МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Кафедра прикладной математики

Вл. Пономарев

Системное программирование

Учебно-методическое пособие

Озерск — 2010

УДК 681.3.06
П56

Пономарев В.В. Системное программирование. Учебно-методическое пособие. Редакция 05.05.2010. Озерск: ОТИ НИЯУ МИФИ, 2010. — 60 с., ил.

Пособие предназначено для изучения дисциплины «Системное программирование» студентами, обучающимися по специальности «Программное обеспечение вычислительной техники и автоматизированных систем». В пособии описываются системные функции операционной системы *Windows*, используемые для работы с процессами, потоками, объектами ядра, каналами, почтовыми ящиками, проецируемыми в память файлами. Рассматриваются также вопросы создания сервисов NT.

Рецензенты:
1) Начальник отдела ПО ЗАО «Озерск Телеком» Н.Г. Ведюшкин.
2) Ст. преподаватель кафедры ПМ ОТИ НИЯУ МИФИ Р.А. Федотов.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

ОТИ НИЯУ МИФИ, 2010

Содержание

Введение

Системное программирование охватывает программное обеспечение, разработка которого требует детального знания устройства компьютера и операционной системы. Традиционно к области системного программирования относятся базы данных, языковые и инструментальные системы, драйверы и т.п., требующие от программиста высокой квалификации, а от программы — высокой степени отлаженности и надежности.

В настоящем пособии описываются функции API Windows для работы с объектами ядра, процессами, потоками, средствами синхронизации процессов и потоков, а также основы построения сервисов.

В главе «Обработка ошибок» рассматриваются вопросы контроля выполнения функций API Windows, классификация возвращаемых ими значений и получение информации об ошибках.

В главе «Объекты ядра» описываются основные сведения об объектах ядра операционной системы Windows, рассматриваются вопросы безопасности этих объектов, их структура.

В главе «Процессы и потоки» приводятся базовые сведения о процессах и потоках, рассматриваются функции по управлению процессами и потоками, управление приоритетом потоков.

В главе «Синхронизация потоков» описываются средства синхронизации процессов и потоков, такие как *interlocked*-функции, критические секции, мьютексы и семафоры, а также *Wait*-функции.

В главе «Асинхронный ввод-вывод» описываются функции, предназначенные для организации асинхронного ввода-вывода.

В главе «Средства межпроцессного взаимодействия» рассматриваются основные средства взаимодействия между процессами — каналы, почтовые ящики и проецируемые в память файлы. Приведены функция для работы с этими средствами и основные алгоритмы.

В главе «Сервисы» рассматриваются сервисы NT. Описывается структура сервисов, назначение и содержание основных функций сервисов, приведены описания системных функций для управления сервисами, примеры их использования.

Обработка ошибок

В основе системного программирования для операционной системы Windows лежит использование системных функций API. Различные системные функции API возвращают разные результаты. Иногда возвращаемый результат является признаком успешности выполнения функции, иногда служит результатом выполнения задачи.

В таблице 1 показаны типы данных для возвращаемых значений большинства функций API Windows.

Таблица 1. Стандартные типы значений, возвращаемых функциями API

Тип данных	Значение, свидетельствующее об ошибке
VOID	Функция всегда (или почти всегда) выполняется успешно.
BOOL	Если вызов функции завершается неудачно, возвращается 0; в остальных случаях возвращаемое значение отлично от 0.
HANDLE	Если вызов функции заканчивается неудачно, то обычно возвращается NULL; в остальных случаях HANDLE идентифицирует объект, которым Вы можете манипулировать. Некоторые функции возвращают HANDLE со значением INVALID_HANDLE_VALUE, равным –1, свидетельствующим о неуспешности.
PVOID	Если вызов функции заканчивается неудачно, возвращается NULL; в остальных случаях PVOID сообщает адрес блока данных в памяти.
LONG или DWORD	Функции, которые сообщают значения каких-либо счетчиков, обычно возвращают LONG или DWORD. Если по какой-то причине функция не сумела сосчитать то, что Вы хотели, она обычно возвращает 0 или –1 (все зависит от конкретной функции).

При возникновении ошибки нужно разобраться, почему вызов данной функции оказался неудачен.

За каждой ошибкой закреплен свой код (32-битное число). Функция API, обнаружив ошибку, через механизм локальной памяти потока сопоставляет соответствующий код ошибки с вызывающим потоком. Это позволяет потокам работать независимо друг от друга, не вмешиваясь в чужие ошибки. Возвращаемое значение будет указывать на то, что произошла какая-то ошибка. Код ошибки можно узнать, вызвав функцию GetLastError.

```
DWORD GetLastError();
```

Функция возвращает 32-битный код ошибки для данного потока.

Список кодов ошибок, определенных *Microsoft*, содержится в заголовочном файле WinError.h. Приведем фрагмент этого файла (который содержит описание нескольких тысяч ошибок):

```
// MessageId: ERROR_SUCCESS
// The operation completed successfully.
#define ERROR_SUCCESS                  0L
#define NO_ERROR                       0L  // dderror
#define SEC_E_OK                       ((HRESULT)0x00000000L)
//
// MessageId: ERROR_INVALID_FUNCTION
// Incorrect function.
#define ERROR_INVALID_FUNCTION         1L  // dderror
//
// MessageId: ERROR_FILE_NOT_FOUND
// The system cannot find the file specified.
#define ERROR_FILE_NOT_FOUND           2L
//
// MessageId: ERROR_PATH_NOT_FOUND
// The system cannot find the path specified.
#define ERROR_PATH_NOT_FOUND           3L
```

Как видно, для каждой ошибки в данном файле определена константа, которая должна использоваться в программе. Например, для ошибки с кодом 2 определена константа ERROR_FILE_NOT_FOUND.

Функцию GetLastError нужно вызывать сразу же после неудачного вызова функции Windows, иначе код ошибки может быть потерян.

GetLastError возвращает последнюю ошибку, возникшую в потоке. Если этот поток вызывает другую функцию API и все проходит успешно, код последней ошибки не перезаписывается и не используется как индикатор благополучного вызова функции. Несколько функций API нарушают это правило и изменяют код последней ошибки. В документации Platform SDK утверждается: после успешного выполнения функция API изменяет код последней ошибки.

Некоторые функции API всегда завершаются успешно, но по разным причинам. Например, попытка создать объект ядра «событие» с определенным именем может быть успешна либо потому, что объект действительно создан, либо потому, что такой объект уже есть. Но иногда нужно знать причину успеха. Для возврата этой информации *Microsoft* использует механизм установки кода последней ошибки. При успешном выполнении некоторых функций API вызов GetLastError дает дополнительную информацию. К числу таких функций относится, например, CreateEvent.

Полезно отслеживать код последней ошибки в процессе отладки. *Microsoft Visual* C++ 6.0 позволяет настраивать окно *Watch* так, чтобы оно всегда показывало код и описание последней ошибки в текущем потоке. Для этого введите в какую-нибудь строку в окне *Watch* значение «@err, hr» (рисунок 1).

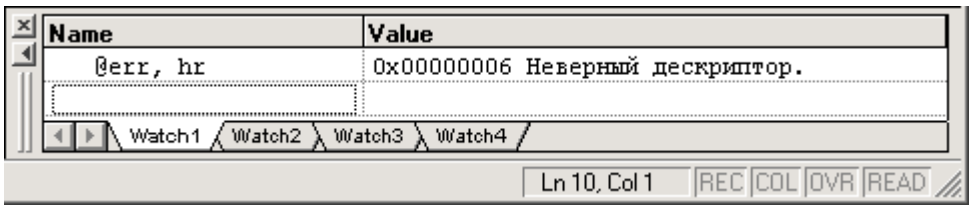

Рисунок 1 - Отображение последней ошибки в окне Watch

С *Visual Studio* 6.0 поставляется утилита *Error Lookup*, которая позволяет получать описание ошибки по ее коду (рисунок 2).

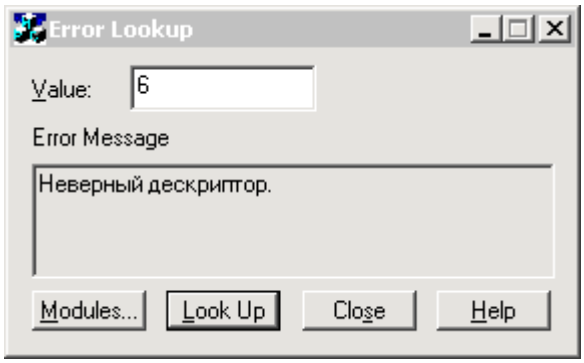

Рисунок 2 - Утилита Error Lookup

Если приложение обнаруживает ошибку, оно может уведомлять пользователя при помощи функции MessageBox. Для получения описания ошибки следует использовать функцию FormatMessage, которая преобразует код ошибки в описание на языке пользователя.

```
DWORD FormatMessage (
    DWORD dwFlags,      // флаги
    LPCVOID pSource,    // источник
    DWORD dwMessageId,  // код ошибки
    DWORD dwLanguageId, // идентификатор языка
    PTSTR pszBuffer,    // буфер для сообщения
    DWORD nSize,        // размер буфера
    va_list *Arguments  // аргументы сообщения (массив)
);
```

Пример применения этой функции:

```
HLOCAL hlocal = NULL;   // буфер для строки с описанием ошибки
DWORD dwError = GetLastError();
```

```
BOOL fOk = FormatMessage(
    FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_ALLOCATE_BUFFER,
    NULL,
    dwError,
    MAKELANGID(LANG_RUSSIAN, SUBLANG_DEFAULT),
    (LPTSTR) & hlocal,
    0,
    NULL);
if (hlocal != NULL) {
    MessageBox( 0, (LPCTSTR)hlocal, "Error", MB_OK );
    LocalFree( hlocal );
} else {
    MessageBox( 0, "Error FormatMessage.", "Error", MB_OK );
}
```

В данном примере кода формируется системное сообщение, при этом буфер для сообщения резервирует система (на это указывают флаги). Поскольку сообщение формируется системой, источник описания равен нулю. В других случаях он может указывать на программный модуль, например, на DLL.

При помощи макроса MAKELANGID формируется идентификатор языка, который необходимо использовать для формирования сообщения. Поскольку буфер резервирует система, вместо размера буфера подставляется нулевое значение.

Параметры используются тогда, когда нужно подставлять фактические значения в сообщение, например, номер или имя диска и т.п.

Установить текущую ошибку можно при помощи функции SetLastError.

```
VOID SetLastError(DWORD dwErrorCode);
```

При этом подставляемый код должен содержать код ошибки в битах 0-15, бит 29 должен содержать 1, а биты 30-31 — код 1 для формирования сообщения, код 2 для формирования предупреждения и код 3 для формирования ошибки.

Объекты ядра

Объект ядра — это блок памяти, выделенный ядром и принадлежащий ядру. Фактически это структура данных, описывающая объект.

Объектами ядра являются:

- файл;
- порт завершения (*completion port*);
- проекция файла (*file mapping*);
- задание (*job*);
- почтовый ящик (*mailslot*);
- канал (*pipe*);
- мьютекс (*mutex*), семафор (*semaphore*);
- процесс (*process*);
- поток (*thread*);
- ожидаемый таймер (*waitable timer*);
- маркер доступа (*access token*).

Объект ядра может быть изменен только операционной системой. Приложения используют API для доступа к объектам ядра. При создании объекта ядра возвращается его описатель, имеющий тип HANDLE. Этот описатель используется далее в функциях API. Значения описателей зависят от процесса и не могут быть использованы в других процессах обычным образом.

Ядро знает, сколько процессов используют объект ядра, учитывая процессы при помощи счетчика пользователей. При достижении счетчика значения ноль объект ядра уничтожается.

Объект ядра может быть защищен дескриптором безопасности (*security descriptor*), который описывает тот, кто создает объект. Дескриптор безопасности используется в основном при написании серверных и системных приложений. Клиентское приложение вместо дескриптора защиты обычно передает NULL. При этом создается объект с защитой по умолчанию, что означает, что использовать объект ядра могут только владелец (тот, кто создал) и члены группы администраторов.

Для описания дескриптора безопасности используется структура:

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD Length,
    LPVOID lpSecurityDescriptor,
    BOOL InheritHandle
};
```

здесь lpSecurityDescriptor — адрес инициализированной структуры SECURITY_DESCRIPTOR.

В следующем примере показано, как инициализировать дескриптор безопасности. Сначала нужно создать дескриптор и инициализировать его:

```
// Дескриптор безопасности
PSECURITY_DESCRIPTOR pSD = 0;
pSD = (PSECURITY_DESCRIPTOR)malloc(SECURITY_DESCRIPTOR_MIN_LENGTH);
if ( pSD == NULL )
    goto cleanup;
if (!InitializeSecurityDescriptor(pSD,
SECURITY_DESCRIPTOR_REVISION))
    goto cleanup;
```

Здесь cleanup — метка, соответствующая ошибке.

Затем нужно сформировать DACL (*discretionary access-control list*):

```
if ( !SetSecurityDescriptorDacl(pSD, TRUE, (PACL)NULL, FALSE) )
    goto cleanup;
```

Наконец, формируется структура SECURITY_ATTRIBUTES:

```
// Атрибуты безопасности
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = pSD;
sa.bInheritHandle = TRUE;
```

Чтобы получить доступ к существующему объекту ядра, необходимо указать операции, которые предполагается выполнять с объектом.

Например, при вызове функции OpenFileMapping первым параметром мы передаем флаг FILE_MAP_READ, указывающий на то, что мы предполагаем выполнение операции чтения:

```
HANDLE hFile = OpenFileMapping( FILE_MAP_READ, FALSE, "FileName" );
```

Доступ предоставляется, если тип защиты объекта разрешает доступ, иначе возвращается NULL и функция GetLastError возвращает значение 5 — ERROR_ACCESS_DENIED.

Кроме объектов ядра программы создают также объекты User и GDI, которые не имеют параметров безопасности.

Создание объекта ядра

Как только процесс вызывает функцию, содержащую объект ядра, ядро создает запись в таблице описателей процесса. Все функции, создающие объекты ядра, возвращают описатель, привязанный к процессу (индекс в таблице описателей). Описатель может быть использован лю

бым потоком процесса. При передаче неверного описателя функции возвращают 6 — ERROR_INVALID_HANDLE.

Функции, возвращающие описатель, в случае неудачи возвращают либо 0, либо –1 (INVALID_HANDLE_VALUE). Поэтому нужно быть внимательным при проверке возвращаемого значения. Например, следующий код работать не будет:

```
HANDLE hFile = CreateFile( ... );
if ( !hFile ) {
    // ошибка
}
```

Правильно будет:

```
HANDLE hFile = CreateFile( ... );
if ( hFile == INVALID_HANDLE_VALUE ) {
    // ошибка
}
```

Закрытие объекта ядра

По окончании работы с объектом ядра его нужно закрыть при помощи функции CloseHandle.

```
BOOL CloseHandle(HANDLE);
```

Эта функция уменьшает счетчик пользователей объекта ядра на единицу. Функция возвращает FALSE в случае, если описатель неверный, а GetLastError возвращает ERROR_INVALID_HANDLE.

Совместное использование объектов ядра

Иногда объекты ядра совместно используются несколькими процессами. Например:

- объекты «проекции файлов» позволяют двум процессам совместно использовать одни и те же блоки данных;
- почтовые ящики и именованные каналы позволяют обмениваться данными с процессами, исполняемыми на других машинах;
- мьютексы, семафоры и события позволяют синхронизировать потоки разных процессов.

Есть три механизма совместного использования объектов ядра.

Наследование описателя объекта

Наследование описателя объекта используется в случае, если процессы связаны «родственными» отношениями, — когда один процесс является порождением другого.

При создании объекта ядра родительский процесс сообщает системе, что ему требуется наследуемый описатель данного объекта (наследуются описатели, но не объекты).

Чтобы создать наследуемый описатель, родительский процесс создает дескриптор безопасности и формирует атрибуты безопасности с признаком наследования:

```
SECURITY_ATTRIBUTES sa;
sa.Length = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
```

Здесь создается дескриптор безопасности с защитой по умолчанию.

Для передачи наследуемого описателя дочернему процессу родительский процесс может использовать аргумент командной строки или переменную блока окружения.

При создании дочернего процесса указываем, что описатели наследуются. Операционная система, создав дочерний процесс, копирует в него таблицу описателей родительского процесса и увеличивает счетчик пользователей объекта ядра.

В ситуации, когда нужно породить два дочерних процесса, один из которых наследует описатели, а другой — нет, используется изменение флага наследования при помощи функции SetHandleInformation.

```
BOOL SetHandleInformation (
    HANDLE hObject,     // объект ядра
    DWORD dwMask,       // флаги, которые подлежат изменению
    DWORD dwFlags       // новые значения флагов
);
```

С каждым описателем связано два флага:

```
#define HANDLE_FLAG_INHERIT             0x00000001
#define HANDLE_FLAG_PROTECT_FROM_CLOSE  0x00000002
```

Чтобы установить флаг наследования, например, нужно вызвать функцию SetHandleInformation следующим образом:

```
SetHandleInformation( hObj, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT );
```

а чтобы сбросить, вызов функции должен иметь вид:

```
SetHandleInformation( hObj, HANDLE_FLAG_INHERIT, 0 );
```

Здесь hObj — описатель объекта ядра.

Именованные объекты

Именованные объекты ядра создаются функциями:

CreateMutex,
CreateEvent,
CreateSemaphore,
CreateWaitableTimer,
CreateFileMapping,
CreateJobObject.

Имя объекта задается последним параметром pszName. Если он равен NULL, создается анонимный объект ядра. Максимальная длина имени объекта ядра ограничена константой MAX_PATH (260).

Система не дает сведений об именах объектов ядра, поэтому следующий код дает ошибку:

```
HANDLE hMutex = CreateMutex( 0, 0, "MyObject" );
HANDLE hSem = CreateSemaphore( 0, 1, 1, "MyObject" );
DWORD dwError = GetLastError();
```

Возвращаемое значение ошибки 6 — ERROR_INVALID_HANDLE.

Когда создается объект ядра с именем, которое уже используется для данного типа объекта, система возвращает описатель имеющегося, а не нового объекта ядра, если атрибуты безопасности разрешают использовать данный объект.

При создании объекта ядра с именем всегда можно узнать, создан новый объект, или создана копия описателя существующего объекта.

```
HANDLE hMutex = CreateMutex( &sa, FALSE, "MyObject" );
if ( GetLastError() == ERROR_ALREADY_EXISTS ) {
    // создана копия описателя
} else {
    // создан новый объект
}
```

Другой способ получить описатель существующего объекта ядра — вместо функции вида CreateXXXX использовать функцию вида OpenXXXX, которая требует имя существующего объекта.

Именованные объекты могут быть также использованы для предотвращения повторного запуска приложения.

Дублирование описателя

Дублирование описателя объекта ядра выполняется при помощи функции DuplicateHandle.

```
BOOL DuplicateHandle (
    HANDLE hSourceProcessHandle,    // описатель процесса-источника
    HANDLE hSourceHandle,           // дублируемый описатель
    HANDLE hTargetProcessHandle,    // описатель процесса-назначения
```

```
    PHANDLE phTargetHandle,          // возвращаемый дубликат описателя
    DWORD dwDesiredAccess            // требуемый доступ
    BOOL bInheritHandle,             // наследуемый описатель
    DWORD dwOptions                  // флаги
);
```

Параметр dwOptions может принимать значение 0 или комбинацию флагов:

DUPLICATE_SAME_ACCESS — игнорировать dwDesiredAccess,

DUPLICATE_CLOSE_SOURCE — закрыть описатель-оригинал.

Пример использования этой функции для получения дубликата описателя текущего процесса:

```
HANDLE hProcess;
DuplicateHandle(
    // описатель процесса, к которому относится фиктивный описатель
    GetCurrentProcess(),
    // фиктивный описатель процесса
    GetCurrentProcess(),
    // описатель процесса, к которому относится настоящий описатель
    GetCurrentProcess(),
    // возвращаемый настоящий описатель процесса
    & hParentThread,
    // игнорируется из-за DUPLICATE_SAME_ACCESS
    0,
    // новый описатель не наследуемый
    FALSE,
    // атрибуты безопасности нового описателя
    // те же, что и у фиктивного описателя
    DUPLICATE_SAME_ACCESS
);
```

Процессы и потоки

Процесс состоит из объекта ядра «процесс» и адресного пространства, в котором находятся код и данные всех EXE и DLL модулей. Поток состоит из объекта ядра «поток», через который операционная система управляет потоком, и стека потока.

Процесс ничего не исполняет и служит контейнером потоков. Код исполняют потоки в адресном пространстве процесса. Потоки используют существенно меньше ресурсов, чем процессы. Все основные ресурсы принадлежат процессу, потоки используют их совместно или по раздельности.

При инициализации процесса операционная система создает первичный поток. Другие потоки создает программист с целью:

- обеспечения равномерной загрузки процессора;
- параллельного выполнения алгоритмов;
- выполнения вспомогательных задач в фоновом режиме.

Обычно первичный поток отвечает за интерфейс приложения, другие потоки выполняют вычисления, ввод-вывод и другие операции.

Создание и завершение процесса

Для создания процесса обычно ничего делать не нужно — он создается автоматически при запуске приложения. В других случаях можно создавать процессы при помощи функции CreateProcess:

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,                // имя exe-файла
    LPTSTR lpCommandLine,                     // командная строка
    LPSECURITY_ATTRIBUTES lpProcessAttributes, // дескр безопасности
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  // дескр безопасности
    BOOL bInheritHandles,                     // наслед описателя
    DWORD dwCreationFlags,                    // флаги создания
    LPVOID lpEnvironment,                     // блок окружения
    LPCTSTR lpCurrentDirectory,               // рабочий каталог
    LPSTARTUPINFO lpStartupInfo,              // информация о старте
    LPPROCESS_INFORMATION lpProcessInformation // инф о процессе
);
```

Параметр «имя exe-файла» указывает на исполняемый модуль (путь к файлу).

Параметр «командная строка» задает аргументы командной строки, передаваемые входной функции первичного потока при старте процесса. Этот параметр может быть равен нулю.

Дескрипторы безопасности определяют, может ли описатель процесса или первичного потока наследоваться дочерними процессами. Если дескриптор равен нулю, наследование запрещено.

Параметр «наследование описателя» определяет, наследует ли процесс описатели родительского процесса.

Флаги создания определяют класс приоритета процесса и дополнительные условия его создания.

Параметр «блок окружения» определяет указатель на блок окружения процесса. Если параметр равен нулю, используется блок окружения родительского процесса.

Параметр «рабочий каталог» задает диск и каталог нового процесса.

Параметр «информация о старте» — это указатель на структуру STARTUPINFO, описывающую, в каком состоянии будет находиться окно процесса после старта процесса.

Параметр «информация о процессе» — это указатель на структуру PROCESS_INFORMATION, в которой по завершении данной функции будут содержаться описатели и идентификаторы процесса и первичного потока.

При успешном завершении данной функции возвращается ненулевое значение, при неуспешном — нулевое.

Завершается процесс одним из четырех способов:

- входная функция первичного потока завершает свое исполнение (предпочтительный способ);
- один из потоков процесса вызывает функцию ExitProcess (нежелательный способ);
- поток другого процесса вызывает функцию TerminateProcess (нежелательный способ);
- все потоки процесса завершили свое исполнение (редкий случай).

Наилучшим способом завершения процесса является нормальное завершение входной функции первичного потока. В этом случае:

- любые созданные C++ объекты уничтожаются соответствующими деструкторами;
- система освобождает память, занимаемую стеком потока;
- система устанавливает код завершения процесса (возвращаемый входной функцией первичного потока);
- счетчик пользователей объекта ядра «процесс» уменьшается на единицу.

Создание потока

Поток начинает выполнение с некоторой входной функции, которая выглядит примерно следующим образом:

```
DWORD WINAPI ThreadProc( PVOID pvParam ) {
    DWORD dwExitCode = 0;
    . . .
    return dwExitCode;
}
```

Правила относительно функции потока:

1) функции потока передается единственный параметр, — указатель на произвольную структуру данных (программист решает, что будет передаваться);

2) функция потока возвращает операционной системе код завершения;

3) функция потока должна обходиться локальными переменными.

Поток создает функция CreateThread.

```
HANDLE CreateThread (
    PSECURITY_ATTRIBUTES psa,           // атрибуты безопасности
    DWORD cbStack,                      // размер стека потока
    PTHREAD_START_ROUTINE pfnStartAddr, // адрес функции потока
    PVOID pvParam,                      // параметры функции потока
    DWORD fdwCreate                     // флаг создания потока
    PWORD pdwThreadID           // возвращаемый идентификатор потока
);
```

При этом создается объект ядра и из адресного пространства процесса выделяется стек потока. Поток выполняется в контексте же потока, что и родительский поток. Поэтому он получает доступ ко всем описателям объектов ядра, всей памяти и стекам всех потоков процесса. За счет этого потоки процесса легко взаимодействуют друг с другом.

По умолчанию, если параметр cbStack равен нулю, потоку выделяется 1 Мб памяти.

Флаг создания потока fdwCreate либо равен нулю, либо значению CREATE_SUSPENDED. Во втором случае поток не запускается на выполнение сразу после создания, а ожидает принудительного запуска вызовом функции ResumeThread.

Параметр pdwThreadID — это адрес переменной, которая получит значение идентификатора потока.

Завершение потока

Поток завершает свою работу в случаях:

а) функция потока завершает свою работу и возвращает управление (рекомендуемый способ);

б) поток самоуничтожается вызовом функции ExitThread;

в) другой поток или процесс вызывают функцию TerminateThread;

г) завершается процесс, содержащий данный поток.

В случае а):

- C++ объекты, созданные потоком, корректно уничтожаются соответствующими деструкторами;
- корректно освобождается память стека потока;
- устанавливается код завершения потока;
- счетчик пользователей объекта ядра «поток» уменьшается на единицу.

В случае б):

- освобождаются ресурсы, выделенные потоку операционной системой;
- C++ объекты не уничтожаются;
- устанавливается код завершения, указанный в вызове функции ExitThread.

Устройство потока

На рисунке 3 показано, что выполняет система во время создания потока. Сначала создается объект ядра «поток». При этом счетчику числа пользователей присваивается начальное значение, равное 2. Инициализируется счетчик числа простоев и код завершения. Объект переводится в состояние «Занято».

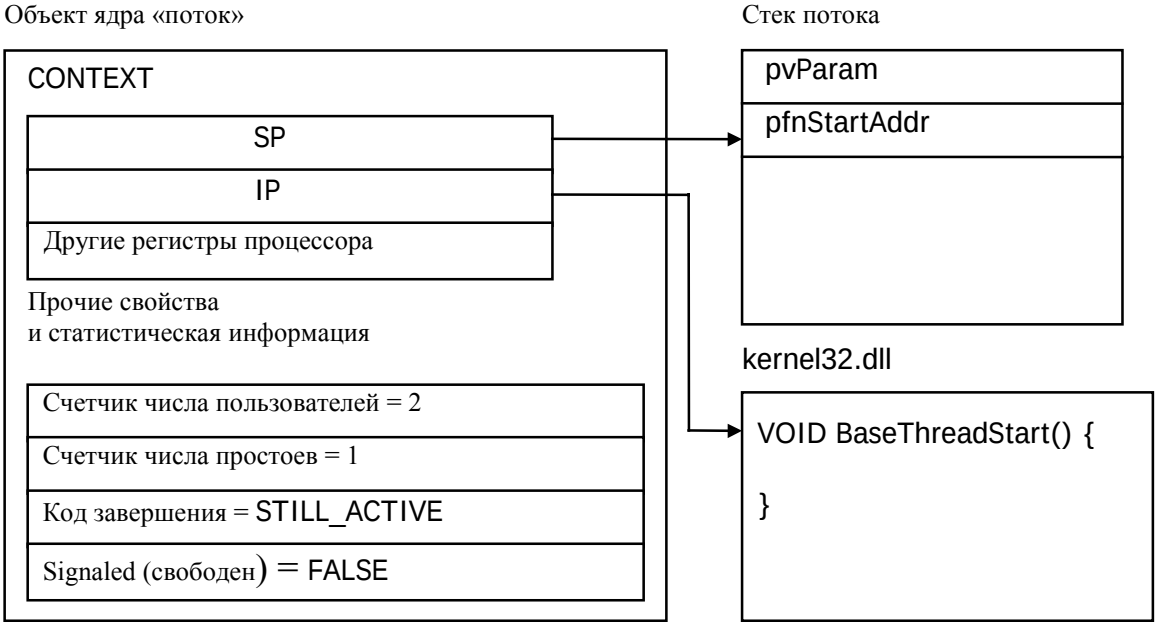

Рисунок 3 - Создание и инициализация потока

У каждого потока есть собственный набор регистров процессора, называемый «контекстом». Контекст отражает состояние регистров на

момент последнего исполнения потока и записывается в структуру CONTEXT.

В момент инициализации потока указатель стека потока SP указывает на адрес функции потока, а счетчик команд IP — на адрес недокументированной функции BaseThreadStart. Действия этой функции поясняет следующий фрагмент кода:

```
VOID BaseThreadStart(PTHERAD_START_ROUTINE pfnStAddr, PVOID pvParam) {
    __try {
        ExitThread((pfnStartAddr) (pvParam));
    }
    __except(UnhandledExceptionFilter(GetExceptionInformation())) {
        ExitProcess(GetExceptionCode());
    }
}
```

При этом функция потока включается во фрейм структурной обработки исключений.

Для старта первичной функции потока используется аналогичная недокументированная функция BaseProcessStart.

При проектировании многопоточных приложений вместо функции CreateThread нужно использовать функцию _beginthreadex. У этой функции тот же список параметров, что и у функции CreateThread, отличаются лишь их имена и типы.

При этом проект должен быть связан с многопоточными библиотеками, вместо однопоточных по умолчанию:

Mtdll.h (содержит описание структуры tiddata),

libcmt.lib,

msvcrt.lib.

Описатели процессов и потоков

Описатель процесса или потока используется для изменения характеристик процесса или потока, например, для изменения приоритета.

Функция GetCurrentProcess возвращает фиктивный описатель процесса (псевдо-описатель). Его получение не влияет на счетчик пользователей объекта ядра «процесс». Фиктивный описатель не требует своего закрытия при помощи функции CloseHandle.

```
HANDLE GetCurrentProcess(VOID);
```

Функция GetCurrentThread возвращает фиктивный описатель потока.

```
HANDLE GetCurrentThread(VOID);
```

Фиктивные описатели могут быть использованы в функциях, требующих описатели процесса и потока.

Функция GetCurrentProcessId возвращает уникальный в пределах системы идентификатор процесса.

```
DWORD GetCurrentProcessId(VOID);
```

Функция GetCurrentThreadId возвращает уникальный в пределах системы идентификатор потока.

```
DWORD GetCurrentThreadId(VOID);
```

Преобразование фиктивного описателя в настоящий

Иногда требуется получить не фиктивный, а настоящий описатель процесса или потока. Рассмотрим пример кода, в котором родительский поток передает дочернему свой описатель.

```
DWORD WINAPI ParentThread( PVOID pvParam ) {
    HANDLE hParentThread = GetCurrentThread();
    CreateThread( NULL, 0, ChildThread, hParentThread, 0, NULL );
    . . .
}

DWORD WINAPI ChildThread( PVOID pvParam ) {
    HANDLE hParentThread = (HANDLE)pvParam;
    FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
    GetThreadTimes( hParentThread, & ftCreationTime,
        & ftExitTime, & ftKernelTime, & ftUserTime );
    . . .
}
```

Поскольку описатель hParentThread является фиктивным, функция GetThreadTimes возвращает временные характеристики не родительского, а своего потока.

Для преобразования фиктивного описателя в настоящий используется функция DuplicateHandle, описанная в разделе «Объекты ядра». Правильное описание функции родительского потока может иметь вид:

```
DWORD WINAPI ParentThread( PVOID pvParam ) {
    HANDLE hParentThread;
    DuplicateHandle(
      GetCurrentProcess(),  // handle процесса, к которому относится
                            // фиктивный handle потока
       GetCurrentThread(), // фиктивный handle родительского потока
       GetCurrentProcess(),// handle процесса, к которому относится
                            // новый, настоящий handle потока
       & hParentThread,    // возвращаемый настоящий handle потока
       0,                      // игнорируется
```

```
        FALSE,                  // новый handle не наследуемый
        DUPLICATE_SAME_ACCESS   // атр безопасности нового описателя
                                // те же, что и у фиктивного handle
    );
    CreateThread( NULL, 0, ChildThread, hParentThread, 0, NULL );
    . . .
}
```

Соответственно, функция дочернего потока также изменяется — настоящий описатель потока следует закрыть:

```
DWORD WINAPI ChildThread( PVOID pvParam ) {
    HANDLE hParentThread = (HANDLE)pvParam;
    FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
    GetThreadTimes( hParentThread, & ftCreationTime,
        & ftExitTime, & ftKernelTime, & ftUserTime );
    CloseHandle( hParentThread );
    . . .
}
```

При помощи функции DuplicateHandle можно преобразовать и фиктивный описатель процесса. Пример приведен в разделе «Дублирование описателя» (глава «Объекты ядра»).

Приоритеты потоков

Приоритет потока — это число от нуля (самый низкий приоритет) до 31 (самый высокий). При очередного выборе потока на выполнение, система сначала рассматривает потоки с приоритетом 31 и ставит их на выполнение по очереди. При этом ни один поток с более низким приоритетом не выполняется. Потоки с более высоким приоритетом всегда вытесняют потоки с более низким приоритетом, независимо от того, использовал поток с низким приоритетом свой квант времени, или нет.

Ни один поток не может иметь приоритет, равный нулю. В системе есть только один такой поток — поток обнуления страниц (*zero page thread*).

Приоритет потока задается двумя абстрактными понятиями — классом приоритета процесса и относительным приоритетом потока.

Операционная система *Windows* поддерживает шесть классов приоритета, приведенных в таблице 2.

Таблица 2. Классы приоритетов потоков

Класс приоритета	Описание
Real-time	Потоки в этом процессе немедленно реагируют на событие, обеспечивая выполнение критических по времени задач. Такие потоки вытесняют даже компоненты операционной системы.
High	Потоки в этом процессе немедленно реагируют на события, обеспечивая выполнение критических по времени задач.
Above normal	Промежуточный между *normal* и *high*.
Normal	Потоки в этом процессе не предъявляют особых требований к предоставлению процессорного времени.
Below normal	Промежуточный между *normal* и *idle*.
Idle	Потоки в этом процессе выполняются, когда система не занята другой работой. Используется для утилит, работающих в фоновом режиме.

Класс приоритета процесса определяет назначение приложения и его взаимодействие с другими приложениями. После его выбора необходимо установить относительные приоритеты потоков процесса. В системе предусмотрено семь относительных приоритетов потоков, приведенных в таблице 3.

Таблица 3. Относительные приоритеты потоков

Относительный приоритет	Приоритет
Time-critical	31 в классе *real-time* и 15 в других классах.
Highest	Normal + 2.
Above normal	Normal + 1.
Normal	Поток выполняется с обычным приоритетом.
Below normal	Normal - 1.
Lowest	Normal - 2.
Idle	16 в классе *real-time* и 1 в других классах.

В таблице 4 показано, как формируется приоритет потока в операционной системе *Windows* NT. Заметим, что таблица справедлива для определенной операционной системы, в других версиях фактические значения приоритетов могут быть другими.

Таблица 4. Приоритеты потоков

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

Для установки класса приоритета процесса используется функция SetPriorityClass.

```
BOOL SetPriorityClass(
    HANDLE hProcess,     // описатель процесса
    DWORD fdwPriroty     // требуемый класс приоритета
);
```

Для задания класса приоритета процесса используются константы:

REALTIME_PRIORITY_CLASS
HIGH_PRIORITY_CLASS
ABOVE_NORMAL_PRIORITY_CLASS
NORMAL_PRIORITY_CLASS
BELOW_NORMAL_PRIORITY_CLASS
IDLE_PRIORITY_CLASS

Для установки относительного приоритета потока используется функция SetThreadPriority.

```
BOOL SetThreadPriority(
    HANDLE hThread,      // описатель потока
    int nPriority        // требуемый приоритет потока
);
```

Для задания приоритета потока используются константы:

THREAD_PRIORITY_TIME_CRITICAL
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_IDLE

Заметим, что константы, используемые для задания класса приоритета процесса и относительного приоритета потока, не равны значениям самого приоритета, приведенным в таблице 4. Кроме того, некоторые константы доступны в проекте, только если установлен пакет SDK.

Определить текущий класс приоритета процесса можно при помощи функции GetPriorityClass.

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

Функция возвращает класс приоритета в виде одной из приведенных выше констант класса приоритета, или ноль в случае неудачи.

Определить текущий относительный приоритет потока можно при помощи функции GetThreadPriority.

```
int GetThreadPriority(
    HANDLE hThread       // описатель потока
);
```

Функция возвращает уровень приоритета в виде одной из приведенных выше констант относительного приоритета. При неудаче возвращается значение THREAD_PRIORITY_ERROR_RETURN.

Пример определения класса приоритета процесса:

```
DWORD c = GetPriorityClass( GetCurrentProcess() );
```

Для обычного процесса при этом возвращается значение 32.

Пример определения относительного приоритета потока:

```
int p = GetThreadPriority( GetCurrentThread() );
```

Возвращаемое значение для обычного потока при этом равно нулю.

Динамическое изменение уровня приоритета

Уровень приоритета потока, получаемый комбинацией класса приоритета процесса и относительного приоритета потока, называют базовым уровнем приоритета потока.

Система может изменять уровень приоритета потока в некоторых случаях, например, при возникновении событий, связанных с вводом-выводом. Система повышает уровень приоритета только тех потоков, базовый уровень которых находится в пределах 1-15, а сам этот диапазон приоритетов называется областью динамического приоритета.

Для отключения механизма динамического изменения приоритета используются функции SetProcessPriorityBoost и SetThreadPriorityBoost.

```
BOOL SetProcessPriorityBoost(
    HANDLE hProcess,            // описатель процеса
```

```
    BOOL DisablePriorityBoost   // динамическое изменение приоритета
);

BOOL SetThreadPriorityBoost(
    HANDLE hThread,             // описатель потока
    BOOL DisablePriorityBoost   // динамическое изменение приоритета
);
```

Второй параметр, равный TRUE, отключает динамическое изменение приоритета.

Функции управления потоками

Для приостановки выполнения любого потока используется функция SuspendThread.

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

Каждый поток имеет счетчик простоев (*suspend count*). Если он больше нуля, поток приостанавливается. Вызов функции SuspendThread увеличивает счетчик простоев на единицу. Функция возвращает предыдущее значение счетчика простоев.

Для возобновления выполнения приостановленного потока используется функция ResumeThread.

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

Действие этой функции противоположно предыдущей — она уменьшает счетчик простоев. Если значение счетчика становится равным нулю, система возобновляет выполнение потока.

Функция возвращает предыдущее значение счетчика в случае успеха, ноль в случае, если поток не был приостановлен ранее, и –1 в случае неудачи.

Функция SwitchToThread ставит на выполнение другой поток, если он есть.

```
BOOL SwitchToThread();
```

Сначала система проверяет, есть ли поток, которому не хватает процессорного времени. Если нет, функция немедленно возвращает управление. Если да, планировщик выделяет потоку дополнительный квант времени, при этом приоритет потока может быть ниже, чем у вызывающего функцию.

Для исключения потока из планирования на некоторое время используется функция Sleep.

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

Эта функция приостанавливает планирование потока на указанное количество миллисекунд. При вызове этой функции текущий поток немедленно завершает свой квант времени. Следует учитывать, что время выключения потока задается приблизительно.

Если задано выключить поток на 0 миллисекунд, поток просто завершает текущий квант.

Синхронизация потоков

Все потоки системы могут иметь доступ к системным ресурсам, таким, как кучи, последовательные порты, файлы, окна и т.п. Если один из потоков запрашивает доступ к системному ресурсу в монопольном режиме, другие потоки не могут использовать этот ресурс. С другой стороны, при доступе к ресурсу в разделяемом режиме потоки могут вызывать конфликты доступа к ресурсам.

Потоки должны взаимодействовать друг с другом в следующих двух основных случаях:

а) при совместном использовании разделяемого ресурса;

б) при уведомлении других потоков о завершении операций.

Для синхронизации взаимодействия потоков операционная система Windows предлагает множество различных механизмов для синхронизации как потоков одного процесса, так и потоков разных процессов.

Interlocked-функции

Во многих случаях взаимодействие потоков представляет атомарный доступ (atomic access), заключающийся в монопольном доступе к ресурсу одним из потоков. Рассмотрим пример, в котором два потока одного процесса используют общую переменную процесса:

```
// общая переменная процесса
long g_x_count = 0;
// поток 1
DWORD WINAPI Thread_1(PVOID) {
    g_x_count++;
    return 0;
}
// поток 2
DWORD WINAPI Thread_2(PVOID) {
    g_x_count++;
    return 0;
}
```

Периодический запуск данных потоков на выполнение предполагает, что глобальный счетчик g_x_count получит значение, равное количеству запусков потоков. Однако на самом деле счетчик получит некоторое произвольное значение, зависящее от состояния системы, в частности — от количества выполняемых потоков.

Если рассмотреть выполняемый потоками код на ассемблере, то он может иметь примерно следующий вид:

```
MOV     EAX, [g_x_count]    // переменная загружается в регистр
```

```
INC     EAX                 // регистр инкрементриуется
MOV     [g_x_count], EAX    // регистр записывается в переменную
RET                         // завершение
```

При нормальном выполнении потоков (один за другим) мы получим следующую последовательность команд процессора:

```
MOV     EAX, [g_x_count]    // поток 1: в регистр загружается 0
INC     EAX                 // поток 1: регистр инкрементриуется
MOV     [g_x_count], EAX    // поток 1: в переменную записывается 1
RET                         // поток 1: завершение

MOV     EAX, [g_x_count]    // поток 2: в регистр загружается 1
INC     EAX                 // поток 2: регистр инкрементриуется
MOV     [g_x_count], EAX    // поток 2: в переменную записывается 2
RET                         // поток 2: завершение
```

Однако, поскольку операционная система Windows является многозадачной и многопоточной, порядок выполнения операций может оказаться и другим:

```
MOV     EAX, [g_x_count]    // поток 1: в регистр загружается 0
INC     EAX                 // поток 1: регистр инкрементриуется

MOV     EAX, [g_x_count]    // поток 2: в регистр загружается 0
INC     EAX                 // поток 2: регистр инкрементриуется

MOV     [g_x_count], EAX    // поток 1: в переменную записывается 1
RET                         // поток 1: завершение
MOV     [g_x_count], EAX    // поток 2: в переменную записывается 1
RET                         // поток 2: завершение
```

Вследствие многопоточности операционной системы порядок выполнения операций потоками не является последовательным, как предполагает программист. Поэтому нужен механизм, гарантирующий атомарность выполнения операции над глобальной переменной. Таким механизмом являются interlocked-функции. Следующая функция изменяет значение переменной типа LONG на указанное значение.

```
LONG InterlockedExchangeAdd(
    PLONG plAddend,       // указатель на переменную типа LONG,
                          // к которой нужен атомарный доступ
    LONG lIncrement   // добавляемое значение
);
```

Функция возвращает предыдущее значение переменной.

С использованием этой функции потоки из приведенного выше примера могут быть записаны следующим образом:

```
// общая переменная процесса
long g_x_count = 0;
// поток 1
DWORD WINAPI Thread_1(PVOID) {
    InterlockedExchangeAdd( & g_x_count++, 1 );
    return 0;
}
// поток 2
DWORD WINAPI Thread_2(PVOID) {
    InterlockedExchangeAdd( & g_x_count++, 1 );
    return 0;
}
```

Interlocked-функции работают примерно следующим образом:

1) установить специальный битовый флаг процессора, указывающий, что данный адрес памяти занят;

2) считать значение из памяти в регистр;

3) изменить значение регистра;

4) если флаг сброшен, перейти к п. 2, иначе записать значение из регистра в переменную.

Флаг может быть сброшен, например, другим процессором.

Interlocked-функции работают очень быстро (примерно 50 тактов), так как при их выполнении не происходит перехода в режим ядра. Важным при использовании *interlocked*-функций является правильное выравнивание адреса переменной по границе 32 или 64 бита.

Следующие две функции заменяют содержимое переменной новым значением.

```
LONG InterlockedExchange(
    PLONG plTarget,   // указатель на переменную типа LONG,
                          // к которой нужен атомарный доступ
    LONG lValue       // новое значение
);

LONG InterlockedExchangePointer(
    PVOID* ppvTarget,     // указатель на указатель на переменную типа
LONG,
                          // к которой нужен атомарный доступ
    PVOID pvValue     // новое значение
);
```

Обе функции возвращают предыдущее значение переменной.

Функция InterlockedExchange может быть использована для реализации спин блокировки (spinlock).

```
// индикатор занятости ресурса
BOOL g_f_in_use = FALSE;
. . .
VOID SomeFunc() {
    // ожидаем доступа к ресурсу
    while ( InterlockedExchange( & g_f_in_use, TRUE ) == TRUE ) {
        Sleep( 0 );
    }
    // получаем доступ к ресурсу
    . . .
    // освобождаем ресурс
    InterlockedExchange( & g_f_in_use, FALSE );
}
```

Здесь цикл продолжается до тех пор, пока возвращаемое функцией InterlockedExchange значение равно TRUE, что означает, что ресурс занят. Недостаток спин блокировки заключается в непрерывном бесполезном использовании процессорного времени.

Следующие две функции выполняют операции атомарного сравнения и замены значения переменной.

```
LONG InterlockedCompareExchange(
    PLONG plDestination,    // указатель на переменную
    LONG lExchange,         // новое значение
    LONG lComparand         // значение для сравнения
);

LONG InterlockedCompareExchangePointer(
    PVOID* plDestination,   // указатель на указатель на переменную
    PVOID lExchange,        // указатель на новое значение
    PVOID lComparand        // указатель на значение для сравнения
);
```

Функции сравнивают значение переменной со значением для сравнения. Если эти значения равны, происходит замена значения переменной новым значением. Функции возвращают предыдущее значение переменной.

Заметим, что с помощью рассмотренных функций возможна синхронизация потоков процесса при доступе к переменной типа LONG.

Критические секции

Критическая секция как понятие — это часть кода программы, требующая монопольного доступа к разделяемым данным. При входе одно

го потока в критическую секцию некоторых разделяемых данных другие потоки блокируются при попытке войти в критическую секцию этих же разделяемых данных.

Для защиты критических секций потоков операционная система реализует механизм под названием «объект критическая секция». Объявление объекта «критическая секция» выполняется следующим образом:

```
CRITICAL_SECTION cs;
```

Таким образом, критическая секция — это некая структура данных, а приведенный код выделяет память под данный объект.

Далее критическая секция должна быть инициализирована при помощи функции InitializeCriticalSection.

```
VOID InitializeCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection // объект «критическ секция»
);
```

Для входа в критическую секцию потоки используют функцию EnterCriticalSection.

```
VOID EnterCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection // объект «критическ секция»
);
```

В случае нехватки памяти вызов этой функции может привести к исключению STATUS_INVALID_HANDLE.

По завершении работы с разделяемыми данными поток выходит из критической секции при помощи функции LeaveCriticalSection.

```
VOID LeaveCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection // объект «критическ секция»
);
```

Наконец, после использования критической секции она должна быть уничтожена вызовом функции DeleteCriticalSection. При этом освобождаются ресурсы, используемые объектом «критическая секция».

```
VOID DeleteCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection // объект «критическ секция»
);
```

Функция TryEnterCriticalSection предпринимает попытку войти в критическую секцию без блокировки.

```
BOOL TryEnterCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection // объект «критическ секция»
);
```

В случае входа в критическую секцию, данная функция возвращает ненулевое значение.

В отличие от функции EnterCriticalSection, данная функция немедленно возвращает управление, если объект «критическая секция» занят другим потоком, в то время как функция EnterCriticalSection переходит в блокированное состояние (исключается из процесса планирования до момента освобождения критической секции).

Следующий пример кода показывает использование критической секции потоком.

```
int g_x_count = 0;
CRITICAL_SECTION cs;

DWORD WINAPI CountThread(PVOID) {
    EnterCriticalSection( & cs );
    g_x_count++;
    LeaveCriticalSection( & cs );
    return 0;
}

void main() {
    InitializeCriticalSection( & cs );
    for ( int i = 0; i < 10; i++ ) {
        CreateThread( NULL, 0, CountThread, NULL, 0, NULL);
    }
    while ( g_x_count < 10 ) ;
    DeleteCriticalSection( & cs );
}
```

Критические секции могут быть использованы для синхронизации потоков одного процесса.

Синхронизация потоков с помощью объектов ядра

В данном разделе рассматриваются объекты ядра, которые используются для синхронизации взаимодействия процессов и потоков.

К синхронизирующим объектам ядра относятся:

- процесс;
- поток;
- задание (*job*);
- файл;
- консольный ввод;
- уведомление об изменении файла;
- событие;

- ожидаемый таймер (*waitable timer*);
- семафор и мьютекс.

Синхронизирующий объект ядра может находиться в одном из двух состояний:

- свободное (*signaled state*);
- занятое (*non-signaled state*).

Переход синхронизирующего объекта ядра из одного состояния в другое происходит по правилам, определенным для каждого объекта индивидуально.

Объект ядра находится в занятом состоянии, если процесс, сопоставленный с ним, выполняется, и переходит в свободное состояние по завершении данного процесса.

Потоки могут переходить в блокированное состояние и ждать освобождения одного или нескольких синхронизирующих объектов ядра.

Wait-функции

Wait-функция приостанавливает планирование некоторого потока до момента освобождения одного или нескольких синхронизирующих объектов ядра.

Функция WaitForSingleObject блокирует процесс до освобождения одного синхронизирующего объекта ядра.

```
DWORD WaitForSingleObject(
    HANDLE hHandle,        // описатель синхронизирующего объекта ядра
    DWORD dwMilliseconds   // максимальное время ожидания
);
```

Параметр dwMilliseconds задает максимальное время ожидания освобождения синхронизирующего объекта ядра. Для задания бесконечного времени ожидания используется константа INFINITE (равная –1).

В следующем примере функция WaitForSingleObject используется для ожидания завершения потока.

```
int g_x_count = 0;
CRITICAL_SECTION cs;
HANDLE hCountThread = 0;

DWORD WINAPI CountThread(PVOID) {
    EnterCriticalSection( & cs );
    g_x_count++;
    LeaveCriticalSection( & cs );
    return 0;
}

void main() {
```

```
    InitializeCriticalSection( & cs );
    hCountThread = CreateThread( 0, 0, CountThread, 0, 0, 0);
    if ( hCountThread == 0 ) {
        // поток не создан
        DeleteCriticalSection( & cs );
        return;
    }
    DWORD dwWaitResult = WaitForSingleObject( hCountThread, 1000 );
    switch ( dwWaitResult ) {
    case WAIT_OBJECT_0: // поток завершен
        MessageBox( 0, "Success.", "Wait", MB_OK );
        break;
    case WAIT_TIMEOUT:  // истекло заданное время ожидания
        MessageBox( 0, "Timeout.", "Wait", MB_OK );
        break;
    case WAIT_FAILED:   // неправильный вызов функции
        MessageBox( 0, "Failure.", "Wait", MB_OK );
        break;
    }
    DeleteCriticalSection( & cs );
}
```

Функция возвращается немедленно в случае, если ее выполнение невозможно (передан недействительный описатель). Для демонстрации этого состояния подставьте в вызов функции ноль вместо описателя. Состояние фиксируется константой WAIT_FAILED.

Функция нормально возвращается в случае, если синхронизирующий объект ядра перешел в свободное состояние. Это состояние фиксируется константой WAIT_OBJECT_0.

Функция нормально возвращается в случае, если истекло заданное время ожидания. Продемонстрировать состояние можно, если уменьшить время ожидания до значения, например, 10. Состояние фиксируется константой WAIT_TIMEOUT.

Функция WaitForMultipleObjects блокирует выполнение потока до момента перехода в свободное состояние одного или всех синхронизирующих объектов.

```
DWORD WaitForMultipleObjects(
    DWORD nCount,               // количество синхр. объектов
    CONST HANDLE *lpHandles,    // массив описателей объектов
    BOOL bWaitAll,              // вариант ожидания всех объектов
    DWORD dwMilliseconds        // максимальное время ожидания
);
```

Максимальное количество синхронизирующих объектов определено константой MAX_WAIT_OBJECTS, равной 64.

Если параметр bWaitAll равен значению TRUE, функция ожидает перехода в свободное состояние всех синхронизирующих объектов.

Пример использования этой функции предполагает, что поток ожидает завершения одного из трех других потоков.

```
HANDLE handles[3];
handles[0] = hThread_1;
handles[1] = hThread_2;
handles[2] = hThread_3;

DWORD dwWaitResult = WaitForMultipleObjects(3,handles,FALSE,5000);

switch ( dwWaitResult ) {
case WAIT_FAILED:
    . . .
    break;
case WAIT_TIMEOUT:
    . . .
    break;
case WAIT_OBJECT_0 + 0:
    // завершен поток Thread_1
    break;
case WAIT_OBJECT_0 + 1:
    // завершен поток Thread_2
    break;
case WAIT_OBJECT_0 + 2:
    // завершен поток Thread_3
    break;
}
```

События

Событие — самый примитивный объект ядра. Он содержит счетчик числа пользователей, тип события и текущее состояние.

События используются для уведомления об окончании каких-либо операций. Есть два типа событий:

- сбрасываемые вручную (*manual-reset event*);
- сбрасываемые автоматически (*auto-reset event*).

Сбрасываемые вручную события позволяют возобновить выполнение сразу нескольких потоков. События, сбрасываемые автоматически, позволяют возобновить выполнение только одного потока.

Событие создается функцией CreateEvent.

```
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes,   // атр безопасности
    BOOL bManualReset,                         // тип события
    BOOL bInitialState,                        // начальное состояние
    LPCTSTR lpName                             // имя объекта
);
```

Пример создания анонимного события с ручным сбросом и начальным состоянием «свободно»:

```
HANDLE hEvent = CreateEvent( NULL, TRUE, TRUE, NULL );
```

Если создано поименованное событие, далее можно получить его описатель при помощи функции OpenEvent.

```
HANDLE OpenEvent(
    DWORD dwDesiredAccess,  // требуемый доступ
    BOOL bInheritHandle,    // возможность наследования описателя
    LPCTSTR lpName          // имя объекта
);
```

Описатель поименованного события можно также получить при помощи вызова функции CreateEvent, в котором используется имя существующего события.

Для управления состоянием события используются две функции.

Функция SetEvent устанавливает в свободное состояние.

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

При успешном выполнении функция возвращает ненулевое значение.

Функция ResetEvent устанавливает занятое состояние события.

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

При успешном выполнении функция возвращает ненулевое значение.

Ожидаемые таймеры

Ожидаемый таймер — это объект ядра, самостоятельно переходящий в свободное состояние в определенное время или через регулярные интервалы времени.

Ожидаемый таймер создается функцией CreateWaitableTimer.

```
HANDLE CreateWaitableTimer(
    LPSECURITY_ATTRIBUTES lpTimerAttributes, // атрибуты безопасности
    BOOL bManualReset,                       // тип таймера
    LPCTSTR lpTimerName                      // имя объекта
);
```

Для получения описателя существующего ожидаемого таймера можно использовать функцию OpenWaitableTimer, аналогичную функции OpenEvent.

```
HANDLE OpenWaitableTimer(
    DWORD dwDesiredAccess,  // требуемый доступ
    BOOL bInheritHandle,    // возможность наследования описателя
    LPCTSTR lpTimerName     // имя объекта
);
```

Тип ожидаемого таймера может быть с автоматическим сбросом и сбросом вручную. Смысл этих понятий такой же, как и у событий.

При создании объекта «ожидаемый таймер» всегда создается объект, находящийся в занятом состоянии. Для установки времени перехода ожидаемого таймера в свободное состояние используется функция SetWaitableTimer.

```
BOOL SetWaitableTimer(
    HANDLE hTimer,                          // описатель
    const LARGE_INTEGER * pDueTime,         // интервал до первого
    LONG lPeriod,                           // периодичность
    PTIMERAPCROUTINE pfnCompletionRoutine, // процедура завершения
    LPVOID lpArgToCompletionRoutine,        // парам проц завершения
    BOOL fResume                            // флаг возобновления
);
```

Параметр pDueTime задает время первого срабатывания таймера, а параметр lPeriod — интервал последующих срабатываний. Если параметр lPeriod задать равным нулю, таймер сработает только один раз.

Если задан параметр pfnCompletionRoutine, в момент срабатывания таймера вызывается указанная функция.

Семафоры

Объект ядра «семафор» используется для учета множества однотипных ресурсов. Кроме счетчика числа пользователей, этот объект содержит счетчик максимального числа ресурсов и счетчик текущего числа ресурсов.

Семафоры подчиняются следующим правилам:

- семафор переходит в свободное состояние, когда счетчик числа текущих ресурсов становится больше нуля;

- если счетчик числа текущих ресурсов становится равным нулю, семафор переходит в занятое состояние;

- счетчик числа текущих ресурсов не может быть отрицательным;

- счетчик числа текущих ресурсов не может быть больше, чем счетчик максимального числа ресурсов.

Функция CreateSemaphore создает семафор.

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, // атр безопасности
    LONG lInitialCount,                  // начальное число ресурсов
    LONG lMaximumCount,                  // макс число ресурсов
    LPCTSTR lpName                       // имя объекта
);
```

В следующем примере кода создается анонимный семафор с максимальным числом ресурсов 5, числом свободных ресурсов 5:

```
HANDLE hSem = CreateSemaphore( NULL, 5, 5, NULL );
if (hSem == NULL) {
    // Ошибка создания
}
```

Если создан поименованный семафор, получить его описатель можно при помощи функции OpenSemaphore.

```
HANDLE OpenSemaphore(
    DWORD dwDesiredAccess,  // требуемый доступ
    BOOL bInheritHandle,    // возможность наследования описателя
    LPCTSTR lpName          // имя объекта
);
```

Поток получает доступ к ресурсу, вызывая одну из *wait*-функций.

Если счетчик текущего числа ресурсов семафора больше нуля (семафор свободен), *wait*-функция уменьшает счетчик текущего числа ресурсов на единицу и возвращает управление вызвавшему потоку. В противном случае *wait*-функция ожидает свободного состояния семафора, переводя вызывающий поток в блокированное состояние.

Поток освобождает ресурс вызовом функции ReleaseSemaphore.

```
BOOL ReleaseSemaphore(
    HANDLE hSemaphore,       // описатель семафора
    LONG lReleaseCount,      // число освобождаемых ресурсов
    LPLONG lpPreviousCount   // предыдущее состояние счетчика ресурсов
);
```

При этом система увеличивает счетчик числа текущих ресурсов на значение, заданное параметром lReleaseCount. Обычно это значение равно единице, но это не обязательно.

Получить значение счетчика числа текущих ресурсов семафора нельзя.

Мьютексы

Мьютексы гарантируют взаимоисключающий доступ к одному ресурсу. Объект ядра «мьютекс» содержит счетчик числа пользователей, счетчик рекурсии и переменную, содержащую идентификатор потока.

Использование мьютекса основано на принципе «обладания». Мьютекс свободен, если ни один поток не захватал его (не обладает им). Как только поток захватывает мьютекс, последний переходит в занятое состояние. Переменная объекта ядра «мьютекс», содержащая идентификатор потока, указывает на поток, который обладает мьютексом. Счетчик числа рекурсий показывает, сколько раз поток захватывал данный мьютекс.

Для мьютексов применяются следующие правила:

- если идентификатор потока равен нулю, мьютекс свободен;
- если идентификатор потока не равен нулю, мьютекс занят;

Мьютекс создается функцией CreateMutex.

```
HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,  // атр безопасности
    BOOL bInitialOwner,                       // начальный владелец
    LPCTSTR lpName                            // имя объекта
);
```

В следующем примере создается свободный анонимный мьютекс:

```
HANDLE hMutex = CreateMutex( NULL, FALSE, NULL );
```

Следующий пример показывает, как создать анонимный мьютекс, принадлежащий текущему потоку:

```
HANDLE hMutex = CreateMutex( NULL, TRUE, NULL );
```

Как и для других объектов, получить описатель существующего поименованного мьютекса можно при помощи функции OpenMutex.

```
HANDLE OpenMutex(
    DWORD dwDesiredAccess,  // требуемый доступ
    BOOL bInheritHandle,    // возможность наследования описателя
    LPCTSTR lpName          // имя объекта
);
```

Поток получает доступ к ресурсу, вызывая одну из *wait*-функций. Если мьютекс свободен, *wait*-функция записывает идентификатор вызывающего потока (мьютекс становится занятым) и немедленно возвращает управление в вызвавший поток. Если мьютекс занят, система блокирует вызывающий поток до момента освобождения мьютекса.

Поток освобождает мьютекс при помощи функции ReleaseMutex.

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

Если функция завершилась успешно, возвращается ненулевое значение.

Если функцию вызвал поток, не обладающий данным мьютексом, функция возвращает нулевое значение, а функция GetLastError возвращает значение ERROR_NOT_OWNER.

Если поток завершил свое выполнение, не освободив мьютекс, система автоматически переводит мьютекс в свободное состояние, а *wait*-функция возвращает значение WAIT_ABANDONED.

Асинхронный ввод-вывод

Функции WriteFile, ReadFile, ConnectNamedPipe, TransactNamedPipe и некоторые другие могут выполняться как в синхронном, так и в асинхронном режиме. При выполнении в синхронном режиме функции возвращают управление только после завершения запрошенной операции. В асинхронном режиме эти функции возвращают управление немедленно. При этом выполнение запрошенной операции выполняется параллельно с выполнением текущего потока.

Функции WriteFileEx и ReadFileEx выполняют операции только в асинхронном режиме. Асинхронный режим называется также overlapped (перекрывающийся).

Асинхронная операция инициализируется функцией CreateFile или CreateNamedPipe, заданной с флагом FILE_FLAG_OVERLAPPED.

Чтобы выполнять операцию асинхронно, поток должен передать в функцию создания файла указатель на инициализированную структуру OVERLAPPED.

```
typedef struct _OVERLAPPED {
    ULONG_PTR  Internal;      // зарезервировано для ОС
    ULONG_PTR  InternalHigh;  // зарезервировано для ОС
    DWORD  Offset;            // позиция файла в начале операции
    DWORD  OffsetHigh;        // позиция файла в начале операции
    HANDLE hEvent;            // описатель объекта ядра «событие»
} OVERLAPPED;
```

В следующем примере открывается именованный канал для выполнения асинхронных операций:

```
OVERLAPPED os;
LPCTSTR lpszPipeName = "\\\\.\\pipe\\pipe_one";

HANDLE hPipe = CreateNamedPipe(
    lpszPipeName,
    FILE_FLAG_OVERLAPPED | PIPE_ACCESS_DUPLEX,
    PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT,
    1,
    0,
    0,
    1000,
    NULL);

memset( & os, 0, sizeof( OVERLAPPED ) );
os.hEvent = CreateEvent( NULL, TRUE, FALSE, NULL );

ResetEvent( os.hEvent );
```

```
ConnectNamedPipe( hPipe, & os );
if ( GetLastError() == ERROR_IO_PENDING ) {
    DWORD dwWaitResult = WaitForSingleObject( os.hEvent, INFINITE );
    if ( dwWaitResult == WAIT_OBJECT_0 ) {
        // подключение к каналу
    } else {
        // ошибка или тайм-аут
    }
}
```

Когда вызывается функция асинхронного ввода-вывода, может оказаться, что выполнение операции завершится раньше, чем возврат из функции. Чтобы убедиться, что поток перешел в асинхронный режим, используется проверка текущей ошибки:

```
if ( GetLastError == ERROR_IO_PENDING ) {
    // операция выполняется асинхронно
} else {
    // операция завершена
}
```

Для ожидания завершения асинхронной операции используется также функция GetOverLappedResult.

```
BOOL GetOverlappedResult(
    HANDLE hFile,                       // описатель файла, канала
    LPOVERLAPPED lpOverlapped,          // структура OVERLAPPED
    LPDWORD lpNumberOfBytesTransferred, // число переданных байт
    BOOL bWait                          // признак ожидания завершения
);
```

Если параметр bWait равен TRUE, функция ожидает окончания операции, блокируя поток.

Если параметр bWait равен FALSE и операция не завершена, функция GetLastError возвращает значение ERROR_IO_INCOMPLETE.

Средства межпроцессного взаимодействия

Каналы

Канал (*pipe*) — это проводник информации (рисунок 4) между двумя процессами, работающими как на одной, так и на разных машинах в локальной сети. Приложение, создающее канал, называется сервером. Один сервер может работать одновременно с несколькими клиентами.

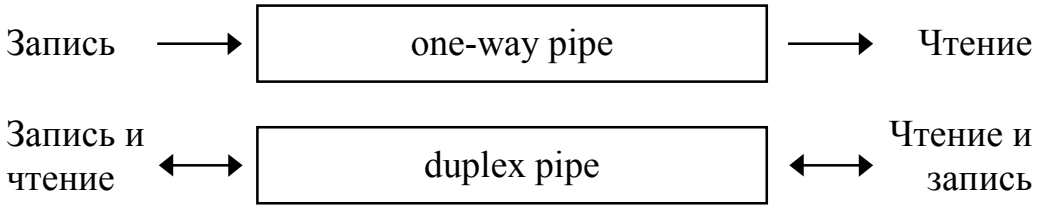

Рисунок 4 - Одно и двунаправленные каналы

Канал может быть создан анонимным (*anonymous pipe*) и поименованным (*named pipe*), одно (*one-way*) и двунаправленным (*duplex*), ориентированным на передачу потока байт и потока сообщений.

Имя канала задается в одной из следующих форм:

```
\\сервер\pipe\имя_канала
\\.\pipe\имя_канала
```

Первая форма предполагает указание сетевого адреса машины, на которой создается канал. Вторая форма используется, если канал создается на текущей машине.

Именованный канал создается функцией CreateNamedPipe.

```
HANDLE CreateNamedPipe(
    LPCTSTR lpName,                          // имя канала
    DWORD dwOpenMode,                        // режим открытия
    DWORD dwPipeMode,                        // режим канала
    DWORD nMaxInstances,                     // макс. копий канала
    DWORD nOutBufferSize,                    // размер вых буфера
    DWORD nInBufferSize,                     // размер вх буфера
    DWORD nDefaultTimeOut,                   // тайм-аут
    LPSECURITY_ATTRIBUTES lpSecurityAttributes  // атр безопасности
);
```

Режимы открытия:

PIPE_ACCESS_DUPLEX — создается двунаправленный канал.

PIPE_ACCESS_INBOUND — создается канал для передачи информации от клиента к серверу.

PIPE_ACCESS_OUTBOUND — создается канал для передачи информации от сервера к клиенту.

Дополнительно могут быть указаны флаги:

FILE_FLAG_OVERLAPPED — создается канал для асинхронной передачи.

FILE_FLAG_WRITE_THROUGH — функция записи в канал не возвращается до тех пор, пока операция не будет завершена.

Параметр nMaxInstances задает максимальное количество копий канала, которые могут быть созданы. При создании копий должно быть указано одинаковое количество копий.

Режимы канала:

PIPE_TYPE_BYTE — создается канал для передачи потока байт.

PIPE_TYPE_MESSAGE — создается канал для передачи потока сообщений.

Дополнительно при этом указывается режим чтения:

PIPE_READMODE_BYTE — данные читаются из канала как поток байт.

PIPE_READMODE_MESSAGE — данные читаются из канала как поток сообщений.

Дополнительно может быть задан режим ожидания данных:

PIPE_WAIT — функции чтения и записи в канал ждут завершения операции.

PIPE_NOWAIT — функции чтения и записи в канал возвращаются немедленно, если канал не готов.

Параметры nOutBufferSize и nInBufferSize задают размеры выходного и входного буферов канала для резервирования памяти.

Тайм-аут задается для использования в функции WaitNamedPipe.

После создания канала клиент должен подключиться к нему, используя функцию CreateFile или CallNamedPipe.

```
HANDLE CreateFile(
    LPCTSTR lpFileName,             // имя файла, канала, устройства
    DWORD dwDesiredAccess,                      // требуемый доступ
    DWORD dwShareMode,                          // режим разделения
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // атр безопасности
    DWORD dwCreationDisposition,                // режим создания
    DWORD dwFlagsAndAttributes,                 // атрибуты файла
    HANDLE hTemplateFile                        // описатель шаблона
);
```

Параметр «требуемый доступ» является комбинацией флагов:

GENERIC_READ — файл открывается для чтения.

GENERIC_WRITE — файл открывается для записи.

Параметр «режим разделения» определяет, как файл можно использовать для одновременного доступа нескольких потоков:

FILE_SHARE_READ — разрешено одновременное чтение.

FILE_SHARE_WRITE — разрешена одновременная запись.

Если этот параметр равен нулю, разделение файла запрещено.

Параметр «режим создания» определяет, что выполняет функция во время создания:

CREATE_NEW — создается новый файл. Если файл существует, функция возвращает признак ошибки.

CREATE_ALWAYS — новый файл создается в любом случае. Если файл существует, он перезаписывается.

OPEN_EXISTING — открывает существующий файл. Если файл не существует, функция возвращает признак ошибки.

OPEN_ALWAYS — файл открывается в любом случае. Если файл не существует, создается новый файл.

TRUNCATE_EXISTING — после открытия файла он усекается до нулевой длины. Если файла не существует, функция возвращает признак ошибки.

Параметр «атрибуты файла» является комбинацией флагов:

FILE_ATTRIBUTE_ARCHIVE — создается файл с атрибутом archive.

FILE_ATTRIBUTE_HIDDEN — создается файл с атрибутом hidden.

FILE_ATTRIBUTE_NORMAL — создается файл без атрибутов. Этот флаг нельзя использовать в комбинации с другими флагами.

FILE_ATTRIBUTE_READONLY — создается файл с атрибутом только для чтения.

FILE_ATTRIBUTE_SYSTEM — создается файл с атрибутом system.

FILE_ATTRIBUTE_TEMPORARY — создается временный файл.

```
BOOL CallNamedPipe(
    LPCTSTR lpNamedPipeName,  // имя канала
    LPVOID lpInBuffer,        // входной буфер
    DWORD nInBufferSize,      // размер входного буфера
    LPVOID lpOutBuffer,       // выходной буфер
    DWORD nOutBufferSize,     // размер выходного буфера
    LPDWORD lpBytesRead,      // число прочитанных байт
    DWORD nTimeOut            // тайм-аут
);
```

Входной буфер предназначен для задания данных, записываемых в канал, а выходной — для получения данных из канала.

Сервер использует функцию ConnectNamedPipe для того, чтобы определить, что клиент подключился к каналу.

```
BOOL ConnectNamedPipe(
    HANDLE hNamedPipe,          // описатель канала
    LPOVERLAPPED lpOverlapped   // указатель на структуру OVERLAPPED
);
```

После того, как произошло соединение сервера и клиента, они могут использовать функции ReadFile, WriteFile, ReadFileEx и WriteFileEx для выполнения операций чтения и записи в канал в синхронном и асинхронном режимах.

Функция TransactNamedPipe используется для выполнения транзакции записи и чтения именованного канала. Канал должен быть ориентирован на сообщения.

```
BOOL TransactNamedPipe(
    HANDLE hNamedPipe,          // имя канала
    LPVOID lpInBuffer,          // входной буфер (для записи в канал)
    DWORD nInBufferSize,        // размер входного буфера
    LPVOID lpOutBuffer,         // выходной буфер (для чтения)
    DWORD nOutBufferSize,       // размер выходного буфера
    LPDWORD lpBytesRead,        // число прочитанных байт
    LPOVERLAPPED lpOverlapped   // указатель на структуру OVERLAPPED
);
```

Для отключения от поименованного канала используется функция DisconnectNamedPipe.

```
BOOL DisconnectNamedPipe(
    HANDLE hNamedPipe   // описатель канала
);
```

Дополнительно может быть использована функция FlashFileBuffers для принудительного завершения операций чтения и записи.

```
BOOL FlushFileBuffers(
    HANDLE hFile  // описатель файла
);
```

Почтовые ящики

Почтовый ящик (*mailslot*) — это механизм однонаправленной передачи сообщений между процессами.

Принцип работы почтовых ящиков поясняет рисунок 5.

Любой поток, создающий потовый ящик, является сервером. Любой поток, читающий почтовый ящик сервера или записывающий в него, является клиентом.

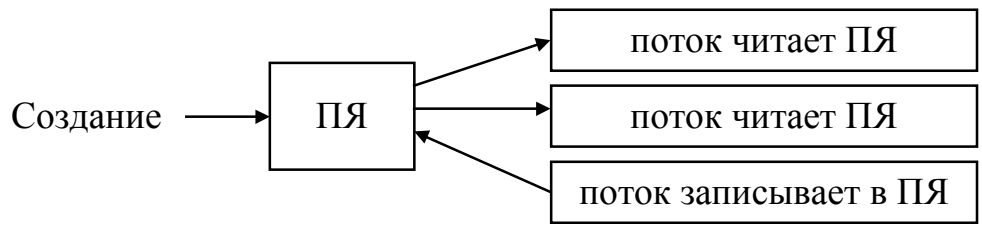

Рисунок 5 - Почтовые ящики

Почтовые ящики являются именованными объектами. Имя почтового ящика задается в одной из следующих форм:

```
\\.\mailslot\[путь]имя_ящика
\\сервер\mailslot\[путь]имя_ящика
\\домен\mailslot\[путь]имя_ящика
```

Первая форма используется для создания почтового ящика на текущей машине. Вторая форма описывает почтовый ящик на одной из машин сети. Третья форма описывает почтовый ящик домена.

Почтовый ящик создается при помощи функции CreateMailSlot.

```
HANDLE CreateMailslot(
    LPCTSTR lpName,                         // имя почтового ящика
    DWORD nMaxMessageSize,                  // макс. размер сообщения
    DWORD lReadTimeout,                     // тайм-аут чтения
    LPSECURITY_ATTRIBUTES lpSecurityAttributes // атр безопасности
);
```

Если параметр nMaxMessageSize равен нулю, размер сообщения не ограничен.

Пример создания почтового ящика:

```
LPCTSTR lpszSlotName = "\\\\.\\mailslot\\slot_one";
HANDLE hSlot = CreateMailSlot(
    lpszSlotName,
    0,
    MAILSLOT_WAUT_FOREVER,
    0
);
```

Клиент использует функцию CreateFile, чтобы подключиться к почтовому ящику, например:

```
HANDLE hSlot = CreateFile(
    lpszSlotName, // имя почтового ящика
    GENERIC_WRITE,
    FILE_SHARE_READ,
    0,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    0);
```

Сервер и клиент используют функции GetMailslotInfo и SetMailslotInfo для определения наличия сообщений и задания характеристик почтового ящика. С помощью функции GetMailslotInfo можно получить сведения о наличии сообщений в почтовом ящике.

```
BOOL GetMailslotInfo(
```

```
    HANDLE hMailslot,           // описатель почтового ящика
    LPDWORD lpMaxMessageSize,   // макс. размер сообщения
    LPDWORD lpNextSize,         // размер следующего сообщения
    LPDWORD lpMessageCount,     // количество сообщений в ящике
    LPDWORD lpReadTimeout       // тайм-аут чтения
);
```

С помощью функции SetMailslotInfo можно установить характеристики почтового ящика.

```
BOOL SetMailslotInfo(
    HANDLE hMailslot,    // описатель почтового ящика
    DWORD lReadTimeout   // тайм-аут чтения
);
```

Для выполнения операция чтения и записи в почтовый ящик используются функции ReadFile, ReadFileEx, WriteFile, WriteFileEx.

Проецируемые в память файлы

Проецируемые в память файлы используются для:

- загрузки и выполнения EXE и DLL файлов;
- совместного доступа к файлу данных;
- разделения данных между несколькими процессами, выполняемыми на одной машине.

Файл данных можно спроецировать на адресное пространство процесса для манипуляций с большими потоками данных. В качестве пример рассмотрим приложение, которое меняет порядок следования байтов файла данных на обратный. Методы решения этой задачи:

Метод 1: один файл, один буфер.

Считываем файл в буфер, меняем порядок следования и записываем буфер в существующий файл. Недостаток метода — операция выполняется медленно, невозможно обработать файл большого размера.

Метод 2: два файла, один буфер.

Конец файла считываем в буфер N Кб. Меняем байты местами и записываем буфер в новый файл. Недостаток метода — медленный, требует место на диске для второго файла.

Метод 3: один файл, два буфера.

Начало файла считывается в буфер N Кб, конец файла — в аналогичный буфер. Первый буфер записывается в конец файл, а второй — в начало. Недостатки метода — слишком сложный.

Метод 4: один файл, нет буферов.

Открываем файл, указывая системе зарезервировать регион виртуального адресного пространства. Сообщаем, что первый байт файла

нужно спроецировать на первый байт региона. Далее работаем с файлом, как с памятью. Например, если конец файла содержит нулевой байт, вызываем функцию _strrev. Плюс метода заключается в том, что всю работу по кэшированию файла выполняет система.

Порядок использования проецируемых в память файлов.

1) Создать или открыть объект ядра «файл», идентифицирующий файл на диске.

2) Создать объект ядра «проекция файла», чтобы сообщить системе размер файла и способ доступа к нему.

3) Указать системе, как спроецировать в адресное пространство процесса объект «проекция файла» — целиком или частично.

По окончании работы с проецируемым в память файлом нужно:

1) сообщить системе об отмене проецирования на адресное пространство процесса объекта «проекция файла»;

2) закрыть объект ядра «проекция файла»;

3) закрыть объект ядра «файл».

Создание или открытие объекта ядра «файл» выполняется при помощи функции CreateFile.

Создание объекта ядра «проекция файла» выполняется при помощи функции CreateFileMapping.

```
HANDLE CreateFileMapping(
    HANDLE hFile,                       // описатель файла
    LPSECURITY_ATTRIBUTES lpAttributes, // атрибуты безопасности
    DWORD flProtect,                    // защита проекции
    DWORD dwMaximumSizeHigh,            // макс. размер проекции файла
    DWORD dwMaximumSizeLow,             // макс. размер проекции файла
    LPCTSTR lpName                      // имя объекта
);
```

Параметр flProtect может принимать значения флагов:

PAGE_READONLY — разрешено только чтение региона.

PAGE_READWRITE — разрешены чтение и запись региона.

PAGE_WRITECOPY — создает копию записываемого региона.

Есть еще флаги:

SEC_NOCACHE — запрещает кэшировать страницы файла;

SEC_IMAGE — указывает, что файл является исполняемым (PE).

После создания объекта ядра «проекция файла» необходимо чтобы система, зарезервировав регион адресного пространства под данные файла, передала их как физическую память, отображенную на регион. Это делает функция MapViewOfFile:

```
LPVOID MapViewOfFile(
    HANDLE hFileMappingObject,   // описатель проекции
```

```
    DWORD dwDesiredAccess,        // режим доступа
    DWORD dwFileOffsetHigh,       // смещение (старшая часть)
    DWORD dwFileOffsetLow,        // смещение (младшая часть)
    SIZE_T dwNumberOfBytesToMap   // число проецируемых байт
);
```

Параметр dwDesiredAccess задается константами:

FILE_MAP_WRITE — файл можно читать и записывать;

FILE_MAP_READ — файл можно только читать;

FILE_MAP_ALL_ACCESS — файл можно читать и записывать;

FILE_MAP_COPY — файл можно читать и записывать, но запись приводит к созданию закрытой копии.

При помощи параметров 3 и 4 задается начальный байт проецирования, а при помощи параметра 5 — число проецируемых байт.

Функция MapViewOfFile возвращает базовый адрес региона памяти, в который спроецирован файл.

Функция UnmapViewOfFile отключает файл данных:

```
BOOL UnmapViewOfFile(LPCVOID lpBaseAddress);
```

Параметром UnmapViewOfFile функции является адрес, возвращаемый функцией MapViewOfFile.

В следующем примере проекция файла используется для инверсии содержимого файла. Инверсия используется здесь как пример обработки, на самом деле обработка файла может быть произвольной:

```
#include <windows.h>
void main() {
    HANDLE hFile = 0, hFileMapping = 0;
    DWORD dwSize = 0;
    TCHAR * pstrFile;
    // Открываем файл
    hFile = CreateFile(
        "C:\\test.txt",
        GENERIC_READ | GENERIC_WRITE,
        0,
        0,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        0);
    // Размер файла
    dwSize = GetFileSize(hFile, 0);
    // Создаем проекцию
    hFileMapping = CreateFileMapping(
        hFile,
        0,
        PAGE_READWRITE,
```

```
        0,
        dwSize,
        0);
    // Проецируем файл на память
    pstrFile = (TCHAR*)MapViewOfFile(
        hFileMapping,
        FILE_MAP_WRITE,
        0,
        0,
        dwSize);
    // Выполняем операции с памятью, которые записываются в файл
    pstrFile = _strrev((LPCVOID)pstrFile);
    // Отключаем проекцию от памяти
    UnmapViewOfFile(pstrFile);
    // Закрываем объекты ядра
    CloseHandle(hFileMapping);
    CloseHandle(hFile);
}
```

Сервисы

Приложение-сервис должно удовлетворять правилам SCM (Service Control Manager). Оно стартует автоматически при старте системы, вручную при помощи панели управления (приложение «Сервисы»), или программно при помощи специальных функций сервисов. Сервисы функционируют независимо от наличия пользователей в системе.

Драйвер-сервис должен удовлетворять протоколу драйвера устройства. Драйвер-сервис не взаимодействует с SCM.

SCM поддерживает базу данных установленных сервисов и драйвер-сервисов и определяет средства для управления ими.

Функции SCM используют:

- сервисы (service program) — программы, который содержат код одного или нескольких сервисов (сервисы NT);
- программы конфигурирования сервисов (*service configuration program*) — управляют базой данных сервисов (устанавливают и деинсталируют сервисы, изменяют их настройки).
- программы управления сервисами (*service control program*) — запускают и останавливают сервисы, посылают запросы к сервисам через SCM.

Типы сервисов

Сервис — это обычное консольное приложение, работающее в фоновом режиме и построенное по определенным правилам. Когда SCM стартует сервис, ожидается, что основная функция main немедленно вызывает функцию StartServiceCtrlDispatcher. Инициализация сервиса (или сервисов) осуществляется функцией с условным названием ServiceMain, которая является точкой входа в сервис.

Различают два типа сервисов:

SERVICE_WIN32_OWN_PROCESS — программа-сервис, содержащий код только одного сервиса;

SERVICE_WIN32_SHARE_PROCESS — программа-сервис, содержащий код нескольких сервисов.

При старте сервис типа SERVICE_WIN32_OWN_PROCESS должен немедленно вызвать функцию StartServiceCtrlDispatcher, а если сервис имеет тип SERVICE_WIN32_SHARE_PROCESS и требуется инициализация отдельных его сервисов, ее (инициализацию) можно выполнять до вызова функции StartServiceCtrlDispatcher только если инициализация длится менее 30 секунд. В противном случае инициализация выполняется параллельно с вызовом функции StartServiceCtrlDispatcher при помощи отдельного потока.

Регистрация диспетчера управления

Функция StartServiceCtrlDispatcher выполняет регистрацию диспетчера управления сервисом:

```
BOOL StartServiceCtrlDispatcher(
   CONST LPSERVICE_TABLE_ENTRY lpServiceTable   // service table
);
```

Функция StartServiceCtrlDispatcher использует следующую структуру для описания таблицы сервисов, при этом каждый отдельный сервис описывается отдельной записью:

```
typedef struct _SERVICE_TABLE_ENTRY {
    LPTSTR lpServiceName;
    LPSERVICE_MAIN_FUNCTION lpServiceProc;
} SERVICE_TABLE_ENTRY, *LPSERVICE_TABLE_ENTRY;
```

Если функция StartServiceCtrlDispatcher завершается успешно, вызывающий её поток не завершается до тех пор, пока не будут завершены все сервисы. SCM посылает запросы этому потоку, называемому диспетчером управления (*control dispatcher*) через именованный канал.

Для сервиса типа SERVICE_WIN32_OWN_PROCESS функция main может иметь следующий вид:

```
// прототип функции сервиса
VOID service_main( DWORD argc, LPTSTR *argv );

void main() {
    SERVICE_TABLE_ENTRY DispatchTable[] =  {
        { "MyService", service_main }, { 0, 0 }
    };
    if ( !StartServiceCtrlDispatcher( DispatchTable ) ) {
        // ошибка инициализации сервиса
    }
}
```

Как видно из приведенного кода, таблица, описывающая сервисы, завершается записью, содержащей нули.

Основная функция сервиса

Функция с условным названием ServiceMain является точкой входа в сервис. Функция сервиса должна выполнять следующие действия:

1. Вызвать функцию RegisterServiceCtrlHandler для регистрации диспетчера управления сервисом.

2. Выполнить инициализацию сервиса.

Если инициализация длится менее секунды, она может быть выполнена непосредственно в функции сервиса.

Если инициализация требует большего времени, функция сервиса должна установить состояние SERVICE_START_PENDING при помощи функции SetServiceStatus, используя структуру SERVICE_STATUS и указывая время ожидания следующей смены состояния (*wait hint*). По мере продвижения инициализации могут выполняться дополнительные вызовы SetServiceStatus для подтверждения нахождения сервиса в процессе инициализации, иначе SCM может принять решение об остановке сервиса.

3. После завершения инициализации функция сервиса должна установить состояние SERVICE_RUNNING, указывающее, что инициализация завершена и сервис перешел в рабочий режим.

4. Выполнить основную задачу сервиса или вернуться, если нет текущих задач для выполнения.

5. Если во время инициализации или выполнения сервиса возникла ошибка, необходимо установить состояние SERVICE_STOP_PENDING, выполнить очистку и установить состояние SERVICE_STOPPED.

Пример функции ServiceMain:

```
VOID WINAPI service_main(DWORD dwArgc, LPTSTR *lpszArgv) {
    // Регистрируем функцию управления service_ctrl
    shStatusHandle = RegisterServiceCtrlHandler(SERV_NAME),
service_ctrl);
    if (shStatusHandle) {
        // Структура SERVICE_STATUS
        sStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
        sStatus.dwServiceSpecificExitCode = 0;
        // Отсылаем состояние "инициализация"
        if (ReportStatusToSCM(SERVICE_START_PENDING, 0, WHINT)) {
            ServiceStart(dwArgc, lpszArgv);
        }
        // Пробуем остановить сервис
        (VOID)ReportStatusToSCM(SERVICE_STOPPED, dwError, 0);
    }
    return;
}
```

Здесь функция сервиса вызывает функцию ServiceStart, которая выполняет инициализацию сервиса, основную задачу и очистку.

Кроме того, посылка уведомлений о текущем состоянии осуществляется при помощи вспомогательной функции ReportStatusToSCM, примерный вид которой приведен ниже:

```
BOOL ReportStatusToSCM(DWORD dwCurrentState,
    DWORD dwWin32ExitCode, DWORD dwWaitHint)
```

```
{
    static DWORD dwCheckPoint = 1;
    if (dwCurrentState == SERVICE_START_PENDING) {
        sStatus.dwControlsAccepted = 0;
    } else {
        sStatus.dwControlsAccepted = SERVICE_ACCEPT_STOP;
    }
    sStatus.dwCurrentState = dwCurrentState;
    sStatus.dwWin32ExitCode = dwWin32ExitCode;
    sStatus.dwWaitHint = dwWaitHint;
    if ((dwCurrentState == SERVICE_RUNNING)
    || (dwCurrentState == SERVICE_STOPPED)) {
        sStatus.dwCheckPoint = 0;
    } else {
        sStatus.dwCheckPoint = dwCheckPoint++;
    }
    // Устанавливаем состояние сервиса
    return SetServiceStatus(shStatusHandle, &sStatus);
}
```

Используются следующие глобальные переменные:

```
SERVICE_STATUS sStatus;                 // Текущее состояние сервиса
SERVICE_STATUS_HANDLE shStatusHandle;   // Дескр текущего состояния
DWORD dwError = 0;                      // Ошибка
```

Функция управления сервисом

Сервис должен иметь функцию управления сервисом (*service control handler*), которая вызывается диспетчером управления (*control dispatcher*) при получении запроса на управление сервисом от программы управления.

Функция управления сервисом должна уведомлять SCM о текущем состоянии сервиса независимо от того, меняется оно, или нет. Функция также должна завершать свою работу в течении 30 секунд, в противном случае SCM выдаст ошибку.

Программа управления сервисом посылает запрос сервису при помощи функции ControlService. Все сервисы должны принимать код управления SERVICE_CONTROL_INTERROGATE, который уведомляет сервис о необходимости подтвердить текущее состояние. Другим кодом, который должна обрабатывать функция управления, является код остановки сервиса SERVICE_CONTROL_STOP.

Для остановки сервиса необходимо послать SCM уведомление о состоянии SERVICE_STOP_PENDING, иначе SCM выдаст ошибку 1053 (*The*

Service did not respond...). После этого должна быть выполнена очистка, а затем уведомление о состоянии SERVICE_STOPPED.

Пример функции управления сервисом:

```
VOID WINAPI service_ctrl(DWORD dwCtrlCode) {
    switch ( dwCtrlCode ) {
    case SERVICE_CONTROL_STOP:
        // Уведомляем о начале остановки сервиса
        ReportStatusToSCM(SERVICE_STOP_PENDING, NO_ERROR, 0);
        // Останавливаем сервис, выполняем очистку
        ServiceStop();
        return;
    case SERVICE_CONTROL_INTERROGATE:
        // Обновить состояние сервиса
        break;
    default:   // Недопустимый код управления
        break;
    }
    ReportStatusToSCMgr(sStatus.dwCurrentState, NO_ERROR, 0);
}
```

Здесь функция ServiceStop вызывает завершение функции сервиса ServiceStart, которая производит очистку, а затем основная функция сервиса service_main уведомляет о состоянии SERVICE_STOPPED.

Управление сервисами

Для использования сервиса он должен быть создан и установлен в базу данных SCM. Сделать это можно при помощи вспомогательной программы конфигурирования сервиса, либо при помощи самого сервиса, запускаемого с ключом установки (или удаления). Перед установкой сервиса нужно открыть SCM при помощи функции OpenSCManager:

```
SC_HANDLE OpenSCManager(
    LPCTSTR lpMachineName,   // имя компьютера, 0 - локальный
    LPCTSTR lpDatabaseName,  // имя базы данных SCM, 0 - по умолчанию
    DWORD dwDesiredAccess    // доступ - SC_MANAGER_ALL_ACCESS
);
```

Заметим, что здесь используется специальный тип SC_HANDLE.

Функция CreateService создает сервис и устанавливает его в базу данных SCM. Функция использует три разных имени сервиса: имя в виде идентификатора, имя, отображаемое в панели сервисов, и имя файла сервиса. Использование функций OpenSCManager и CreateService поясняет следующий пример:

```
void CustomServiceInstall() {
```

```
    SC_HANDLE hSRV = 0;
    SC_HANDLE hSCM = 0;
    if ( GetModuleFileName( NULL, szPath, 260 ) == 0 ) {
        return;
    }
    hSCM = OpenSCManager( NULL, NULL, SC_MANAGER_ALL_ACCESS );
    if ( !hSCM ) {
        // не удалось открыть SCM и подключиться к базе данных
    } else {
        hSRV = CreateService(
            hSCM,                  // база данных SCM
            SZ_SERVICE_NAME,           // имя сервиса
            SZ_SERVICE_DISPLAYNAME,       // отображаемое имя
            SERVICE_ALL_ACCESS,           // требуемый доступ
            SERVICE_WIN32_OWN_PROCESS,    // тип сервиса
            SERVICE_DEMAND_START,      // тип старта
            SERVICE_ERROR_NORMAL,      // тип управления ошибками
            szPath,                // путь к файлу сервера
            NULL,                  // нет порядка загрузки групп
            NULL,                  // нет тега
            NULL,                  // зависимости
            NULL,                  // счет LocalSystem account
            NULL);                 // нет пароля
        if ( !hSRV ) {
            // не удалось создать сервис
        } else {
            CloseServiceHandle(hSRV);
        }
        CloseServiceHandle(hSCM);
    }
}
```

Здесь используются константы для имени и отображаемого имени сервиса и переменная szPath, которая принимает значение пути к файлу сервиса при помощи функции GetModuleFileName.

Удаление сервиса из базы данных SCM производится при помощи функции DeleteService. Перед удалением сервиса он при необходимости должен быть остановлен. Следующая функция поясняет процесс удаления сервиса:

```
void CustomServiceRemove() {
    SC_HANDLE hSRV;
    SC_HANDLE hSCM = OpenSCManager( 0, 0, SC_MANAGER_ALL_ACCESS );
    if ( hSCM ) {
        // открываем сервис
        hSRV = OpenService(hSCM, SZ_SERVICE_NAME, SERVICE_ALL_ACCESS);
```

```
        if ( hSRV ) {
            // Пытаемся остановить сервис
            if(ControlService(hSRV, SERVICE_CONTROL_STOP, &sStatus)){
                Sleep( 1000 );
                while ( QueryServiceStatus( hSRV, &sStatus) ) {
                    if (sStatus.dwCurrentState == SERVICE_STOP_PENDING)
                        Sleep( 1000 );
                    else break;
                }
                if (sStatus.dwCurrentState == SERVICE_STOPPED) {
                    // сервис остановлен
                } else {
                    // не удалось остановить сервис
                }
            }
            // Удаляем сервис
            if ( DeleteService( hSRV ) ) {
                // сервис удален
            } else {
                // не удалось удалить сервис
            }
            CloseServiceHandle( hSRV );
        } else {
            // не удалось открыть сервис
        }
        CloseServiceHandle( hSCM );
    } else {
        // не удалось открыть SCM
    }
}
```

Здесь также используется функция открытия сервиса:

```
SC_HANDLE OpenService(
    SC_HANDLE hSCManager,  // описатель от OpenSCManager
    LPCTSTR lpServiceName, // имя сервиса
    DWORD dwDesiredAccess  // доступ - SC_MANAGER_ALL_ACCESS
);
```

Если эта функция завершилась неуспешно, а функция GetLastError возвращает при этом ERROR_SERVICE_DOES_NOT_EXIST, сервер не установлен в базе данных SCM.

Функция StartService запускает сервис:

```
BOOL StartService(
    SC_HANDLE hService,            // описатель hSRV
    DWORD dwNumServiceArgs,        // число аргументов
```

```
    LPCTSTR* lpServiceArgVectors   // массив аргументов
);
```

Пример функции, которая запускает сервис:

```
DWORD CustomServiceStart( SC_HANDLE hSRV ) {
    DWORD dwCheckPoints = 0;
    if ( !StartService( hSRV, 0, 0 ) ) {
        return GetLastError();
    }
    if ( !QueryServiceStatus( hSRV, & sStatus ) ) {
        // Не удалось проверить сервис
        return GetLastError();
    while ( sStatus.dwCurrentState != SERVICE_RUNNING ) {
        // число проверок
        if ( ++dwCheckPoints > 10 ) return sStatus.dwCurrentState;
        Sleep( 1000 );
    }
    // Сервис запущен.
}
```

Здесь предполагается, что старт сервиса ожидается в течение 10 секунд. В MSDN приведены другие способы проверки старта сервиса, основанные на контрольных точках.

Рекомендуемая литература

1. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Пер. с англ. — 4-е изд. — СПб.: Питер; Издательско-торговый дом «Русская Редакция», 2001. — 753 с.: ил.

2. MSDN, October 2001.

Владимир Вадимович Пономарев
Системное программирование
Учебно-методическое пособие

Отпечатано с готового оригинал-макета
Издательство ОТИ НИЯУ МИФИ, 2010
Тираж 40 экз.