL
  lpSd,               // адрес дескриптора безопасности
  0,                  // определяем длину буфера
  &dwLength))         // адрес для требуемой длины
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
    // распределяем память для буфера
    lpSd = (SECURITY_DESCRIPTOR*) new char[dwLength];
  else
  {
    // выходим из программы
    printf("Get file security failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// распределяем память для дескриптора безопасности
lpSd = (PSECURITY_DESCRIPTOR) new char[dwLength];

// читаем дескриптор безопасности
if (!GetFileSecurity(
  chDirName,     // имя файла
  DACL_SECURITY_INFORMATION,   // получаем DACL
```

```
  lpSd,          // адрес дескриптора безопасности
  dwLength,      // длина буфера
  &dwLength))    // адрес для требуемой длины
{
  dwRetCode = GetLastError();
  printf("Get file security failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// получаем список DACL из дескриптора безопасности
if (!GetSecurityDescriptorDacl(
  lpSd,              // адрес дескриптора безопасности
  &bDaclPresent,     // признак присутствия списка DACL
  &lpDacl,           // адрес указателя на DACL
  &bDaclDefaulted))  // признак списка DACL по умолчанию
{
  dwRetCode = GetLastError();
  printf("Get security descriptor DACL failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// проверяем, есть ли DACL
if (!bDaclPresent)
{
  printf("Dacl is not present.");

  return 0;
}

// удаляем запрещающие доступ элементы списка DACL
for (unsigned i = 0; i < lpDacl->AceCount; ++i)
{
  // получить элемент списка DACL
  if (!GetAce(
    lpDacl,    // адрес DACL
    i,         // индекс элемента
```

```
      &lpAce))   // указатель на элемент списка
    {
      dwRetCode = GetLastError();
      printf("Get ace failed.\n");
      printf("Error code: %d\n", dwRetCode);

      return dwRetCode;
    }
    // проверяем тип элемента
    if (((ACE_HEADER*)lpAce)->AceType == ACCESS_DENIED_ACE_TYPE)
      // удаляем элемент из списка DACL
      if (!DeleteAce( lpDacl,  i))
      {
        dwRetCode = GetLastError();
        printf("Delete ace failed.\n");
        printf("Error code: %d\n", dwRetCode);

        return dwRetCode;
      }
  }
  // устанавливаем новый дескриптор безопасности
  if (!SetFileSecurity(
    chDirName,                    // имя файла
    DACL_SECURITY_INFORMATION,    // устанавливаем DACL
    lpSd))                        // адрес дескриптора безопасности
  {
    dwRetCode = GetLastError();
    printf("Set file security failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }

  printf("Denied elements are deleted from the DACL.\n");

  // освобождаем память
  delete[] lpSd;

  return 0;
}
```

44.8. Получение информации о списке управления доступом

Для получения информации о версии или размерах списка управления доступом используется функция GetAclInformation, которая имеет следующий прототип:

```
BOOL  GetAclInformation(
  PACL    pAcl,                    // адрес списка управления доступом
  LPVOID  pAclInformation,         // адрес буфера для информации
  DWORD   nAclInformationLength,   // длина буфера
  ACL_INFORMATION_CLASS  dwAclInformationClass   // тип информации
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError. Параметры функции имеют следующее назначение.

Параметр pAcl должен указывать на область памяти, в которой хранится список управления доступом.

Параметр pAclInformation должен указывать на область памяти, в которую функция запишет затребованную информацию.

Параметр nAclInformationLength должен содержать длину области памяти, на которую указывает параметр pAclInformation.

В параметре dwAclInformationClass должна быть установлена одна из перечислимых констант, принадлежащая перечислению типа:

```
typedef enum  _ACL_INFORMATION_CLASS {
  AclRevisionInformation = 1,    // версия ACL
  AclSizeInformation             // длина ACL
} ACL_INFORMATION_CLASS;
```

Если в параметре dwAclInformationClass установлено значение AclRevisionInformation, то функция запишет в область памяти, на которую указывает параметр pAclInformation, информацию о версии списка управления доступом в формате структуры типа:

```
typedef struct  _ACL_REVISION_INFORMATION {
  DWORD  AclRevision;      // версия ACL
}  ACL_REVISION_INFORMATION;
```

тип указателя на которую определяется также как

```
typedef  ACL_REVISION_INFORMATION *PACL_REVISION_INFORMATION;
```

Если же в параметре `dwAclInformationClass` установлено значение `AclSizeInformation`, то функция запишет в область памяти, на которую указывает параметр `pAclInformation`, информацию о размерах списка управления доступом в формате структуры типа:

```
typedef struct  _ACL_SIZE_INFORMATION {
  DWORD  AceCount;         // количество элементов в списке
  DWORD  AclBytesInUse;    // количество используемых байтов
  DWORD  AclBytesFree;     // количество свободных байтов
  }  ACL_SIZE_INFORMATION;
```

Тип указателя на которую определяется как:

```
typedef ACL_SIZE_INFORMATION *PACL_SIZE_INFORMATION;
```

В листинге 44.6 приведена программа, которая получает информацию из списка управления доступом DACL, используя для этого функцию `GetAclInformation`.

Листинг 44.6. Получение информации из списка управления доступом DACL

```
#include <windows.h>
#include <stdio.h>

int main()
{
  char chDirName[248];   // имя файла

  PSECURITY_DESCRIPTOR lpSd = NULL;  // указатель на SD

  PACL lpDacl = NULL;    // указатель на список управления доступом
  BOOL bDaclPresent;     // признак присутствия списка DACL
  BOOL bDaclDefaulted;   // признак списка DACL по умолчанию

  ACL_REVISION_INFORMATION ari;  // версия списка DACL
  ACL_SIZE_INFORMATION asi;      // размер списка DACL

  DWORD dwLength;      // длина дескриптора безопасности

  DWORD dwRetCode;     // код возврата

  // вводим имя файла
  printf("Input a directory or file name: ");
```

```
gets(chDirName);

// получаем длину дескриптора безопасности
if (!GetFileSecurity(
  chDirName,          // имя файла
  DACL_SECURITY_INFORMATION,   // получаем DACL
  lpSd,               // адрес дескриптора безопасности
  0,                  // определяем длину буфера
  &dwLength))         // адрес для требуемой длины
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
    // распределяем память для буфера
    lpSd = (SECURITY_DESCRIPTOR*) new char[dwLength];
  else
  {
    // выходим из программы
    printf("Get file security failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// распределяем память для дескриптора безопасности
lpSd = (PSECURITY_DESCRIPTOR) new char[dwLength];

// читаем дескриптор безопасности
if (!GetFileSecurity(
  chDirName,          // имя файла
  DACL_SECURITY_INFORMATION,   // получаем DACL
  lpSd,               // адрес дескриптора безопасности
  dwLength,           // длину буфера
  &dwLength))         // адрес для требуемой длины
{
  dwRetCode = GetLastError();
  printf("Get file security failed.\n");
  printf("Error code: %d\n", dwRetCode);
```

```
  return dwRetCode;
}

// получаем список DACL из дескриптора безопасности
if (!GetSecurityDescriptorDacl(
  lpSd,              // адрес дескриптора безопасности
  &bDaclPresent,     // признак присутствия списка DACL
  &lpDacl,           // адрес указателя на DACL
  &bDaclDefaulted))  // признак списка DACL по умолчанию
{
  dwRetCode = GetLastError();
  printf("Get security descriptor DACL failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// проверяем, есть ли DACL
if (!bDaclPresent)
{
  printf("Dacl is not present.");

  return 0;
}

// получаем версию списка DACL
GetAclInformation(
  lpDacl,        // адрес DACL
  &ari,          // адрес буфера
  sizeof(ari),   // размер буфера
  AclRevisionInformation);   // информация о версии

// печатаем версию списка DACL
printf("Acl Revision Information: %u\n", ari.AclRevision);

// получаем размеры списка DACL
GetAclInformation(
  lpDacl,        // адрес DACL
  &asi,          // адрес буфера
```

```
    sizeof(asi),   // размер буфера
    AclSizeInformation);   // информация о размерах DACL

  // печатаем информацию о размерах списка DACL
  printf("Ace Count: %u\n", asi.AceCount);
  printf("AclBytesInUse: %u\n", asi.AclBytesInUse);
  printf("Acl Bytes Free: %u\n", asi.AclBytesFree);

  // освобождаем память
  delete[] lpSd;

  return 0;
}
```

44.9. Установка версии списка управления доступом

Для установки версии списка управления доступом используется функция `SetAclInformation`, которая имеет следующий прототип:

```
BOOL  SetAclInformation(
  PACL    pAcl,                    // адрес списка управления доступом
  LPVOID  pAclInformation,         // адрес буфера с информацией
  DWORD   nAclInformationLength,   // длина буфера
  ACL_INFORMATION_CLASS  dwAclInformationClass   // тип информации
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`.

Параметр `pAcl` должен указывать на область памяти, в которой хранится список управления доступом.

Параметр `pAclInformation` должен указывать на область памяти, в которой хранится версия списка управления доступом.

Параметр `nAclInformationLength` должен содержать длину области памяти, на которую указывает параметр `pAclInformation`.

В параметре `dwAclInformationClass` должно быть установлено значение `AclRevisionInformation`, принадлежащее перечислению типа `ACL_INFORMATION_CLASS`. Это перечисление было рассмотрено в *разд. 44.8*.

44.10. Определение доступной памяти

Для определения адреса первого свободного байта в области памяти, хранящей список управления доступом, предназначена функция `FindFirstFreeAce`, которая имеет следующий прототип:

```
BOOL  FindFirstFreeAce(
  PACL    pAcl,    // адрес списка управления доступом
  LPVOID  *pAce    // указатель на переменную для адреса первого
                   // свободного байта
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`. Параметры функции имеют следующее назначение.

Параметр `pAcl` должен указывать на область памяти, в которой хранится список управления доступом.

Параметр `pAce` должен указывать на переменную, в которую функция запишет адрес первого свободного байта в области памяти, на которую указывает параметр `pAcl`. Если структура этой памяти не соответствует структуре списка управления доступом, то функция запишет в переменную значение `NULL`. Если память заполнена полностью, то функция запишет в переменную адрес байта, следующего за списком управления доступом.

Глава 45

Управление безопасностью объектов на низком уровне

В этой главе будут рассмотрены функции, используемые для управления безопасностью объектов конкретного типа. Прежде чем переходить к самим функциям скажем, что для этих же целей могут использоваться функции `GetSecurityInfo`, `GetNamedSecurityInfo`, `SetNamedSecurityInfo` и `SetNamedSecurityInfo`, которые позволяют работать с атрибутами безопасности объектов на более высоком уровне. Эти функции, а также другие функции, предназначенные для извлечения и установки информации в дескриптор безопасности объекта, были рассмотрены в *гл. 40*.

Кроме того, отметим, что при работе с функциями, которые получают доступ к дескриптору безопасности объекта, нужно различать абсолютный и относительный форматы дескрипторов безопасности. То есть функции чтения атрибутов безопасности объекта получают дескриптор безопасности этого объекта в относительном формате. Нельзя изменять атрибуты безопасности в дескрипторе, который получен функцией чтения атрибутов безопасности объекта, т. к. это нарушит относительный формат дескриптора безопасности. Поэтому для изменения атрибутов безопасности объекта нужно создать новый дескриптор безопасности в абсолютном формате и установить в нем новые атрибуты безопасности объекта. После этого можно изменять атрибуты безопасности объекта посредством вызова функции установки новых атрибутов безопасности. Подробнее о форматах дескриптора безопасности было рассказано в *разд. 40.1*.

В заключение нашего введения также скажем, что функции, которые используются для доступа к информации из дескриптора безопасности, рассматривались также в *гл. 40, 44*. Там же приведены другие примеры использования этих функций.

45.1. Доступ к информации о владельце объекта

Для установки в дескриптор безопасности объекта сведений о владельце объекта используется функция SetSecurityDescriptorOwner, которая имеет следующий прототип:

```
BOOL  SetSecurityDescriptorOwner(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,  // SD объекта
  PSID  pOwner,               // SID нового владельца объекта
  BOOL  bOwnerDefaulted       // признак владельца объекта по умолчанию
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError.

Параметр pSecurityDescriptor должен указывать на дескриптор безопасности, в который функция устанавливает нового владельца объекта.

Параметр pOwner должен указывать на идентификатор безопасности учетной записи, которая становится владельцем объекта. Если в этом параметре установлено значение NULL, то функция удаляет старого владельца объекта, а нового не устанавливает. То есть в этом случае объект не имеет владельца.

В параметре bOwnerDefaulted должно быть установлено одно из значений FALSE или TRUE. Если установлено значение FALSE, то функция сбрасывает в дескрипторе безопасности флаг SE_OWNER_DEFAULTED. Если же этот параметр содержит значение TRUE, то владелец объекта определяется системой, при этом в дескрипторе безопасности устанавливается флаг SE_OWNER_DEFAULTED.

Для получения из дескриптора безопасности информации о владельце объекта используется функция GetSecurityDescriptorOwner, которая имеет следующий прототип:

```
BOOL  GetSecurityDescriptorOwner(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,  // на SD объекта
  PSID    *pOwner,            // SID владельца объекта
  LPBOOL  lpbOwnerDefaulted  // признак владельца объекта по умолчанию
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError.

Параметр pSecurityDescriptor должен указывать на дескриптор безопасности, из которого функция получает информацию о владельце объекта.

Параметр pOwner должен содержать адрес указателя, в который функция запишет адрес идентификатора безопасности владельца объекта. Если объект не имеет владельца, то функция запишет по этому адресу значение NULL.

Параметр lpbOwnerDefaulted должен указывать на булеву переменную, в которую функция запишет значение флага SE_OWNER_DEFAULTED.

В листинге 45.1 приведена программа, которая изменяет владельца файла, используя для доступа к информации о владельце файла функции GetSecurityDescriptorOwner и SetSecurityDescriptorOwner.

Листинг 45.1 Изменение владельца файла

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>
#include <lm.h>

int main()
{
  wchar_t wchDirName[248];          // имя каталога
  wchar_t wchUserName[UNLEN];       // имя пользователя

  DWORD dwSdLength = 0;             // длина SD
  DWORD dwSidLength = 0;            // длина SID
  DWORD dwLengthOfDomainName = 0;   // длина имени домена

  PSID lpSid = NULL;                // указатель на SID нового владельца
  LPTSTR lpDomainName = NULL;       // указатель на имя домена

  SID_NAME_USE typeOfSid;           // тип учетной записи

  SECURITY_DESCRIPTOR sdAbsoluteSd;  // абсолютный формат дескриптора
                                     // безопасности

  DWORD dwRetCode;      // код возврата

  // читаем имя файла или каталога
```

```
printf("Input a file or directory name: ");
_getws(wchDirName);

// вводим имя пользователя, который будет новым владельцем файла
printf("Input a user name: ");
_getws(wchUserName);

// определяем длину SID пользователя
if (!LookupAccountName(
  NULL,             // ищем имя на локальном компьютере
  wchUserName,      // имя пользователя
  NULL,             // определяем длину SID
  &dwSidLength,     // длина SID
  NULL,             // определяем имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &typeOfSid))      // тип учетной записи
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
  {
    // распределяем память для SID
    lpSid = (SID*) new char[dwSidLength];
    lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];
  }
  else
  {
    // выходим из программы
    printf("Lookup account name length failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// определяем SID
if(!LookupAccountName(
  NULL,               // ищем имя на локальном компьютере
  wchUserName,        // имя пользователя
```

```
    lpSid,              // указатель на SID
    &dwSidLength,       // длина SID
    lpDomainName,       // указатель на имя домена
    &dwLengthOfDomainName,   // длина имени домена
    &typeOfSid))        // тип учетной записи
  {
    dwRetCode = GetLastError();

    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }

  // создаем дескриптор безопасности
  if (!InitializeSecurityDescriptor(
    &sdAbsoluteSd,      // адрес структуры SD
    SECURITY_DESCRIPTOR_REVISION))
  {
    dwRetCode = GetLastError();
    perror("Initialize security descriptor failed.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  // устанавливаем нового владельца в дескриптор безопасности
  if (!SetSecurityDescriptorOwner(
    &sdAbsoluteSd,      // адрес дескриптора безопасности
    lpSid,              // указатель на SID
    FALSE))             // SID не задан по умолчанию
  {
    dwRetCode = GetLastError();
    perror("Set security descriptor owner failed.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  // проверяем структуру дескриптора безопасности
```

```
  if (!IsValidSecurityDescriptor(&sdAbsoluteSd))
  {
    dwRetCode = GetLastError();
    perror("Security descriptor is invalid.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }
  // устанавливаем новый дескриптор безопасности
  if (!SetFileSecurity(
    wchDirName,          // имя файла
    OWNER_SECURITY_INFORMATION,  // устанавливаем SID
    &sdAbsoluteSd))      // адрес дескриптора безопасности
  {
    dwRetCode = GetLastError();

    if (dwRetCode == ERROR_INVALID_OWNER)
      printf("The user can not be the owner of the file.");

    printf("Set file security failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }

  // освобождаем память
  delete[] lpSid;
  delete[] lpDomainName;

  return 0;
}
```

45.2. Доступ к информации о первичной группе владельца объекта

Для установки в дескриптор безопасности объекта сведений о первичной группе владельца объекта используется функция `SetSecurityDescriptorGroup`, которая имеет следующий прототип:

```
BOOL  SetSecurityDescriptorGroup(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,  // на SD объекта
```

```
  PSID  pGroup,              // указатель на SID первичной группы
  BOOL  bGroupDefaulted      // признак группы по умолчанию
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError.

Параметр pSecurityDescriptor должен указывать на дескриптор безопасности, в который функция устанавливает новую первичную группу владельца объекта.

Параметр pGroup должен указывать на идентификатор безопасности учетной записи группы, которая становится первичной группой владельца объекта. Если в этом параметре установлено значение NULL, то функция удаляет старую первичную группу, а новую первичную группу не устанавливает. То есть в этом случае объект не имеет первичной группы владельца объекта.

В параметре bOwnerDefaulted должно быть установлено одно из значений FALSE или TRUE. Если установлено значение FALSE, то функция сбрасывает в дескрипторе безопасности флаг SE_GROUP_DEFAULTED. Если же этот параметр содержит значение TRUE, то первичная группа владельца объекта определяется системой, при этом в дескрипторе безопасности устанавливается флаг SE_GROUP_DEFAULTED.

Для получения из дескриптора безопасности информации о первичной группе владельца объекта используется функция GetSecurityDescriptorGroup, которая имеет следующий прототип:

```
BOOL  GetSecurityDescriptorGroup(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,  // SD объекта
  PSID    * pGroup,          // адрес указателя на SID первичной группы
  LPBOOL  lpbGroupDefaulted  // признак первичной группы по умолчанию
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки, в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError. Параметры функции имеют следующее назначение.

Параметр pSecurityDescriptor должен указывать на дескриптор безопасности, из которого функция получает информацию о первичной группе владельца объекта.

Параметр pOwner должен содержать адрес указателя, в который функция запишет адрес идентификатора безопасности первичной группы владельца объекта. Если объект не имеет первичной группы, то функция запишет по этому адресу значение NULL.

Параметр `lpbOwnerDefaulted` должен указывать на булеву переменную, в которую функция запишет значение флага `SE_GROUP_DEFAULTED`.

В листинге 45.2 приведена программа, которая изменяет первичную группу владельца файла, используя для этого функции `GetSecurityDescriptorOwner` и `SetSecurityDescriptorOwner`.

Листинг 45.2. Получение идентификатора безопасности первичной группы

```
#define _WIN32_WINNT 0x0500

#include <windows.h>
#include <stdio.h>
#include <sddl.h>

int main()
{
  char chDirName[248];       // имя файла

  PSECURITY_DESCRIPTOR lpSd = NULL;  // указатель на SD

  PSID lpGroup;              // указатель на SID первичной группы
  LPTSTR StringSid;          // указатель на строку SID
  BOOL bGroupDefaulted;      // признак группы по умолчанию

  DWORD dwLength;            // длина дескриптора безопасности

  DWORD dwRetCode;     // код возврата

  // вводим имя файла
  printf("Input a directory or file name: ");
  gets(chDirName);

  // получаем длину дескриптора безопасности
  if (!GetFileSecurity(
    chDirName,       // имя файла
    GROUP_SECURITY_INFORMATION,  // получаем SID первичной группы
    lpSd,            // адрес дескриптора безопасности
    0,               // определяем длину буфера
    &dwLength))      // адрес для требуемой длины
```

```
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
    // распределяем память для буфера
    lpSd = (SECURITY_DESCRIPTOR*) new char[dwLength];
  else
  {
    // выходим из программы
    printf("Get file security failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// распределяем память для дескриптора безопасности
lpSd = (PSECURITY_DESCRIPTOR) new char[dwLength];

// читаем дескриптор безопасности
if (!GetFileSecurity(
  chDirName,       // имя файла
  GROUP_SECURITY_INFORMATION,  // получаем SID первичной группы
  lpSd,            // адрес дескриптора безопасности
  dwLength,        // длина буфера
  &dwLength))      // адрес для требуемой длины
{
  dwRetCode = GetLastError();
  printf("Get file security failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// получаем SID первичной группы
if (!GetSecurityDescriptorGroup(
  lpSd,                // адрес дескриптора безопасности
  &lpGroup,            // адрес указателя на SID первичной группы
```

```
    &bGroupDefaulted))   // признак первичной группы по умолчанию
  {
    dwRetCode = GetLastError();
    printf("Get security descriptor group failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }

  // проверяем, есть ли первичная группа
  if (lpGroup == NULL)
  {
    printf("The primary group is not present.\n");

    return 0;
  }

  // преобразуем SID в строку
  if (!ConvertSidToStringSid(
    lpGroup,       // указатель на SID первичной группы
    &StringSid))   // строка с SID
  {
    dwRetCode = GetLastError();
    printf("Convert sid to string sid failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }

  printf("%s\n",StringSid);

  // освобождаем память
  LocalFree(StringSid);
  delete[] lpSd;

  return 0;
}
```

45.3. Доступ к списку DACL

Для установки в дескриптор безопасности объекта списка управления доступом DACL используется функция SetSecurityDescriptorDacl, которая имеет следующий прототип:

```
BOOL  SetSecurityDescriptorDacl(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,  // SD объекта
  BOOL    bDaclPresent,    // признак присутствия DACL
  PACL    pDacl,           // указатель на список DACL
  BOOL    bDaclDefaulted   // признак DACL, заданного по умолчанию
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError.

Параметр pSecurityDescriptor должен указывать на дескриптор безопасности, в который функция устанавливает новый список управления доступом DACL.

В параметре bDaclPresent должно быть установлено одно из значений FALSE или TRUE. Если установлено значение FALSE, то функция сбрасывает флаг безопасности SE_DACL_PRESENT в дескрипторе, и, в этом случае, параметры pDacl и bDaclDefaulted игнорируются. Если же в этом параметре задано значение TRUE, то в дескрипторе безопасности устанавливается флаг SE_DACL_PRESENT.

Параметр pDacl должен указывать на список управления доступом DACL, который устанавливается в дескриптор безопасности объекта. Если в этом параметре установлено значение NULL, то функция устанавливает значение указателя на список управления доступом DACL в дескрипторе безопасности также в NULL. Это означает отсутствие списка DACL у объекта, что разрешает полный доступ к объекту всем учетным записям.

В параметре bDaclDefaulted должно быть установлено одно из значений FALSE или TRUE. Если установлено значение FALSE, то функция сбрасывает флаг безопасности SE_DACL_DEFAULTED в дескрипторе. Если же этот параметр содержит значение TRUE, то список управления доступом объекта определяется системой, при этом в дескрипторе безопасности устанавливается флаг SE_DACL_DEFAULTED.

Для получения из дескриптора безопасности информации о списке управления доступом DACL к объекту используется функция GetSecurityDescriptorDacl, которая имеет следующий прототип:

```
BOOL  GetSecurityDescriptorDacl(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,  // SD объекта
```

```
    LPBOOL  lpbDaclPresent,    // указатель на признак присутствия DACL
    PACL    *pDacl,            // адрес указателя на DACL
    LPBOOL  lpbDaclDefaulted   // признак DACL, заданного по умолчанию
  );
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить с помощью функции GetLastError.

Параметр pSecurityDescriptor должен указывать на дескриптор безопасности, из которого функция получает информацию о списке управления доступом DACL.

Параметр lpbDaclPresent должен содержать адрес переменной типа BOOL. Если дескриптор безопасности объекта содержит список управления доступом DACL, то в эту переменную функция запишет значение TRUE. В противном случае в эту переменную будет записано значение FALSE.

Параметр pDacl должен содержать адрес указателя, в который функция запишет адрес списка DACL. Если объект не имеет списка DACL, то значение указателя не изменится.

Параметр lpbDaclDefaulted должен указывать на переменную типа BOOL, в которую функция запишет значение флага SE_DACL_DEFAULTED.

Примеры использования функций SetSecurityDescriptorDacl и GetSecurityDescriptorDacl для доступа к списку управления DACL можно посмотреть в программах, приведенных в предыдущей главе. Теперь же приведем в листинге 45.3 программу, которая создает мьютекс с атрибутами безопасности.

Листинг 45.3. Создание мьютекса с атрибутами безопасности

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>
#include <lm.h>

int main()
{
  HANDLE hMutex;          // дескриптор мьютекса

  ACL *lpDacl;            // указатель на список доступа
```

```
DWORD dwDaclLength;    // длина списка доступа

wchar_t wchUserName[UNLEN];      // имя пользователя

DWORD dwLengthOfDomainName = 0;  // длина имени домена

DWORD dwLengthOfSid = 0;         // длина SID
SID *lpSid = NULL;               // указатель на SID
LPTSTR lpDomainName = NULL;      // указатель на имя домена

SID_NAME_USE typeOfSid;    // тип учетной записи

SECURITY_DESCRIPTOR sd;    // дескриптор безопасности мьютекса
SECURITY_ATTRIBUTES sa;    // атрибуты защиты мьютекса

DWORD dwRetCode;   // код возврата

// вводим имя пользователя, которому разрешим доступ к мьютексу
printf("Input a user name: ");
_getws(wchUserName);

// определяем длину SID пользователя
if (!LookupAccountName(
  NULL,              // ищем имя на локальном компьютере
  wchUserName,       // имя пользователя
  NULL,              // определяем длину SID
  &dwLengthOfSid,    // длина SID
  NULL,              // определяем имя домена
  &dwLengthOfDomainName,  // длина имени домена
  &typeOfSid))       // тип учетной записи
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
  {
    // распределяем память для SID и имени домена
    lpSid = (SID*) new char[dwLengthOfSid];
    lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];
  }
```

```
  else
  {
    // выходим из программы
    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// определяем SID и имя домена пользователя
if(!LookupAccountName(
  NULL,              // ищем имя на локальном компьютере
  wchUserName,       // имя пользователя
  lpSid,             // указатель на SID
  &dwLengthOfSid,    // длина SID
  lpDomainName,      // указатель на имя домена
  &dwLengthOfDomainName,  // длина имени домена
  &typeOfSid))       // тип учетной записи
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// проверяем тип SID
if (typeOfSid != SidTypeUser)
{
  printf("This is not a user name.\n");
  return 1;
}

// определем длину списка DACL
dwDaclLength = sizeof(ACL)
  + sizeof(ACCESS_ALLOWED_ACE) - sizeof(DWORD) + dwLengthOfSid
  + sizeof(ACCESS_DENIED_ACE) - sizeof(DWORD) + dwLengthOfSid;
```

```
// распределяем память под DACL
lpDacl = (ACL*)new char[dwDaclLength];

// инициализируем список DACL
if (!InitializeAcl(
  lpDacl,          // адрес DACL
  dwDaclLength,    // длина DACL
  ACL_REVISION))   // версия DACL
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// добавляем запрещающий элемент в список DACL
if (!AddAccessDeniedAce(
  lpDacl,          // адрес DACL
  ACL_REVISION,    // версия DACL
  WRITE_OWNER,     // запрещаем изменять владельца объекта
  lpSid))          // адрес SID
{
  dwRetCode = GetLastError();
  perror("Add access denied ace failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// добавляем разрешающий элемент в список DACL
if (!AddAccessAllowedAce(
  lpDacl,        // адрес DACL
  ACL_REVISION,  // версия DACL
  GENERIC_ALL,   // разрешаем все родовые права доступа
  lpSid))        // адрес SID
{
  dwRetCode = GetLastError();
```

```
  perror("Add access allowed ace failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// инициализируем версию дескриптора безопасности
if (!InitializeSecurityDescriptor(
  &sd,
  SECURITY_DESCRIPTOR_REVISION))
{
  dwRetCode = GetLastError();
  printf("Initialize security descroptor failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// устанавливаем SID владельца объекта
if (!SetSecurityDescriptorOwner(
  &sd,          // адрес дескриптора безопасности
  NULL,         // не задаем владельца
  SE_OWNER_DEFAULTED))    // определить владельца по умолчанию
{
  dwRetCode = GetLastError();
  perror("Set security descriptor owner failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// устанавливаем SID первичной группы владельца
if (!SetSecurityDescriptorGroup(
  &sd,          // адрес дескриптора безопасности
  NULL,         // не задаем первичную группу
  SE_GROUP_DEFAULTED))    // определить первичную группу по умолчанию
{
  dwRetCode = GetLastError();
  perror("Set security descriptor group failed.\n");
```

```
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}
// устанавливаем DACL в дескриптор безопасности
if (!SetSecurityDescriptorDacl(
  &sd,        // адрес дескриптора безопасности
  TRUE,       // DACL присутствует
  lpDacl,     // указатель на DACL
  FALSE))     // DACL не задан по умолчанию
{
  dwRetCode = GetLastError();
  perror("Set security descriptor group failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// проверяем структуру дескриптора безопасности
if (!IsValidSecurityDescriptor(&sd))
{
  dwRetCode = GetLastError();
  perror("Security descriptor is invalid.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// инициализируем атрибуты безопасности
sa.nLength = sizeof(sa);       // устанавливаем длину атрибутов защиты
sa.lpSecurityDescriptor = &sd;   // устанавливаем адрес SD
sa.bInheritHandle = FALSE;     // дескриптор мьютекса ненаследуемый


// создаем мьютекс
hMutex = CreateMutex(&sa, FALSE, L"DemoMutex");
if (hMutex == NULL)
{
  dwRetCode = GetLastError();
```

```
    perror("Create mutex failed.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }

  printf("The mutex is created.\n");

  // освобождаем память
  delete[] lpSid;
  delete[] lpDomainName;
  delete[] lpDacl;

  CloseHandle(hMutex);

  return 0;
}
```

45.4. Доступ к списку SACL

Для установки в дескриптор безопасности объекта списка управления доступом SACL используется функция `SetSecurityDescriptorSacl`, которая имеет следующий прототип:

```
BOOL  SetSecurityDescriptorSacl(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,   // SD объекта
  BOOL  bSaclPresent,    // признак присутствия SACL
  PACL  pSacl,           // указатель на список SACL
  BOOL  bSaclDefaulted   // признак SACL, заданного по умолчанию
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`.

Параметр `pSecurityDescriptor` должен указывать на дескриптор безопасности, в который функция устанавливает новый список управления доступом SACL.

В параметре `bSaclPresent` должно быть установлено одно из значений `FALSE` или `TRUE`. Если установлено значение `FALSE`, то функция сбрасывает в дескрипторе безопасности флаг `SE_SACL_PRESENT` и, в этом случае, параметры `pSacl` и `bSaclDefaulted` игнорируются. Если же в этом параметре задано значение `TRUE`, то в дескрипторе безопасности устанавливается флаг `SE_SACL_PRESENT`.

Параметр `pSacl` должен указывать на список управления доступом SACL, который устанавливается в дескриптор безопасности объекта. Если в этом параметре установлено значение `NULL`, то функция устанавливает значение указателя на список управления доступом SACL в дескрипторе безопасности также в `NULL`. Это означает отсутствие списка SACL у объекта. В этом случае аудит доступа субъектов к объекту проводиться не будет.

В параметре `bSaclDefaulted` должно быть установлено одно из значений `FALSE` или `TRUE`. Если установлено значение `FALSE`, то функция сбрасывает в дескрипторе безопасности флаг `SE_SACL_DEFAULTED`. Если же этот параметр содержит значение `TRUE`, то список управления доступом объекта определяется системой, при этом в дескрипторе безопасности устанавливается флаг `SE_SACL_DEFAULTED`.

Для получения из дескриптора безопасности информации о списке управления доступом SACL к объекту используется функция `GetSecurityDescriptorSacl`, которая имеет следующий прототип:

```
BOOL  GetSecurityDescriptorDacl(
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,   // SD объекта
  LPBOOL  lpbSaclPresent,    // признак присутствия SACL
  PACL    *pSacl,            // адрес указателя на SACL
  LPBOOL  lpbSaclDefaulted   // признак SACL, заданного по умолчанию
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`.

Параметр `pSecurityDescriptor` должен указывать на дескриптор безопасности, из которого функция получает информацию о списке управления доступом SACL.

Параметр `lpbSaclPresent` должен содержать адрес переменной типа `BOOL`. Если дескриптор безопасности объекта содержит список управления доступом SACL, то в эту переменную функция запишет значение `TRUE`. В противном случае в эту переменную будет записано значение `FALSE`.

Параметр `pSacl` должен содержать адрес указателя, в который функция запишет адрес списка SACL. Если объект не имеет списка SACL, то значение указателя не изменится.

Параметр `lpbSaclDefaulted` должен указывать на переменную типа `BOOL`, в которую функция запишет значение флага `SE_SACL_DEFAULTED`.

Примеры использования функций `SetSecurityDescriptorSacl` и `GetSecurityDescriptorSacl` для доступа к списку управления DACL можно посмотреть в программах, приведенных в *гл. 44*.

45.5. Защита файлов и каталогов

Прежде чем перейти к описанию функций, используемых для доступа к атрибутам безопасности файла или каталога на низком уровне, перечислим специфические права доступа, которые могут быть установлены в списках управления доступом к файлам и каталогам. Соответственно, такие же режимы доступа пользователь может указать при попытке открытия доступа к файлу или каталогу.

Для файлов могут быть установлены следующие специфические режимы доступа, которые соответствуют правам субъектов на доступ к файлам:

- `FILE_APPEND_DATA` — добавление данных;
- `FILE_EXECUTE` — исполнение.

Следующие специфические права доступа могут быть установлены как для файлов, так и для каналов:

- `FILE_READ_DATA` — чтение данных;
- `FILE_WRITE_DATA` — запись данных.

Для файлов и каталогов могут быть установлены следующие права доступа:

- `FILE_READ_EA` — чтение расширенных атрибутов;
- `FILE_WRITE_EA` — право записывать расширенные атрибуты.

Для каталогов могут быть установлены следующие права доступа:

- `FILE_LIST_DIRECTORY` — перечисление содержимого каталога;
- `FILE_ADD_FILE` — добавление файлов в каталог;
- `FILE_ADD_SUBDIRECTORY` — добавление подкаталогов;
- `FILE_TRAVERSE` — следование через каталог;
- `FILE_DELETE_CHILD` — удаление дочерних каталогов.

Следующие права доступа определены для всех объектов типа файл:

- `FILE_READ_ATTRIBUTES` — чтение атрибутов;
- `FILE_WRITE_ATTRIBUTES` — запись атрибутов.

Кроме того, для краткого обозначения полного доступа к объектам типа файл определена следующая символьная константа:

- `FILE_ALL_ACCESS` — полный доступ к файлу.

Родовые права доступа к файлам отображаются в следующие специфические и стандартные права доступа:

- `FILE_GENERIC_READ` — включает права `STANDARD_RIGHTS_READ`, `FILE_READ_DATA`, `FILE_READ_ATTRIBUTES`, `FILE_READ_EA` и `SYNCHRONIZE`;

- `FILE_GENERIC_WRITE` — включает права `STANDARD_RIGHTS_WRITE`, `FILE_WRITE_DATA`, `FILE_WRITE_ATTRIBUTES`, `FILE_WRITE_EA`, `FILE_APPEND_DATA` и `SYNCHRONIZE`;
- `FILE_GENERIC_EXECUTE` — включает права `STANDARD_RIGHTS_EXECUTE`, `FILE_READ_ATTRIBUTES`, `FILE_EXECUTE` и `SYNCHRONIZE`.

Теперь перейдем к функциям, используемым для доступа к атрибутам безопасности файлов и каталогов на низком уровне.

Для изменения атрибутов безопасности файла или каталога используется функция `SetFileSecurity`, которая имеет следующий прототип:

```
BOOL  SetFileSecurity(
  LPCSTR  lpFileName,        // имя файла
  SECURITY_INFORMATION  SecurityInformation,   // управляющие флаги
  PSECURITY_DESCRIPTOR  pSecurityDescriptor    // указатель на SD
);
```

Отметим, что если функция изменяет атрибуты безопасности каталога, то эти изменения не распространяются на дочерние объекты каталога. В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`.

Параметр `lpFileName` должен указывать на имя файла или каталога, в котором изменяются атрибуты безопасности.

Параметр `SecurityInformation` имеет тип `SECURITY_INFORMATION`, который является синонимом типа `DWORD`. В этом параметре должны быть установлены флаги, которые отмечают, какие атрибуты безопасности файла или каталога должны быть изменены. Для установки флагов используются следующие символические константы:

- `OWNER_SECURITY_INFORMATION` — владелец объекта;
- `GROUP_SECURITY_INFORMATION` — первичная группа владельца объекта;
- `DACL_SECURITY_INFORMATION` — список управления доступом DACL;
- `SACL_SECURITY_INFORMATION` — список управления доступом SACL.

Операционная система Windows 2000 и более поздние версии также поддерживают следующие флаги:

- `PROTECTED_DACL_SECURITY_INFORMATION` — DACL не наследует элементы;
- `PROTECTED_SACL_SECURITY_INFORMATION` — SACL не наследует элементы;
- `UNPROTECTED_DACL_SECURITY_INFORMATION` — DACL наследует элементы;
- `UNPROTECTED_SACL_SECURITY_INFORMATION` — SACL наследует элементы.

Параметр `pSecurityDescriptor` должен указывать на дескриптор безопасности, в котором установлены новые атрибуты безопасности файла или каталога.

Примечание

Изменять атрибуты безопасности файла или каталога при помощи функции `SetFileSecurity` может только владелец этого файла или каталога или пользователь, который имеет соответствующие стандартные права доступа. Кроме того, для изменения содержимого списка управления доступом SACL пользователь должен обладать привилегией `SE_SECURITY_NAME`, которая находится в разрешенном состоянии.

Для получения атрибутов безопасности файла или каталога используется функция `GetFileSecurity`, которая имеет следующий прототип:

```
BOOL  GetFileSecurity(
  LPCSTR   lpFileName,          // имя файла
  SECURITY_INFORMATION  RequestedInformation,  // управляющие флаги
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,   // указатель на SD
  DWORD    nLength,             // длина буфера для SD
  LPDWORD  lpnLengthNeeded      // требуемая длина буфера
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`.

Параметр `lpFileName` должен указывать на имя файла или каталога, атрибуты безопасности которого получает функция.

Параметр `SecurityInformation` имеет тип `SECURITY_INFORMATION`, который является синонимом типа `DWORD`. В этом параметре должны быть установлены флаги, которые отмечают, какие атрибуты безопасности файла или каталога должны быть прочитаны функцией. Для установки флагов используются те же символические константы, что и в соответствующем параметре функции `SetFileSecurity`.

Параметр `pSecurityDescriptor` должен указывать на буфер, в который функция запишет дескриптор безопасности файла. Дескриптор безопасности будет записан в относительном формате.

В параметре `nLength` должна быть задана длина буфера, на который указывает параметр `pSecurityDescriptor`.

Параметр `lpnLengthNeeded` должен указывать на переменную типа `DWORD`, в которую функция в случае успешного завершения запишет 0, а в случае неудачи требуемую длину буфера.

Примечание

Читать при помощи функции `GetFileSecurity` атрибуты безопасности файла или каталога может только владелец этого файла (каталога) или пользователь,

для которого в список DACL этого файла (каталога) включен разрешающий элемент с правом READ_CONTROL. Кроме того, для чтения содержимого списка SACL пользователь должен обладать привилегией SE_SECURITY_NAME, которая находится в разрешенном состоянии.

Листинг 45.4. Добавление элементов в список DACL файла или каталога

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>
#include <lm.h>

int main()
{
  wchar_t wchDirName[248];       // имя каталога
  wchar_t wchUserName[UNLEN];    // имя пользователя

  ACL *lpOldDacl;    // указатель на старый DACL
  ACL *lpNewDacl;    // указатель на новый DACL
  LPVOID lpAce;      // указатель на элемент ACE

  DWORD dwDaclLength = 0;        // длина DACL
  DWORD dwSdLength = 0;          // длина SD
  DWORD dwSidLength = 0;         // длина SID
  DWORD dwLengthOfDomainName = 0;    // длина имени домена

  PSID lpSid = NULL;             // указатель на разрешающий SID
  LPTSTR lpDomainName = NULL;    // указатель на имя домена

  SID_NAME_USE typeOfSid;        // тип учетной записи

  SECURITY_DESCRIPTOR *lpSd = NULL;  // адрес дескриптора безопасности
  SECURITY_DESCRIPTOR sdAbsoluteSd;  // абсолютный формат SD
  BOOL bDaclPresent;             // признак присутствия списка DACL
  BOOL bDaclDefaulted;           // признак списка DACL по умолчанию

  DWORD dwRetCode;   // код возврата
```

```
// читаем имя файла или каталога
printf("Input a file or directory name: ");
_getws(wchDirName);

// получаем длину дескриптора безопасности
if (!GetFileSecurity(
  wchDirName,    // имя файла
  DACL_SECURITY_INFORMATION,   // получаем DACL
  lpSd,          // адрес дескриптора безопасности
  0,             // определяем длину буфера
  &dwSdLength))  // адрес для требуемой длины
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
    // распределяем память для буфера
    lpSd = (SECURITY_DESCRIPTOR*) new char[dwSdLength];
  else
  {
    // выходим из программы
    printf("Get file security failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// читаем дескриптор безопасности
if (!GetFileSecurity(
  wchDirName,    // имя файла
  DACL_SECURITY_INFORMATION,   // получаем DACL
  lpSd,          // адрес дескриптора безопасности
  dwSdLength,    // длина буфера
  &dwSdLength))  // адрес для требуемой длины
{
  dwRetCode = GetLastError();
  printf("Get file security failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
```

```
}

// вводим имя пользователя, которого добавляем в DACL
printf("Input a user name: ");
_getws(wchUserName);

// определяем длину SID пользователя
if (!LookupAccountName(
  NULL,          // ищем имя на локальном компьютере
  wchUserName,   // имя пользователя
  NULL,          // определяем длину SID
  &dwSidLength,  // длина SID
  NULL,          // определяем имя домена
  &dwLengthOfDomainName,     // длина имени домена
  &typeOfSid))   // тип учетной записи
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
  {
    // распределяем память для SID
    lpSid = (SID*) new char[dwSidLength];
    lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];
  }
  else
  {
    // выходим из программы
    printf("Lookup account name failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// определяем SID
if(!LookupAccountName(
  NULL,          // ищем имя на локальном компьютере
  wchUserName,   // имя пользователя
  lpSid,         // указатель на SID
```

```
  &dwSidLength,  // длина SID
  lpDomainName,  // указатель на имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &typeOfSid))   // тип учетной записи
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// получаем список DACL из дескриптора безопасности
if (!GetSecurityDescriptorDacl(
  lpSd,              // адрес дескриптора безопасности
  &bDaclPresent,     // признак присутствия списка DACL
  &lpOldDacl,        // адрес указателя на DACL
  &bDaclDefaulted))  // признак списка DACL по умолчанию
{
  dwRetCode = GetLastError();
  printf("Get security descriptor DACL failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// определяем длину нового DACL
dwDaclLength = lpOldDacl->AclSize +
  sizeof(ACCESS_ALLOWED_ACE) - sizeof(DWORD) + dwSidLength;

// распределяем память под новый DACL
lpNewDacl = (ACL*)new char[dwDaclLength];

// инициализируем новый DACL
if (!InitializeAcl(
  lpNewDacl,       // адрес DACL
  dwDaclLength,    // длина DACL
  ACL_REVISION))   // версия DACL
```

```
{
  dwRetCode = GetLastError();

  printf("Lookup account name failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// добавляем новый элемент в новый DACL
if (!AddAccessDeniedAce(
  lpNewDacl,       // адрес DACL
  ACL_REVISION,    // версия DACL
  FILE_WRITE_ATTRIBUTES,  // запрещаем писать атрибуты
  lpSid))          // адрес SID
{
  dwRetCode = GetLastError();
  perror("Add access allowed ace failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// получаем адрес первого ACE в старом списке DACL
if (!GetAce(
  lpOldDacl,     // адрес старого DACL
  0,             // ищем первый элемент
  &lpAce))       // адрес первого элемента
{
  dwRetCode = GetLastError();

  printf("Get ace failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// переписываем элементы из старого DACL в новый DACL
if (bDaclPresent)
```

```
{
  if (!AddAce(
    lpNewDacl,       // адрес нового DACL
    ACL_REVISION,    // версия DACL
    MAXDWORD,        // добавляем в конец списка
    lpAce,           // адрес старого DACL
    lpOldDacl->AclSize - sizeof(ACL)))  // длина старого DACL
  {
    dwRetCode = GetLastError();
    perror("Add access allowed ace failed.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }
}

// проверяем достоверность DACL
if (!IsValidAcl(lpNewDacl))
{
  dwRetCode = GetLastError();
  perror("The new ACL is invalid.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// создаем новый дескриптор безопасности в абсолютной форме
if (!InitializeSecurityDescriptor(
  &sdAbsoluteSd,       // адрес структуры SD
  SECURITY_DESCRIPTOR_REVISION))
{
  dwRetCode = GetLastError();
  perror("Initialize security descriptor failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// устанавливаем DACL в новый дескриптор безопасности
```

```
if (!SetSecurityDescriptorDacl(
  &sdAbsoluteSd,   // адрес дескриптора безопасности
  TRUE,            // DACL присутствует
  lpNewDacl,       // указатель на DACL
  FALSE))          // DACL не задан по умолчанию
{
  dwRetCode = GetLastError();
  perror("Set security descriptor DACL failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// проверяем структуру дескриптора безопасности
if (!IsValidSecurityDescriptor(&sdAbsoluteSd))
{
  dwRetCode = GetLastError();
  perror("Security descriptor is invalid.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}
// устанавливаем новый дескриптор безопасности
if (!SetFileSecurity(
  wchDirName,        // имя файла
  DACL_SECURITY_INFORMATION,     // устанавливаем DACL
  &sdAbsoluteSd))    // адрес дескриптора безопасности
{
  dwRetCode = GetLastError();
  printf("Set file security failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// освобождаем память
delete[] lpSd;
delete[] lpSid;
```

```
  delete[] lpDomainName;
  delete[] lpNewDacl;

  return 0;
}
```

45.6. Защита объектов ядра

Напомним, что объектами ядра операционной системы называются те объекты, доступ к которым осуществляется через дескриптор типа `HANDLE`. В этом разделе будут рассматриваться только те объекты ядра, которые существуют во время исполнения программы. К таким объектам ядра операционной системы относятся, например, процессы, потоки, объекты синхронизации, но не файлы. К объектам ядра также относится и маркер доступа. Для доступа к атрибутам безопасности таких объектов ядра операционной системы могут использоваться специальные функции `SetKernelObjectSecurity` и `GetKernelObjectSecurity` (см. листинг 45.5). Для доступа к объектам ядра операционной системы обычно используются стандартные режимы доступа. Но такие объекты ядра операционной системы, как процессы и потоки, имеют большое количество специфических режимов доступа, которые и приведены ниже.

Для процессов можно установить следующие специфические режимы доступа, которые соответствуют правам субъекта на доступ к процессу:

- `PROCESS_TERMINATE` — прекращение процесса;
- `PROCESS_CREATE_THREAD` — создание потока;
- `PROCESS_SET_SESSIONID` — установка идентификатора сессии;
- `PROCESS_VM_OPERATION` — работа с виртуальными адресами процесса;
- `PROCESS_VM_READ` — чтение из виртуальной памяти процесса;
- `PROCESS_VM_WRITE` — запись в виртуальную память процесса;
- `PROCESS_DUP_HANDLE` — дублирование дескриптора процесса;
- `PROCESS_CREATE_PROCESS` — создание процесса;
- `PROCESS_SET_QUOTA` — квотирование физической памяти процесса;
- `PROCESS_SET_INFORMATION` — установка управляющей информации;
- `PROCESS_QUERY_INFORMATION` — чтение управляющей информации;
- `PROCESS_ALL_ACCESS` — вполный доступ к процессу.

Для потоков существуют следующие специфические режимы доступа:

- `THREAD_TERMINATE` — завершение потока;
- `THREAD_SUSPEND_RESUME` — подвешивание и возобновление потока;

- THREAD_GET_CONTEXT — чтение контекста потока;
- THREAD_SET_CONTEXT — установка контекста потока;
- THREAD_SET_INFORMATION — установка управляющей информации;
- THREAD_QUERY_INFORMATION — чтение управляющей информации;
- THREAD_SET_THREAD_TOKEN — установка замещающего маркера доступа;
- THREAD_IMPERSONATE — разрешение использовать атрибуты защиты потока;
- THREAD_DIRECT_IMPERSONATION — разрешение потоку сервера замещать поток клиента.

Кроме того, специфические режимы доступа существуют также и для объектов синхронизации. Эти права доступа перечислены далее.

Для мьютексов можно установить следующие специфические режимы доступа:

- MUTEX_MODIFY_STATE — освобождение мьютекса;
- MUTEX_ALL_ACCESS — полный доступ к мьютексу.

Для событий можно устанавливать следующие специфические режимы доступа:

- EVENT_MODIFY_STATE — изменение состояния события;
- EVENT_ALL_ACCESS — полный доступ к событию.

Для семафоров можно устанавливать следующие специфические режимы доступа:

- SEMAPHORE_MODIFY_STATE — изменение состояния семафора;
- SEMAPHORE_ALL_ACCESS — полный доступ к семафору.

Для ожидающих таймеров можно устанавливать следующие специфические режимы доступа:

- TIMER_QUERY_STATE — запрос состояния;
- TIMER_MODIFY_STATE — изменение состояния;
- TIMER_ALL_ACCESS — полный доступ.

Для маркеров доступа определены следующие специфические режимы доступа:

- TOKEN_ASSIGN_PRIMARY — право присоединить первичный маркер доступа к процессу;
- TOKEN_DUPLICATE — право дублировать маркер доступа;
- TOKEN_IMPERSONATE — право замещать маркер доступа процесса;
- TOKEN_QUERY — право получать информацию из маркера доступа;
- TOKEN_QUERY_SOURCE — право получить информацию об источнике маркера доступа;

- TOKEN_ADJUST_PRIVILEGES — право настраивать привилегии маркера доступа;
- TOKEN_ADJUST_GROUPS — право настраивать свойства групп в маркере доступа;
- TOKEN_ADJUST_DEFAULT — право изменять информацию для установки в дескриптор объекта по умолчанию;
- TOKEN_ADJUST_SESSIONID — право настраивать идентификатор сессии в маркере доступа.

Кроме того, для маркеров доступа можно также использовать следующие символические константы, используемые для обозначения комбинации нескольких различных флагов доступа:

- TOKEN_ALL_ACCESS — включает все права доступа к маркеру доступа;
- TOKEN_READ — включает права доступа STANDARD_RIGHTS_READ и TOKEN_QUERY;
- TOKEN_WRITE — включает права STANDARD_RIGHTS_WRITE, TOKEN_ADJUST_PRIVILEGES, TOKEN_ADJUST_GROUPS и TOKEN_ADJUST_DEFAULT;
- TOKEN_EXECUTE — включат право STANDARD_RIGHTS_EXECUTE.

Теперь перейдем к функциям, предназначенным для доступа к атрибутам безопасности объектов ядра операционной системы. Для установки атрибутов безопасности объекта ядра операционной системы используется функция SetKernelObjectSecurity, которая имеет следующий прототип:

```
BOOL  SetKernelObjectSecurity(
  HANDLE  Handle,          // дескриптор объекта ядра
  SECURITY_INFORMATION  SecurityInformation, // управляющие флаги
  PSECURITY_DESCRIPTOR  pSecurityDescriptor  // дескриптор безопасности
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError.

Параметр Handle должен содержать дескриптор объекта ядра операционной системы, для которого устанавливаются атрибуты безопасности.

Параметр SecurityInformation имеет тип SECURITY_INFORMATION, который является синонимом типа DWORD. В этом параметре должны быть установлены флаги, которые отмечают, какие атрибуты безопасности объекта ядра операционной системы должны быть изменены. Для установки флагов используются следующие символические константы:

- OWNER_SECURITY_INFORMATION — владелец объекта;
- GROUP_SECURITY_INFORMATION — первичная группа владельца объекта;

- `DACL_SECURITY_INFORMATION` — список управления доступом DACL;
- `SACL_SECURITY_INFORMATION` — список управления доступом SACL.

Параметр `pSecurityDescriptor` должен указывать на дескриптор безопасности, в котором установлены новые атрибуты безопасности объекта ядра операционной системы.

> **Примечание**
>
> Изменять атрибуты безопасности объекта ядра операционной системы при помощи функции `SetKernelObjectSecurity` может только владелец этого объекта или пользователь, который имеет соответствующие стандартные права доступа. Кроме того, для изменения содержимого списка управления доступом SACL пользователь должен обладать привилегией `SE_SECURITY_NAME`, которая находится в разрешенном состоянии.

Для получения атрибутов безопасности объекта ядра операционной системы используется функция `GetKernelObjectSecurity`, которая имеет следующий прототип:

```
BOOL  GetKernelObjectSecurity(
  HANDLE   Handle,           // дескриптор объекта ядра
  SECURITY_INFORMATION  SecurityInformation, // управляющие флаги
  PSECURITY_DESCRIPTOR  pSecurityDescriptor  // дескриптор безопасности
  DWORD    nLength,          // длина буфера для SD
  LPDWORD  lpnLengthNeeded   // требуемая длина буфера
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции `GetLastError`.

Параметр `Handle` должен содержать дескриптор объекта ядра операционной системы, атрибуты безопасности которого получает функция.

Параметр `SecurityInformation` имеет тип `SECURITY_INFORMATION`, который является синонимом типа `DWORD`. В этом параметре должны быть установлены флаги, которые отмечают, какие атрибуты безопасности объекта ядра операционной системы должны быть прочитаны функцией. Для установки флагов используются те же символические константы, что и в соответствующем параметре функции `SetKernelObjectSecurity`.

Параметр `pSecurityDescriptor` должен указывать на буфер, в который функция запишет дескриптор безопасности объекта ядра операционной системы. Дескриптор безопасности будет записан в относительном формате.

В параметре `nLength` должна быть задана длина буфера, на который указывает параметр `pSecurityDescriptor`.

Параметр lpnLengthNeeded должен указывать на переменную типа DWORD, в которую функция в случае успешного завершения запишет 0, а в случае неудачи требуемую длину буфера.

Примечание

Получать атрибуты безопасности объекта ядра операционной системы при помощи функции GetKernelObjectSecurity может только владелец этого объекта или пользователь, для которого в список DACL этого объекта включен разрешающий элемент с правом READ_CONTROL. Кроме того, для чтения содержимого списка SACL пользователь должен обладать привилегией SE_SECURITY_NAME, которая находится в разрешенном состоянии.

Листинг 45.5. Получение владельца мьютекса

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>

int main()
{
  HANDLE hMutex;      // дескриптор мьютекса

  DWORD dwLengthOfSd = 0;     // длина SD
  SECURITY_DESCRIPTOR *lpSd = NULL;  // дескриптор безопасности мьютекса

  PSID lpOwner;               // указатель на SID владельца
  BOOL bOwnerDefaulted = FALSE;       // признак владельца по умолчанию

  DWORD  dwLengthOfUserName = 0;      // длина имени учетной записи
  DWORD  dwLengthOfDomainName = 0;    // длина имени домена

  LPTSTR  lpUserName = NULL;          // указатель на имя домена
  LPTSTR  lpDomainName = NULL;        // указатель на имя домена

  SID_NAME_USE type_of_SID;           // тип учетной записи

  DWORD dwRetCode;     // код возврата
```

```
// создаем каталог
hMutex = CreateMutex(NULL, FALSE, L"DemoMutex");
if (hMutex == NULL)
{
  dwRetCode = GetLastError();
  perror("Create mutex failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

printf("The mutex is created.\n");

// определяем требуемую длину буфера
if (!GetKernelObjectSecurity(
  hMutex,          // дескриптор мьютекса
  OWNER_SECURITY_INFORMATION,  // получаем владельца мьютекса
  lpSd,            // адрес SD
  0,               // определяем длину буфера
  &dwLengthOfSd))  // требуемая длина буфера
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
    // распределяем память для буфера
    lpSd = (SECURITY_DESCRIPTOR*) new char[dwLengthOfSd];
  else
  {
    // выходим из программы
    printf("Get kernel object security failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// чтаем идентификатор безопасности мьютекса
if (!GetKernelObjectSecurity(
  hMutex,          // дескриптор мьютекса
```

```
  OWNER_SECURITY_INFORMATION,  // получаем владельца мьютекса
  lpSd,            // адрес SD
  dwLengthOfSd,    // длина буфера
  &dwLengthOfSd))  // требуемая длина буфера
{
  dwRetCode = GetLastError();
  printf("Get kernel object security failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// получаем SID владельца мьютекса
if (!GetSecurityDescriptorOwner(
  lpSd,                // адрес дескриптора безопасности
  &lpOwner,            // адрес указателя на SID владельца
  &bOwnerDefaulted))   // признак владельца по умолчанию
{
  dwRetCode = GetLastError();
  printf("Get security descriptor owner failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// проверяем, есть ли владелец
if (lpOwner == NULL)
{
  printf("The owner is not present.\n");

  return 0;
}

if (bOwnerDefaulted)
  printf("The owner is defaulted.\n");

// определяем длину имени домена
if(!LookupAccountSid(
  NULL,                // ищем на локальном компьютере
```

```
  lpOwner,              // указатель на SID
  lpUserName,           // имя пользователя
  &dwLengthOfUserName,      // длина имени пользователя
  lpDomainName,         // определяем имя домена
  &dwLengthOfDomainName,    // длина имени домена
  &type_of_SID))        // тип учетной записи
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
  {
    // распределяем память под имя пользователя и имя домена
    lpUserName = (LPTSTR) new wchar_t[dwLengthOfUserName];
    lpDomainName = (LPTSTR) new wchar_t[dwLengthOfDomainName];
  }
  else
  {
    printf("Lookup account SID for length failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// определяем имя учетной записи по SID
if(!LookupAccountSid(
  NULL,                 // ищем на локальном компьютере
  lpOwner,              // указатель на SID пользователя
  lpUserName,           // имя пользователя
  &dwLengthOfUserName,      // длина имени пользователя
  lpDomainName,         // определяем имя домена
  &dwLengthOfDomainName,    // длина имени домена
  &type_of_SID))        // тип учетной записи
{
  dwRetCode = GetLastError();

  printf("Lookup account SID failed.\n");
  printf("Error code: %d\n", dwRetCode);
```

```
    return dwRetCode;
  }

  wprintf(L"User name: %s\n", lpUserName);
  wprintf(L"Domain name: %s\n", lpDomainName);

  // освобождаем память
  delete[] lpUserName;
  delete[] lpDomainName;
  delete[] lpSd;

  CloseHandle(hMutex);
  printf("The mutex is deleted.\n");

  return 0;
}
```

45.7. Защита сервисов

Прежде чем перейти к функциям, предназначенным для доступа к атрибутам безопасности сервисов, перечислим специфические права доступа, которые могут контролироваться системой безопасности при доступе субъекта к сервису.

Ниже приведены символические константы, используемые для задания специфических прав доступа к сервису:

- `SERVICE_QUERY_CONFIG` — чтение конфигурации;
- `SERVICE_CHANGE_CONFIG` — изменение конфигурации;
- `SERVICE_QUERY_STATUS` — чтение состояния;
- `SERVICE_ENUMERATE_DEPENDENTS` — перечисление зависимостей;
- `SERVICE_START` — запуск сервиса;
- `SERVICE_STOP` — остановка сервиса;
- `SERVICE_PAUSE_CONTINUE` — приостановка и возобновление сервиса;
- `SERVICE_INTERROGATE` — получать текущее состояние;
- `SERVICE_USER_DEFINED_CONTROL` — определять управление.

Дополнительно для задания режима доступа к сервисам определена символическая константа `SERVICE_ALL_ACCESS` — полный доступ к сервису.

Теперь перейдем к описанию функций, используемых для доступа к атрибутам безопасности сервиса. Для установки атрибутов безопасности сервиса

используется функция SetServiceObjectSecurity, которая имеет следующий прототип:

```
BOOL  SetServiceObjectSecurity(
  SC_HANDLE    hService,       // дескриптор сервиса
  SECURITY_INFORMATION  dwSecurityInformation,   // управляющие флаги
  PSECURITY_DESCRIPTOR  lpSecurityDescriptor     // SD
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError. Менеджер сервисов может установить следующие коды ошибок при неудачном завершении функции SetServiceObjectSecurity:

- ERROR_ACCESS_DENIED — отказано в доступе;
- ERROR_INVALID_HANDLE — недействительный дескриптор сервиса;
- ERROR_INVALID_PARAMETER — неправильные параметры;
- ERROR_SERVICE_MARKED_FOR_DELETE — сервис отмечен для удаления.

Параметры функции SetServiceObjectSecurity имеют следующее назначение.

Параметр hService должен содержать дескриптор сервиса, для которого устанавливаются атрибуты безопасности. Этот дескриптор должен быть получен посредством функций CreateService или OpenService.

Параметр dwSecurityInformation имеет тип SECURITY_INFORMATION, который является синонимом типа DWORD. В этом параметре должны быть установлены флаги, которые отмечают, какие атрибуты безопасности сервиса должны быть изменены. Для установки флагов используются следующие символические константы:

- OWNER_SECURITY_INFORMATION — владелец объекта;
- GROUP_SECURITY_INFORMATION — первичная группа владельца объекта;
- DACL_SECURITY_INFORMATION — список управления доступом DACL;
- SACL_SECURITY_INFORMATION — список управления доступом SACL.

Параметр lpSecurityDescriptor должен указывать на дескриптор безопасности, в котором установлены новые атрибуты безопасности сервиса.

Примечание

Изменять атрибуты безопасности сервиса может только владелец этого сервиса или пользователь, который имеет соответствующие стандартные права доступа. Кроме того, для изменения содержимого списка управления доступом SACL пользователь должен обладать привилегией SE_SECURITY_NAME, которая включена.

Для получения атрибутов безопасности сервиса используется функция QueryServiceObjectSecurity, которая имеет следующий прототип:

```
BOOL  QueryServiceObjectSecurity(
  SC_HANDLE  hService,          // дескриптор сервиса
  SECURITY_INFORMATION  dwSecurityInformation,   // управляющие флаги
  PSECURITY_DESCRIPTOR  lpSecurityDescriptor,    // SD
  DWORD      cbBufSize,         // длина буфера
  LPDWORD    pcbBytesNeeded     // требуемая длина буфера
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. Код ошибки в случае неудачного выполнения функции можно получить посредством вызова функции GetLastError. Менеджер сервисов может установить следующие коды ошибок при неудачном завершении функции QueryServiceObjectSecurity:

- ERROR_ACCESS_DENIED — отказано в доступе;
- ERROR_INVALID_HANDLE — недействительный дескриптор сервиса;
- ERROR_INVALID_PARAMETER — неправильные параметры;
- ERROR_INSUFFICIENT_BUFFER — в буфере недостаточно памяти.

Параметр hService должен содержать дескриптор сервиса, атрибуты безопасности которого получает функция. Этот дескриптор должен быть получен посредством функций CreateService или OpenService.

Параметр dwSecurityInformation имеет тип SECURITY_INFORMATION, который является синонимом типа DWORD. В этом параметре должны быть установлены флаги, которые отмечают, какие атрибуты безопасности сервиса должны быть прочитаны функцией. Для установки флагов используются те же символические константы, что и в соответствующем параметре функции SetServiceObjectSecurity.

Параметр lpSecurityDescriptor должен указывать на буфер, в который функция запишет дескриптор безопасности сервиса. Дескриптор безопасности будет записан в относительном формате.

В параметре cbBufSize должна быть задана длина буфера, на который указывает параметр lpSecurityDescriptor.

Параметр pcbBytesNeeded должен указывать на переменную типа DWORD, в которую функция в случае неудачного завершения запишет требуемую длину буфера.

Примечание

Получить атрибуты безопасности сервиса может только владелец этого сервиса или пользователь, для которого в список DACL этого объекта включен раз-

решающий элемент с правом READ_CONTROL. Кроме того, для чтения содержимого списка SACL пользователь должен обладать привилегией SE_SECURITY_NAME, которая включена, и правом доступа ACCESS_SYSTEM_SECURITY.

Листинг 45.6. Получение владельца сервиса

```
#include <windows.h>
#include <stdio.h>

int main()
{
  SC_HANDLE hManager;             // дескриптор менеджера сервисов
  SC_HANDLE hService;             // дескриптор сервиса

  DWORD dwLengthOfSd = 0;         // длина SD
  SECURITY_DESCRIPTOR *lpSd;      // дескриптор безопасности сервиса

  PSID lpOwner;                   // указатель на SID владельца
  BOOL bOwnerDefaulted = FALSE;  // признак владельца по умолчанию

  DWORD  dwLengthOfUserName = 0;      // длина имени учетной записи
  DWORD  dwLengthOfDomainName = 0;    // длина имени домена

  LPTSTR  lpUserName = NULL;      // указатель на имя домена
  LPTSTR  lpDomainName = NULL;    // указатель на имя домена

  SID_NAME_USE type_of_SID;       // тип учетной записи

  DWORD dwRetCode;    // код возврата

  // инициализируем, так как NULL выдаст ошибку
  lpSd = (SECURITY_DESCRIPTOR*) new char[0];

  // связываемся с менеджером сервисов
  hManager = OpenSCManager(
    NULL,             // локальная машина
    NULL,             // активная база данных сервисов
    GENERIC_READ);    // читаем управляющую информацию
  if (hManager == NULL)
```

```
{
  dwRetCode = GetLastError();
  printf("Open SC manager failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// открываем сервис
hService = OpenService(
  hManager,         // дескриптор менеджера сервисов
  "PlugPlay",       // имя сервиса
  READ_CONTROL);    // режим доступа
if (hService == NULL)
{
  dwRetCode = GetLastError();
  printf("Open service failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// определяем длину идентификатора безопасности сервиса
if (!QueryServiceObjectSecurity(
  hService,         // дескриптор сервиса
  OWNER_SECURITY_INFORMATION,  // получаем владельца сервиса
  lpSd,             // адрес SD
  dwLengthOfSd,     // длина буфера
  &dwLengthOfSd))   // требуемая длина буфера
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
    // захватываем память под дескриптор безопасности
    lpSd = (SECURITY_DESCRIPTOR*)new char[dwLengthOfSd];
  else
  {
    printf("Query Service Object Security failed.\n");
    printf("The last error code: %u\n", dwRetCode);
```

```
    return dwRetCode;
  }
}

// читаем идентификатор безопасности сервиса
if (!QueryServiceObjectSecurity(
  hService,        // дескриптор сервиса
  OWNER_SECURITY_INFORMATION,  // получаем владельца мьютекса
  lpSd,            // адрес SD
  dwLengthOfSd,    // длина буфера
  &dwLengthOfSd))  // требуемая длина буфера
{
  dwRetCode = GetLastError();
  printf("Query Service Object Security failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// получаем SID владельца сервиса
if (!GetSecurityDescriptorOwner(
  lpSd,          // адрес дескриптора безопасности
  &lpOwner,      // адрес указателя на SID владельца
  &bOwnerDefaulted))   // признак владельца по умолчанию
{
  dwRetCode = GetLastError();
  printf("Get security descriptor owner failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// проверяем, есть ли владелец
if (lpOwner == NULL)
{
  printf("The owner is not present.\n");

  return 0;
}
```

```
if (bOwnerDefaulted)
  printf("The owner is defaulted.\n");
else
  printf("The owner is not defaulted.\n");

// определяем длину имени домена
if(!LookupAccountSid(
  NULL,              // ищем на локальном компьютере
  lpOwner,           // указатель на SID
  lpUserName,        // имя пользователя
  &dwLengthOfUserName,     // длина имени пользователя
  lpDomainName,      // определяем имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &type_of_SID))     // тип учетной записи
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
  {
    // распределяем память под имя пользователя и имя домена
    lpUserName = (LPTSTR) new char[dwLengthOfUserName];
    lpDomainName = (LPTSTR) new char[dwLengthOfDomainName];
  }
  else
  {
    printf("Lookup account SID for length failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// определяем имя учетной записи по SID
if(!LookupAccountSid(
  NULL,              // ищем на локальном компьютере
  lpOwner,           // указатель на SID пользователя
  lpUserName,        // имя пользователя
  &dwLengthOfUserName,     // длина имени пользователя
  lpDomainName,      // определяем имя домена
```

```
    &dwLengthOfDomainName,   // длина имени домена
    &type_of_SID))     // тип учетной записи
  {
    dwRetCode = GetLastError();

    printf("Lookup account SID failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }

  printf("The owner name: %s\n", lpUserName);
  printf("The owner domain name: %s\n", lpDomainName);

  // освобождаем память
  delete[] lpUserName;
  delete[] lpDomainName;
  delete[] lpSd;

  CloseServiceHandle(hService);
  CloseServiceHandle(hManager);

  return 0;
}
```

45.8. Защита ключей реестра

Прежде чем перейти к функциям, предназначенным для доступа к атрибутам безопасности ключей реестра, перечислим специфические права доступа, которые могут контролироваться системой безопасности при доступе к реестру.

Следующие символические константы используются для задания специфических прав доступа к ключам реестра:

- `KEY_QUERY_VALUE` — запрос данных из ключа;
- `KEY_SET_VALUE` — запись данных в ключ;
- `KEY_CREATE_SUB_KEY` — создание дочернего ключа;
- `KEY_ENUMERATE_SUB_KEYS` — перечисление ключей;
- `KEY_NOTIFY` — получение сообщения об изменении в ключе;
- `KEY_CREATE_LINK` — создание ссылки на ключ.

Кроме того, для задания прав доступа к ключам реестра определены следующие символические константы:

- KEY_READ — включает права доступа STANDARD_RIGHTS_READ, KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, KEY_NOTIFY, ~SYNCHRONIZE;
- KEY_WRITE — включает права доступа STANDARD_RIGHTS_WRITE, KEY_SET_VALUE, CREATE_SUB_KEY, ~SYNCHRONIZE;
- KEY_EXECUTE — включает права доступа KEY_READ и ~SYNCHRONIZE;
- KEY_ALL_ACCESS — полный доступ к ключу реестра.

Теперь перейдем к описанию функций, используемых для доступа к атрибутам безопасности ключей реестра (см. листинг 45.7). Для установки атрибутов безопасности ключа реестра используется функция RegSetKeySecurity, которая имеет следующий прототип:

```
LONG  RegSetKeySecurity(
  HKEY  hKey,           // дескриптор ключа
  SECURITY_INFORMATION  SecurityInformation, // управляющие флаги
  PSECURITY_DESCRIPTOR  pSecurityDescriptor  // дескриптор безопасности
);
```

В случае успешного завершения функция вернет значение ERROR_SUCCESS. В случае неудачного завершения функции код ошибки можно получить посредством вызова функции GetLastError. Параметры функции RegSetKeySecurity имеют следующее назначение.

Параметр hKey должен содержать дескриптор ключа, для которого устанавливаются атрибуты безопасности.

Параметр SecurityInformation имеет тип SECURITY_INFORMATION, который является синонимом типа DWORD. В этом параметре должны быть установлены флаги, которые отмечают, какие атрибуты безопасности ключа реестра должны быть изменены. Для установки флагов используются следующие символические константы:

- OWNER_SECURITY_INFORMATION — владелец объекта;
- GROUP_SECURITY_INFORMATION — первичная группа владельца объекта;
- DACL_SECURITY_INFORMATION — список управления доступом DACL;
- SACL_SECURITY_INFORMATION — список управления доступом SACL.

Параметр pSecurityDescriptor должен указывать на дескриптор безопасности, в котором установлены новые атрибуты безопасности ключа реестра.

Примечание

Изменять атрибуты безопасности ключа реестра может только владелец этого ключа или пользователь, который имеет соответствующие стандартные права

доступа. Кроме того для изменения содержимого списка управления доступом SACL пользователь должен обладать привилегией SE_SECURITY_NAME, которая находится в разрешенном состоянии.

Для получения атрибутов безопасности ключа реестра используется функция RegGetKeySecurity, которая имеет следующий прототип:

```
LONG  RegGetKeySecurity (
  HKEY  hKey,              // дескриптор ключа
  SECURITY_INFORMATION  SecurityInformation,    // управляющие флаги
  PSECURITY_DESCRIPTOR  pSecurityDescriptor,    // SD
  LPDWORD  lpcbSecurityDescriptor               // длина буфера
);
```

В случае успешного завершения функция вернет значение ERROR_SUCCESS. В случае неудачного завершения функции код ошибки можно получить посредством вызова функции GetLastError.

Параметры функции RegGetKeySecurity имеют следующее назначение.

Параметр hKey должен содержать дескриптор ключа, атрибуты безопасности которого получает функция.

Параметр SecurityInformation имеет тип SECURITY_INFORMATION, который является синонимом типа DWORD. В этом параметре должны быть установлены флаги, которые отмечают, какие атрибуты безопасности ключа реестра должны быть получены функцией. Для установки флагов используются такие же символические константы, как и для соответствующего параметра функции RegSetKeySecurity.

Параметр pSecurityDescriptor должен указывать на буфер, в который функция запишет дескриптор безопасности ключа реестра. Дескриптор безопасности будет записан в относительном формате.

Параметр lpcbSecurityDescriptor должен указывать на переменную типа DWORD, которая содержит длину буфера. В случае недостаточной длины буфера функция вернет код ERROR_INSUFFICIENT_BUFFER, а по этому адресу запишет требуемую длину буфера.

Замечание

Получить атрибуты безопасности ключа реестра может только владелец этого ключа или пользователь, для которого в список DACL этого объекта включен разрешающий элемент с правом READ_CONTROL. Кроме того, для чтения содержимого списка SACL пользователь должен обладать привилегией SE_SECURITY_NAME, которая находится в разрешенном состоянии, и правом доступа ACCESS_SYSTEM_SECURITY.

Листинг 45.7. Получение владельца ключа реестра

```
#include <windows.h>
#include <stdio.h>

int main()
{
  HKEY hKey;        // ключ реестра

  DWORD dwLengthOfSd = 0;             // длина SD
  SECURITY_DESCRIPTOR *lpSd = NULL;  // SD ключа реестра

  PSID lpOwner;     // указатель на SID владельца
  BOOL bOwnerDefaulted = FALSE;       // признак владельца по умолчанию

  DWORD  dwLengthOfUserName = 0;      // длина имени учетной записи
  DWORD  dwLengthOfDomainName = 0;    // длина имени домена

  LPTSTR  lpUserName = NULL;          // указатель на имя домена
  LPTSTR  lpDomainName = NULL;        // указатель на имя домена

  SID_NAME_USE type_of_SID;           // тип учетной записи

  DWORD dwRetCode;      // код возврата

  // открываем ключ каталога
  dwRetCode = RegOpenKeyEx(
    HKEY_LOCAL_MACHINE,     // локальная машина
    NULL,                   // открыть ключ локальной машины
    0,                      // зарезервировано
    KEY_QUERY_VALUE,        // получаем данные из ключа
    &hKey);                 // адрес дескриптора ключа
  if (dwRetCode != ERROR_SUCCESS)
  {
    dwRetCode = GetLastError();
    printf("RegOpenKeyEx failed.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
```

```
}

// определяем длину идентификатора безопасности сервиса
dwRetCode = RegGetKeySecurity(
  hKey,             // дескриптор сервиса
  OWNER_SECURITY_INFORMATION,  // получаем владельца ключа реестра
  lpSd,             // адрес SD
  &dwLengthOfSd);  // требуемая длина буфера
if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
  // захватываем память под дескриптор безопасности
  lpSd = (SECURITY_DESCRIPTOR*)new char[dwLengthOfSd];
else
{
  printf("RegGetKeySecurity failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// читаем идентификатор безопасности ключа реестра
dwRetCode = RegGetKeySecurity(
  hKey,             // дескриптор сервиса
  OWNER_SECURITY_INFORMATION,  // получаем владельца ключа реестра
  lpSd,             // адрес SD
  &dwLengthOfSd);  // требуемая длина буфера
if (dwRetCode != ERROR_SUCCESS)
{
  printf("RegGetKeySecurity failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// получаем SID владельца мьютекса
if (!GetSecurityDescriptorOwner(
  lpSd,          // адрес дескриптора безопасности
  &lpOwner,      // адрес указателя на SID владельца
  &bOwnerDefaulted))   // признак владельца по умолчанию
{
```

```
  dwRetCode = GetLastError();
  printf("Get security descriptor owner failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// проверяем, есть ли владелец
if (lpOwner == NULL)
{
  printf("The owner is not present.\n");

  return 0;
}

if (bOwnerDefaulted)
  printf("The owner is defaulted.\n");
// определяем длину имени домена
if(!LookupAccountSid(
  NULL,               // ищем на локальном компьютере
  lpOwner,            // указатель на SID
  lpUserName,         // имя пользователя
  &dwLengthOfUserName,     // длина имени пользователя
  lpDomainName,       // определяем имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &type_of_SID))      // тип учетной записи
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
  {
    // распределяем память под имя пользователя и имя домена
    lpUserName = (LPTSTR) new char[dwLengthOfUserName];
    lpDomainName = (LPTSTR) new char[dwLengthOfDomainName];
  }
  else
  {
    printf("Lookup account SID for length failed.\n");
    printf("Error code: %d\n", dwRetCode);
```

```
      return dwRetCode;
    }
  }

  // определяем имя учетной записи по SID
  if(!LookupAccountSid(
    NULL,              // ищем на локальном компьютере
    lpOwner,           // указатель на SID пользователя
    lpUserName,        // имя пользователя
    &dwLengthOfUserName,     // длина имени пользователя
    lpDomainName,      // определяем имя домена
    &dwLengthOfDomainName,   // длина имени домена
    &type_of_SID))     // тип учетной записи
  {
    dwRetCode = GetLastError();

    printf("Lookup account SID failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }

  printf("Key owner name: %s\n", lpUserName);
  printf("Key owner domain name: %s\n", lpDomainName);
    // освобождаем память
  delete[] lpUserName;
  delete[] lpDomainName;
  delete[] lpSd;

  RegCloseKey(hKey);

  return 0;
}
```

45.9. Защита объектов пользователя

Объектами пользователя будем называть объекты, доступ к которым осуществляется через дескриптор, тип которого отличается от типа HANDLE. Некоторые объекты пользователя могут содержать атрибуты безопасности. К таким объектам относятся, например: станции окна (window station) и рабочие

столы (desktop). Ниже перечислены специфические права доступа, которые могут устанавливаться в элементах списков управления доступом для контроля доступа субъектов к таким объектам пользователя.

Следующие специфические права доступа могут использоваться для контроля доступа к станциям окон:

- WINSTA_ENUMDESKTOPS — перечисление объектов рабочего стола;
- WINSTA_READATTRIBUTES — чтение свойств станции окон;
- WINSTA_ACCESSCLIPBOARD — доступ к общему буферу (clipboard);
- WINSTA_CREATEDESKTOP — создание рабочего стола;
- WINSTA_WRITEATTRIBUTES — установка свойств станции окон;
- WINSTA_ACCESSGLOBALATOMS — доступ к глобальным атомам;
- WINSTA_EXITWINDOWS — завершение работы пользователя;
- WINSTA_ENUMERATE — перечисление станций окон;
- WINSTA_READSCREEN — чтение содержимого экрана.

Следующие специфические права доступа могут использоваться для контроля доступа к рабочим столам:

- DESKTOP_READOBJECTS — чтение объектов рабочего стола;
- DESKTOP_CREATEWINDOW — создание окна на рабочем столе;
- DESKTOP_CREATEMENU — создание меню на рабочем столе;
- DESKTOP_HOOKCONTROL — установка перехватчиков (hooks) на окна;
- DESKTOP_JOURNALRECORD — выполнение журнальных записей;
- DESKTOP_JOURNALPLAYBACK — просмотр журнальных записей;
- DESKTOP_ENUMERATE — перечисление рабочих столов;
- DESKTOP_WRITEOBJECTS — запись объектов на рабочий стол;
- DESKTOP_SWITCHDESKTOP — переключение рабочих столов.

Теперь перейдем к описанию функций, используемых для доступа к атрибутам безопасности объектов пользователя (см. листинг 45.8). Для установки атрибутов безопасности объекта пользователя используется функция SetUserObjectSecurity, которая имеет следующий прототип:

```
BOOL  SetUserObjectSecurity (
  HANDLE  hObj,          // дескриптор объекта пользователя
  PSECURITY_INFORMATION  pSIRequested,   // управляющие флаги
  PSECURITY_DESCRIPTOR  pSD              // указатель на SD
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — FALSE. В случае неудачного завершения функции код ошибки

можно получить посредством вызова функции `GetLastError`. Параметры функции `SetUserObjectSecurity` имеют следующее назначение.

Параметр `hObj` должен содержать дескриптор объекта пользователя, для которого устанавливаются новые атрибуты безопасности.

Параметр `pSIRequested` имеет тип `SECURITY_INFORMATION`, который является синонимом типа `DWORD`. В этом параметре должны быть установлены флаги, которые отмечают, какие атрибуты безопасности объекта пользователя должны быть изменены функцией. Для установки флагов используются следующие символические константы:

- `OWNER_SECURITY_INFORMATION` — владелец объекта;
- `GROUP_SECURITY_INFORMATION` — первичная группа владельца объекта;
- `DACL_SECURITY_INFORMATION` — список управления доступом DACL;
- `SACL_SECURITY_INFORMATION` — список управления доступом SACL.

Параметр `pSD` должен указывать на дескриптор безопасности, в котором установлены новые атрибуты безопасности для объекта пользователя.

Примечание

Изменять атрибуты безопасности объекта пользователя может только владелец этого объекта или пользователь, который имеет соответствующие стандартные права доступа. Кроме того, для изменения содержимого списка управления доступом SACL пользователь должен обладать привилегией `SE_SECURITY_NAME`, которая находится во включенном состоянии.

Для получения атрибутов безопасности объекта пользователя используется функция `GetUserObjectSecurity`, которая имеет следующий прототип:

```
BOOL  GetUserObjectSecurity (
  HANDLE   hObj,                // дескриптор объекта пользователя
  PSECURITY_INFORMATION  pSIRequested,   // управляющие флаги
  PSECURITY_DESCRIPTOR  pSD,             // указатель на SD
  DWORD    nLength,             // длина буфера
  LPDWORD  lpnLengthNeeded      // требуемая длина буфера
);
```

В случае успешного завершения функция вернет ненулевое значение, а в случае неудачи — `FALSE`. В случае неудачного завершения функции код ошибки можно получить посредством вызова функции `GetLastError`. Параметры функции `GetUserObjectSecurity` имеют следующее назначение.

Параметр `hObj` должен содержать дескриптор объекта пользователя, атрибуты безопасности которого получает функция.

Параметр `pSIRequested` имеет тип `SECURITY_INFORMATION`, который является синонимом типа `DWORD`. В этом параметре должны быть установлены флаги,

которые отмечают, какие атрибуты безопасности объекта пользователя должны быть получены функцией. Для установки флагов используются такие же символические константы, как и для соответствующего параметра функции `SetUserObjectSecurity`.

Параметр `pSD` должен указывать на буфер, в который функция запишет дескриптор безопасности объекта пользователя. Дескриптор безопасности будет записан в относительном формате.

Параметр `nLength` должен содержать длину буфера, на который указывает параметр `pSD`. Длина буфера задается в байтах.

Параметр `lpnLengthNeeded` должен указывать на переменную типа `DWORD`, в которую функция в случае неудачного завершения запишет требуемую длину буфера.

Примечание

Получить атрибуты безопасности объекта пользователя может только владелец этого объекта или пользователь, для которого в список DACL этого объекта включен разрешающий элемент с правом `READ_CONTROL`. Кроме того, для чтения содержимого списка SACL пользователь должен обладать привилегией `SE_SECURITY_NAME`, которая находится в разрешенном состоянии, и правом доступа `ACCESS_SYSTEM_SECURITY`.

Листинг 45.8. Получение владельца станции окон

```
#include <windows.h>
#include <stdio.h>

int main()
{
  HWINSTA hWindow = NULL;       // дескриптор окна

  DWORD dwLengthOfSd = 0;       // длина SD
  SECURITY_DESCRIPTOR *lpSd = NULL;  // дескриптор безопасности мьютекса

  // получаем информацию о владельце окна
  SECURITY_INFORMATION si = OWNER_SECURITY_INFORMATION;

  PSID lpOwner;    // указатель на SID владельца
  BOOL bOwnerDefaulted = FALSE;       // признак владельца по умолчанию

  DWORD  dwLengthOfUserName = 0;      // длина имени учетной записи
```

```
DWORD  dwLengthOfDomainName = 0;    // длина имени домена

LPTSTR  lpUserName = NULL;          // указатель на имя домена
LPTSTR  lpDomainName = NULL;        // указатель на имя домена

SID_NAME_USE type_of_SID;           // тип учетной записи

DWORD dwRetCode;   // код возврата

// получаем дескриптор станции окон
hWindow = GetProcessWindowStation();

// определяем длину идентификатора безопасности окна
if (!GetUserObjectSecurity(
  hWindow,          // дескриптор станции окна
  &si,              // получаем владельца окна
  lpSd,             // адрес SD
  dwLengthOfSd,     // определяем длину
  &dwLengthOfSd))   // требуемая длина буфера
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
    // захватываем память под дескриптор безопасности
    lpSd = (SECURITY_DESCRIPTOR*)new char[dwLengthOfSd];
  else
  {
    printf("GetUserObjectSecurity for length failed.\n");
    printf("The last error code: %u\n", dwRetCode);

    return dwRetCode;
  }
}

// читаем идентификатор безопасности окна
if (!GetUserObjectSecurity(
  hWindow,          // дескриптор станции окна
  &si,              // получаем владельца окна
  lpSd,             // адрес SD
```

```
  dwLengthOfSd,     // определяем длину
  &dwLengthOfSd))   // требуемая длина буфера
{
  dwRetCode = GetLastError();

  printf("GetUserObjectSecurity failed.\n");
  printf("The last error code: %u\n", dwRetCode);

  return dwRetCode;
}

// получаем SID владельца окна
if (!GetSecurityDescriptorOwner(
  lpSd,                 // адрес дескриптора безопасности
  &lpOwner,             // адрес указателя на SID владельца
  &bOwnerDefaulted))    // признак владельца по умолчанию
{
  dwRetCode = GetLastError();
  printf("Get security descriptor owner failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
}

// проверяем, есть ли владелец
if (lpOwner == NULL)
{
  printf("The owner is not present.\n");

  return 0;
}

if (bOwnerDefaulted)
  printf("The owner is defaulted.\n");

// определяем длину имени домена
if(!LookupAccountSid(
  NULL,               // ищем на локальном компьютере
  lpOwner,            // указатель на SID
```

```
  lpUserName,        // имя пользователя
  &dwLengthOfUserName,     // длина имени пользователя
  lpDomainName,      // определяем имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &type_of_SID))     // тип учетной записи
{
  dwRetCode = GetLastError();

  if (dwRetCode == ERROR_INSUFFICIENT_BUFFER)
  {
    // распределяем память под имя пользователя и имя домена
    lpUserName = (LPTSTR) new char[dwLengthOfUserName];
    lpDomainName = (LPTSTR) new char[dwLengthOfDomainName];
  }
  else
  {
    printf("Lookup account SID for length failed.\n");
    printf("Error code: %d\n", dwRetCode);

    return dwRetCode;
  }
}

// определяем имя учетной записи по SID
if(!LookupAccountSid(
  NULL,              // ищем на локальном компьютере
  lpOwner,           // указатель на SID пользователя
  lpUserName,        // имя пользователя
  &dwLengthOfUserName,     // длина имени пользователя
  lpDomainName,      // определяем имя домена
  &dwLengthOfDomainName,   // длина имени домена
  &type_of_SID))     // тип учетной записи
{
  dwRetCode = GetLastError();

  printf("Lookup account SID failed.\n");
  printf("Error code: %d\n", dwRetCode);

  return dwRetCode;
```

```
  }

  printf("Key owner name: %s\n", lpUserName);
  printf("Key owner domain name: %s\n", lpDomainName);

  // освобождаем память
  delete[] lpUserName;
  delete[] lpDomainName;
  delete[] lpSd;

  return 0;
}
```

Приложение

Описание компакт-диска

Папка "Листинги" содержит тексты листингов из книги.

Папка "Проекты" содержит проекты с программами из папки "Листинги".

Примечание

Проекты не содержат исполняемых файлов. Проект восстанавливается по файлам с расширениями: cpp, h, dsp, dsw.

Предметный указатель

В

Г

Д

З

И

К

М

Н

О

П

(окончание рубрики см. на стр. 1048)

Р

С

Т

У

Ф

(продолжение рубрики см. на стр. 1052)

Я