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".

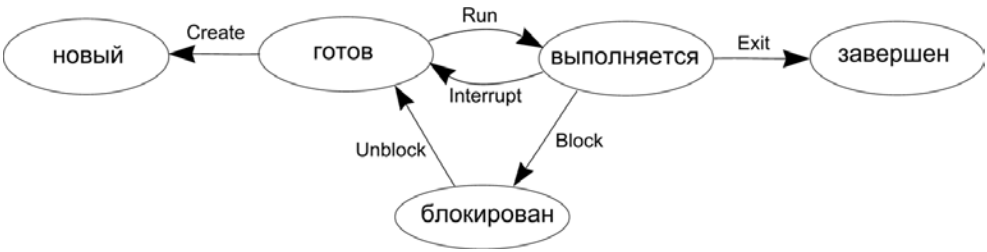

Рис. 2.1. Модель пяти состояний потока.

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.

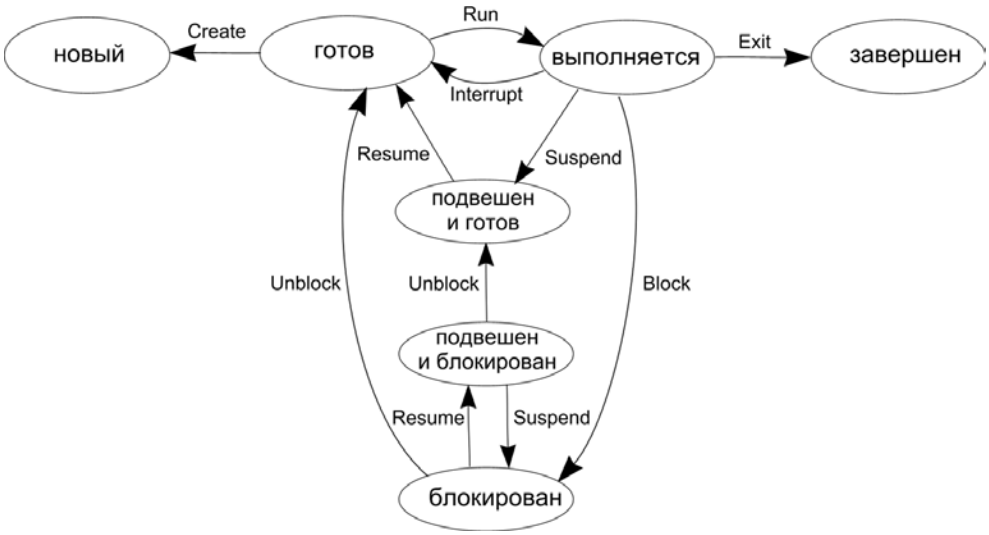

Рис. 2.2. Модель семи состояний потока

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.

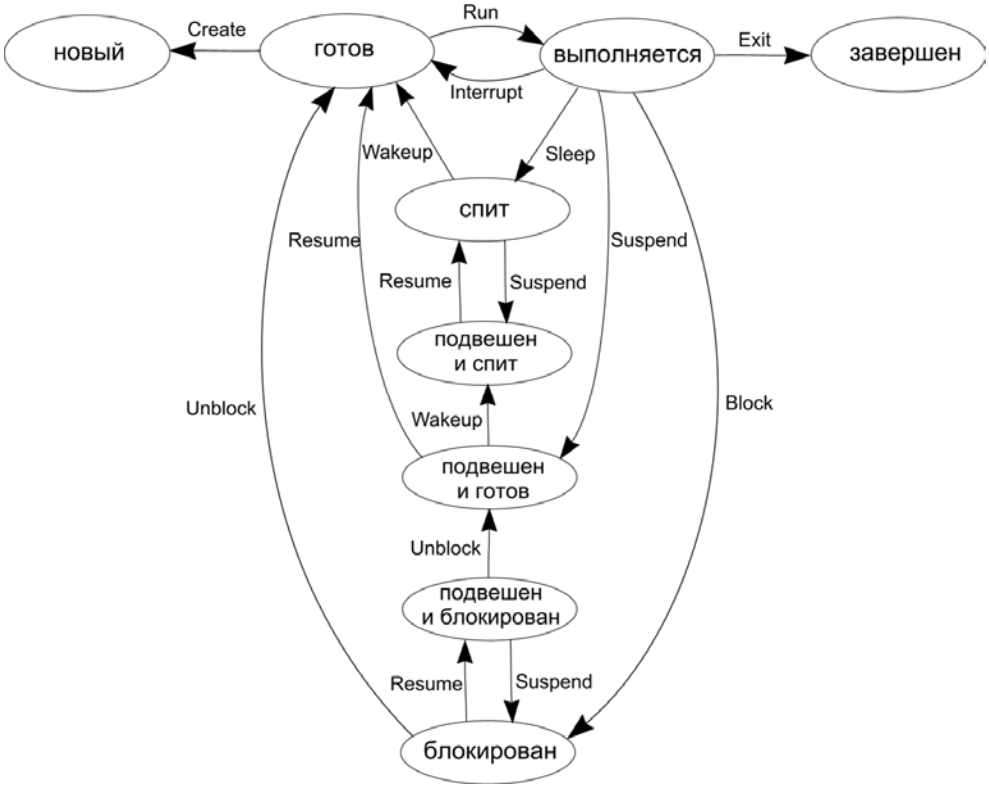

Рис. 2.3. Модель девяти состояний потока

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.

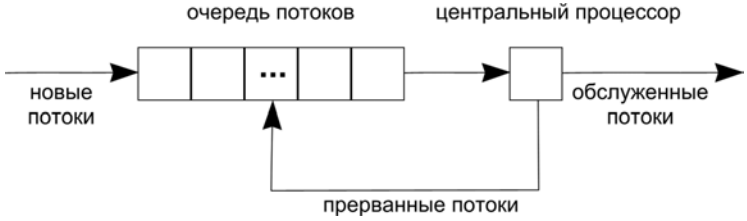

Рис. 2.4. Циклическое обслуживание потоков

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.

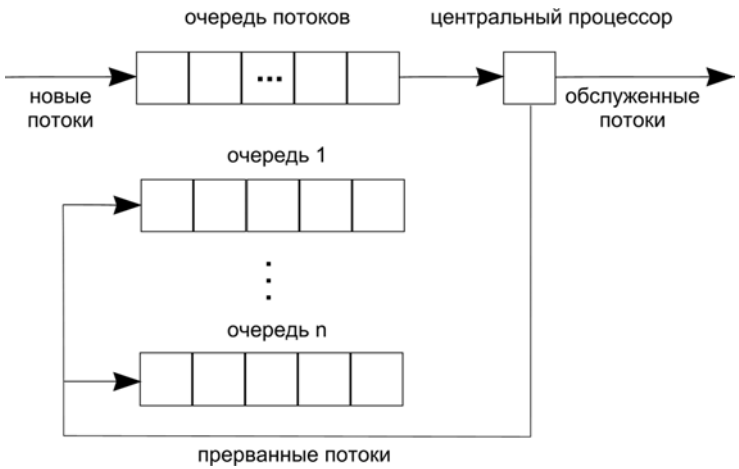

Рис. 2.5. Дисциплины обслуживания с несколькими очередями

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).

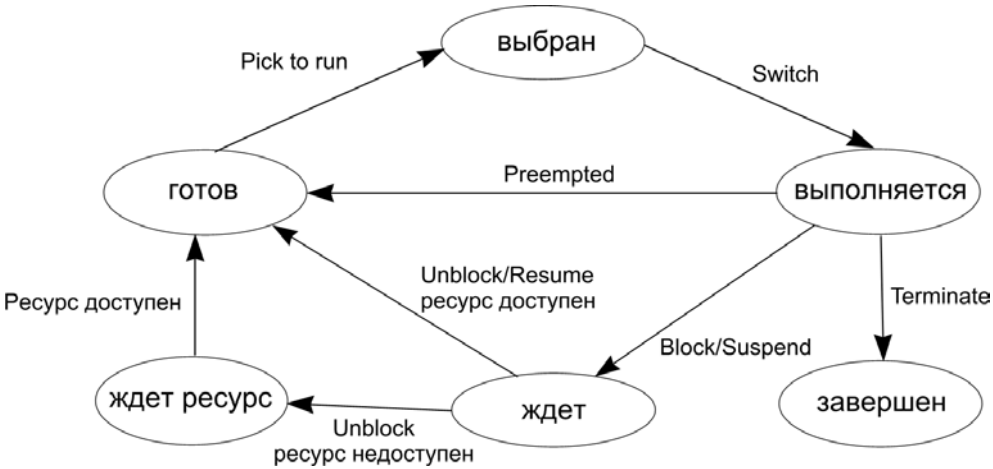

Рис. 3.1. Модель состояний потока в Windows 2000

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
);
```