Министерство науки и высшего образования Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

А.В. ГУНЬКО

ПРОГРАММИРОВАНИЕ (В СРЕДЕ WINDOWS)

Утверждено Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2019

УДК 004.451.9(075.8)
Г 948

Рецензенты:

канд. техн. наук, доцент *В.А. Астапчук*,
канд. техн. наук *Д.О. Романников*

Работа подготовлена на кафедре автоматики

Гунько А.В.

Г 948 Программирование (в среде Windows): учебное пособие / А.В. Гунько – Новосибирск: Изд-во НГТУ, 2019 – 155 с.

ISBN 978-5-7782-3890-9

В учебном пособии изложены основные сведения об интерфейсах прикладного программирования вообще и Win32 API в частности, описаны методы и средства разработки многозадачного и многопоточного программного обеспечения в операционных системах семейства Windows, а также средства межзадачной и межпоточной коммуникации: анонимные и именованные каналы, почтовые ящики, отображаемые на память файлы, события, семафоры, взаимные исключения.

Кроме того, кратко обсуждаются средства коммуникации процессов по сети, а также особенности взаимодействия приложений и системных служб.

Предназначено для студентов II курса, обучающихся по направлениям 27.03.04 «Управление в технических системах» и 09.03.01 «Информатика и вычислительная техника», также может быть полезно студентам других технических специальностей, связанных с разработкой многозадачного и многопоточного программного обеспечения в среде операционных систем семейства Windows.

УДК 004.451.9(075.8)

ISBN 978-5-7782-3890-9

1. ПРИНЦИПЫ ПОСТРОЕНИЯ ИНТЕРФЕЙСОВ ОПЕРАЦИОННЫХ СИСТЕМ

Операционная система (ОС) всегда выступает как интерфейс между аппаратурой компьютера и пользователем с его задачами. Под интерфейсами операционных систем следует понимать специальные интерфейсы системного и прикладного программирования, предназначенные для выполнения следующих задач [1].

1. Управление процессами, которое включает в себя следующий набор основных функций:

- запуск, приостанов и снятие задачи с выполнения;
- задание или изменение приоритета задачи;
- взаимодействие задач между собой (механизмы сигналов, семафоры, очереди, конвейеры, почтовые ящики);
- RPC (Remote Procedure Call) – удаленный вызов подпрограмм.

2. Управление памятью:

- запрос на выделение блока памяти;
- освобождение памяти;
- изменение параметров блока памяти (память может быть заблокирована процессом либо предоставлена в общий доступ);
- отображение файлов на память (имеется не во всех системах).

3. Управление вводом/выводом:

- запрос на управление виртуальными устройствами (управление вводом/выводом является привилегированной функцией ОС);
- файловые операции (запросы к системе управления файлами на создание, изменение и удаление данных, собранных в файлы).

Пользовательский интерфейс операционной системы реализуется с помощью специальных программных модулей, которые принимают его команды на соответствующем языке (возможно, с использованием графического интерфейса) и транслируют их в обычные вызовы в соответствии с основным интерфейсом системы. Обычно эти модули называют интерпретатором команд.

Получив от пользователя команду, такой модуль после лексического и синтаксического анализа либо сам выполняет действие, либо обращается к другим модулям ОС, используя механизм API (Application Program Interface, интерфейс прикладного программирования).

В случае графических интерфейсов (GUI, Graphic User Interface) указание курсором на объекты и щелчок (клик) или двойной щелчок по соответствующим клавишам приводит к каким-либо действиям – запуску программы, ассоциированной с указываемым объектом, выбору и/или активизации пунктов меню и т. д. Такая интерфейсная подсистема транслирует «команды» пользователя в обращения к ОС. Управление GUI – частный случай задачи управления вводом/выводом, не являющийся частью ядра операционной системы, хотя в ряде случаев разработчики ОС относят функции GUI к основному системному API.

Интерфейс прикладного программирования

Необходимо разделить термин API на следующие направления:

- API как интерфейс высокого уровня, принадлежащий к библиотекам RTL. RTL (Run Time Library) – библиотека времени выполнения; она включает в себя те стандартные подпрограммы, которые система программирования подставляет на этапе компиляции. В общем случае RTL включает в себя не только модули из системы программирования, но и модули самой ОС;
- API прикладных и системных программ, входящих в поставку операционной системы;
- прочие API.

Интерфейс прикладного программирования предназначен для использования прикладными программами системных ресурсов ОС и реализуемых ею функций. API описывает совокупность функций и процедур, принадлежащих ядру или надстройкам ОС. Представляет собой набор функций, предоставляемых системой программирования разработчику прикладной программы и ориентированных на организацию взаимодействия результирующей прикладной программы *с целевой вычислительной системой.* Целевая вычислительная система представляет собой совокупность программных и аппаратных средств, в окружении которых выполняется результирующая программа.

API включает в себя не только сами функции, но и соглашения об их использовании, которые регламентируются ОС, архитектурой целевой вычислительной системы и системой программирования.

Существует несколько вариантов реализации API:

- реализация на уровне ОС;
- реализация на уровне системы программирования;
- реализация на уровне внешней библиотеки процедур и функций.

Система программирования в каждом из этих вариантов предоставляет разработчику средства для подключения функций API к исходному коду программы и организации их вызовов. Объектный код функций API подключается к результирующей программе компоновщиком при необходимости.

Возможности API можно оценивать со следующих позиций:

- эффективность выполнения функций API – включает в себя скорость выполнения функций и объем вычислительных ресурсов, необходимых для их выполнения;
- широта предоставляемых возможностей;
- зависимость прикладной программы от архитектуры целевой вычислительной системы.

Реализация функций API на уровне ОС

За их выполнение ответственность несет ОС. Объектный код, выполняющий функции, либо непосредственно входит в состав ОС (или даже ядра ОС), либо поставляется в составе динамически загружаемых библиотек, разработанных для данной ОС. Система программирования ответственна за то, чтобы организовать интерфейс для вызова этого кода. Результирующая программа обращается непосредственно к ОС, поэтому достигается наибольшая эффективность выполнения функций API по сравнению с другими вариантами реализации API.

Недостаток организации API по такой схеме – полное отсутствие переносимости не только кода результирующей программы, но и кода исходной программы. Программа, созданная для одной архитектуры вычислительной системы, не сможет исполняться на другой вычислительной системе даже после того, как ее объектный код будет полностью перестроен. Зачастую система программирования не сможет выполнить перестроение исходного кода дли новой архитектуры вычислительной системы, поскольку многие функции API, ориентированные на определенную ОС, в новой архитектуре просто отсутствуют. Для переноса прикладной программы с одной вычислительной системы на другую требуется изменение исходного кода программы.

Пример API такого рода – набор функций, предоставляемых пользователю со стороны ОС типа Microsoft Windows – WinAPI (Windows-API) [2]. Даже внутри этого корпоративного API существует определенная несогласованность, которая несколько ограничивает переносимость программ между различными ОС типа Windows.

Еще один пример такого API – набор сервисных функций ОС типа MS-DOS, реализованный в виде набора подпрограмм обслуживания программных прерываний.

Реализация функций API на уровне системы программирования

Функции API предоставляются пользователю в виде библиотеки функций соответствующего языка программирования. Обычно речь идет о библиотеке времени исполнения – RTL. Система программирования предоставляет пользователю библиотеку соответствующего языка программирования и обеспечивает подключение к результирующей программе объектного кода, ответственного за выполнение этих функций.

Эффективность функций API в таком варианте будет несколько ниже, чем при непосредственном обращении к функциям ОС. Так происходит, поскольку для выполнения многих функций API библиотека RTL языка программирования все равно должна выполнять обращения к функциям ОС. Наличие всех необходимых вызовов и обращений к функциям ОС в объектном коде RTL обеспечивает система программирования.

Рассмотрим вызов в программе на языке С для ОС Windows функции по запросу 256 байт памяти

```
Unsigned char * ptr = malloc (256);
```

Из кода пользовательской программы будет осуществлен вызов библиотечной функции **malloc**, код которой расположен в RTL. Библиотека времени выполнения в данном случае реализует вызов **malloc** уже как вызов системной функции API **HeapAlloc:**

```
LPVOID HeapAlloc;
HANDLE hHeap; /* указатель на блок */
DWORD dwFlags; /* свойства блока */
DWORD dwBytes; /* размер блока */
```

Параметры выделяемого блока памяти в таком случае задаются системой программирования, пользователь лишен возможности задавать их напрямую. Но возможно использование функций API прямо в тексте программы, если подгрузить системную библиотеку Kernel32.dll:

```
unsigned char * ptr =
= (LPVOID) HeapAlloc( GetProcessHeap(), 0, 256);
```

Переносимость исходного кода программы при использовании RTL будет самой высокой, поскольку синтаксис и семантика всех функций строго регламентированы в стандарте соответствующего языка программирования. Они зависят от языка, а не от архитектуры целевой вычислительной системы. Поэтому для выполнения прикладной программы на новой платформе достаточно заново построить код результирующей программы с помощью соответствующей системы программирования. Для каждой платформы будет требоваться свой код RTL языка программирования.

Реализация функций API с помощью внешних библиотек

Внешние библиотеки предоставляются пользователю в виде библиотеки процедур и функций, созданной сторонним разработчиком.

Система программирования ответственна только за то, чтобы подключить объектный код библиотеки к результирующей программе. Причем внешняя библиотека может быть и динамически загружаемой (загружаемой во время выполнения программы).

Эффективность выполнения – самая низкая, потому что внешняя библиотека обращается как к функциям ОС, так и к функциям RTL языка программирования.

Требование к переносимости исходного кода только одно – используемая внешняя библиотека должна быть доступна в любой из архитектур вычислительных систем, на которые ориентирована прикладная программа. Это возможно, если используемая библиотека удовлетворяет какому-то принятому стандарту, а система программирования поддерживает этот стандарт.

Например, библиотеки, удовлетворяющие стандарту POSIX, доступны в большинстве систем программирования для языка С. И если прикладная программа использует только библиотеки этого стандарта, то ее исходный код будет переносимым.

Платформенно-независимый интерфейс POSIX

POSIX (Portable Operating System Interface for Computer Environments) – платформенно независимый системный интерфейс для компьютерного окружения. Это стандарт IEEE, описывающий системные интерфейсы для открытых операционных систем, в том числе оболочки, утилиты и инструментарии. Стандарт базируется на UNIX-системах, но допускает реализацию и в других ОС.

Этот стандарт подробно описывает VMS (Virtual Memory System, систему виртуальной памяти), многозадачность (MPE, Multiprocess Executing) и технологию переноса операционных систем (CTOS). Таким образом, на самом деле POSIX представляет собой множество стандартов, именуемых POSIX.1 – POSIX.12. Следует отметить, что POSIX.1 предполагает язык С как основной язык описания системных функций API.

На рис. 1 изображена типовая схема реализации строго соответствующего POSIX приложения.

Видно, что для взаимодействия с операционной системой программа использует только библиотеки POSIX.1 и стандартную библиотеку RTL языка С, в которой возможно использование лишь 110 различных функций, также описанных стандартом POSIX.1.

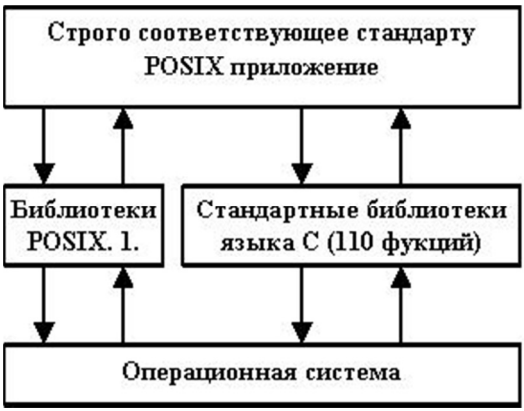

Рис. 1. Приложения, строго соответствующие стандарту POSIX

Реализации POSIX API на уровне операционной системы различны. Если UNIX-системы изначально соответствуют спецификациям IEEE-

Standard 1003.1–1990, то WinAPI не является POSIX-совместимым. Для его поддержки в MS Windows NT введен специальный модуль поддержки POSIX API, работающий на уровне привилегий пользовательских процессов. Он обеспечивает конвертацию и передачу вызовов из пользовательской программы к ядру системы и обратно, работая с ядром через WinAPI. Приложения, написанные с использованием WinAPI, могут передавать информацию POSIX-приложениям через стандартные механизмы потоков ввода/вывода (**`stdin, stdout`**).

Вопросы для самопроверки

1. Перечислите основные задачи, решаемые программными интерфейсами операционных систем.
2. Для чего предназначен и что включает в себя интерфейс прикладного программирования (API)?
3. Какие существуют варианты реализаций API, каковы критерии их сравнения?
4. Каковы особенности реализаций функций API на уровне ОС?
5. Каковы особенности реализаций функций API на уровне системы программирования?
6. Каковы особенности реализаций функций API с помощью внешних библиотек?

2. ОС WINDOWS И WINDOWS API

Назначение операционной системы (ОС)

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

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

Основные возможности операционных систем семейства Windows

ОС семейства Windows (начиная с версии NT 5.0) обеспечивает доступность базовых ресурсов ОС в столь непохожих друг на друга системах, как мобильные телефоны, карманные устройства, переносные компьютеры и серверы масштаба предприятия. Возможности ОС легко охарактеризовать, рассмотрев наиболее важные ресурсы [2], которыми управляют современные операционные системы.

- **Память.** ОС управляет сплошным, или плоским (flat), виртуальным адресным пространством большого объема, перемещая данные между физической памятью и диском или иным накопительным устройством прозрачным для пользователя образом.
- **Файловые системы.** ОС управляет пространством именованных файлов, предоставляя возможности прямого и последовательного доступа к файлам, а также средства управления файлами и каталогами. Используемые в большинстве систем пространства имен являются иерархическими.
- **Именование и расположение ресурсов.** Файлы могут иметь длинные описательные имена, причем принятая схема именования распространяется на такие объекты, как устройства, а также объекты синхронизации или межпроцессного взаимодействия. Размещение именованных объектов и управление доступом к ним тоже являются прерогативой ОС.
- **Многозадачность.** ОС должна располагать средствами управления процессами, потоками и другими единицами, способными независимо выполняться в асинхронном режиме. Задачи могут планироваться и вытесняться в соответствии с динамически определяемыми приоритетами.
- **Взаимодействие и синхронизация.** ОС управляет обменом информацией между задачами и их синхронизацией в изолированных системах, а также взаимодействием сетевых систем между собой и сетью Интернет.
- **Безопасность и защита.** ОС должна предоставлять гибкие механизмы защиты ресурсов от несанкционированного или непреднамеренного доступа и нанесения ущерба системе.

Microsoft Windows Win32/Win64 API обеспечивает поддержку не только этих, но и множества других средств ОС, и делает их доступными в ряде версий Windows, некоторые из которых постепенно выходят из употребления, а некоторые поддерживает лишь то или иное подмножество полного API.

Интерфейс программирования приложений Win32 API

Интерфейс программирования приложений в среде 32-разрядных ОС Windows (Win32 API) представляет собой набор функций и классов, которые используются для программирования приложений, работающих под управлением соответствующих операционных систем фирмы Microsoft. Ссылки на Win64 будут делаться лишь в тех случаях, когда различия между этими интерфейсами будут иметь существенное значение.

Функционально Win32 API подразделяется на следующие категории:

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

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

В курсе системного программирования главным образом изучается назначение и использование функций из категорий Base Services. Функции из категорий Common Control Library, Graphics Device

Interface и User Interface используются для разработки интерфейса приложений и требуют знания объектно-ориентированного подхода к программированию и языка C++.

Принципы, лежащие в основе Win32 API

В Win32 API имеется множество как самых незаметных, так и значительных отличий от других API, таких как POSIX API, с которым знакомы программисты, работающие в UNIX и Linux [3]. И хотя с применением Win32 API не связаны какие-либо специфические трудности в работе, она потребует внесения некоторых изменений в привычные стиль и методику программирования.

Ниже описаны некоторые из важнейших характеристик Win32 API, которые будут детальнее рассмотрены в дальнейшем.

Многие системные ресурсы Windows представляются в виде *объектов ядра* (kernel objects), для идентификации и обращения к которым используются *дескрипторы* (handles). По смыслу эти дескрипторы аналогичны дескрипторам (descriptors) файлов и идентификаторам (ID) процессов в UNIX.

Любые манипуляции с объектами ядра осуществляются только с использованием Windows API. «Лазеек» для обхода этого правила нет. Подобная организация работы согласуется с принципами абстрагирования данных, используемыми в объектно-ориентированном программировании, хотя сама система Windows объектно-ориентированной не является.

К объектам относятся файлы, процессы, потоки, каналы межпроцессного взаимодействия, объекты отображения файлов, события и многое другое. Объекты имеют атрибуты защиты.

Win32 API – богатый возможностями и гибкий интерфейс. Во-первых, одни и те же или аналогичные задачи могут решаться с помощью сразу нескольких функций; так, имеются вспомогательные функции (convenience functions), полученные объединением часто встречающихся последовательностей функциональных вызовов в одну функцию (к числу подобных функций принадлежит и функция **`CopyFile`**, используемая в одном из примеров далее). Во-вторых, функции часто имеют многочисленные параметры и флаги, многие из которых обычно игнорируются.

Windows предлагает большое количество механизмов синхронизации и взаимодействия, обеспечивающих удовлетворение самых разнообразных запросов.

Базовой единицей выполнения в Windows является поток (thread). В одном процессе (process) могут выполняться один или несколько потоков.

Для функций Windows используются длинные описательные имена. Приведенные ниже в качестве примера имена функций иллюстрируют не только соглашения об использовании имен, но и многоликость функций Windows:

```
WaitForSingleObject
WaitForMultipleObjects
WaitNamedPipe
```

Существует также несколько соглашений, регулирующих порядок использования имен типов.

- Имена предопределенных типов данных, необходимых API, также являются описательными, в них должны использоваться прописные буквы. К числу наиболее распространенных относятся следующие типы данных:

BOOL – определен как 32-битовый объект, предназначенный для хранения одного логического значения;

HANDLE – дескриптор объекта, представляет собой запись в таблице, которая поддерживается системой и содержит адрес объекта и средства для идентификации типа объекта;

DWORD – 32-битовое целое без знака;

LPTSTR (указатель на строку, состоящую из 8- или 16-битовых символов);

LPSECURITY_ATTRIBUTES – указатель на структуру атрибутов безопасности.

- В именах предопределенных типов указателей операция «*» не используется, они отражают дополнительные отличия между указателями различного типа, как, например, в случае типов **LPTSTR** (определен как **TCHAR ***) и **LPCTSTR** (определен как **const TCHAR ***). Тип **TCHAR** может обозначать как обычный символьный тип **char**, так и 2-байтовый тип **wchar_t**.

В отношении использования имен переменных (по крайней мере, в прототипах функций) также имеются определенные соглашения. Так, имя **lpszFileName** соответствует «длинному указателю на строку, завершающуюся нулевым символом», которая содержит имя файла. Этот пример иллюстрирует применение так называемой «венгерской

нотации». Точно так же **dwAccess** – двойное слово (32 бита), содержащее флаги прав доступа к файлу, где «dw» означает «double word» – «двойное слово».

Сравнение функций стандартной библиотеки C и функций Win32 API

Для примера рассмотрим ряд коротких программ, реализующих простое последовательное копирование содержимого файла тремя различными способами:

1) с использованием библиотеки C;

2) с использованием Windows;

3) с использованием вспомогательной функции Windows – **CopyFile**.

Кроме того, что эти примеры дают возможность сопоставить между собой различные модели программирования, они также демонстрируют возможности и ограничения, присущие библиотеке C и Windows.

Последовательная обработка файлов является простейшей, наиболее распространенной и самой важной из возможностей, обеспечиваемых любой операционной системой, и почти в каждой большой программе хотя бы несколько файлов обязательно подвергается этому виду обработки. Поэтому простая программа обработки файлов предоставляет прекрасную возможность ознакомиться с Windows и принятыми в ней соглашениями.

Само по себе копирование файлов не представляет особого интереса, однако сравнение программ не только позволит вам быстро оценить, чем отличаются друг от друга различные системы, но и послужит хорошим предлогом для знакомства с Windows. В последующих примерах реализуется ограниченный вариант одной из команд UNIX – **cp**, осуществляющей копирование одного файла в другой и требующей задания имен файлов в командной строке. В приведенных программах организована лишь простейшая проверка ошибок, которые могут возникать на стадии выполнения, а существующие файлы просто перезаписываются.

Копирование файла с использованием стандартной библиотеки языка С

Как видно из текста программы 1, стандартная библиотека С поддерживает объекты потоков ввода/вывода `FILE`, которые напоминают, несмотря на меньшую общность, объекты Windows **`HANDLE`**, представленные в программе 2:

Программа 1. Копирование файлов с использованием библиотеки С

```
/* Программа копирования файлов cpc. Реализация, использующая библиотеку С.*/
/* cpc файл1 файл2: Копировать файл1 в файл2. */
# include <stdio.h>
# include <errno.h>
# define BUF_SIZE 256
int main (int argc, char *argv [])
{
FILE *in_file, *out_file;
char rec [BUF_SIZE];
size_t bytes_in, bytes_out;
if (argc != 3) {
printf ("Использование: cpc файл1 файл2\n");
return 1;
}
in_file = fopen (argv [1], "rb");
if (in_file == NULL) {
perror (argv [1]);
return 2;
}
out_file = fopen (argv [2], "wb"); if (out_file == NULL)
{
perror (argv [2]);
return 3;
}
/* Обработать входной файл по одной записи за один
раз.*/
while((bytes_in = fread (rec, 1, BUF_SIZE, in_file))>0)
{
```

```
bytes_out = fwrite (rec, 1, bytes_in, out_file);
if (bytes_out != bytes_in) {
perror ("Неустранимая ошибка записи.");
return 4;
   }
}
fclose (in_file);
fclose (out_file);
return 0;
}
```

Исходный текст этого примера доступен на сайте дисциплины [4] под именем *cpc.cpp.*

Этот простой пример может служить наглядной иллюстрацией ряда общепринятых допущений и соглашений программирования, которые не всегда применяются в Windows.

1. Объекты открытых файлов идентифицируются указателями на структуры **FILE**. Указателю **NULL** соответствует несуществующий объект. По сути, указатели являются разновидностью дескрипторов объектов открытых файлов.

2. В вызове функции **fopen** указывается, каким образом должен обрабатываться файл – как текстовый или как двоичный. В текстовых файлах содержатся специфические для каждой системы последовательности символов, используемых, например, для обозначения конца строки. Во многих системах, включая Windows, в процессе выполнения операций ввода/вывода каждая из таких последовательностей автоматически преобразуется в нулевой символ, который интерпретируется в языке С как метка конца строки, и наоборот. В данном примере оба файла открываются как двоичные.

3. Диагностика ошибок реализуется с помощью функции **perror**, которая, в свою очередь, получает информацию относительно природы сбоя, возникающего при вызове функции **fopen**, из глобальной переменной **errno**. Вместо этого можно было бы воспользоваться функцией **ferror**, возвращающей код ошибки, ассоциированный не с системой, а с объектом **FILE**.

4. Функции **fread** и **fwrite** возвращают количество обработанных байтов непосредственно, а не через аргумент, что оказывает существенное влияние на логику организации программы. Неотрицательное

возвращаемое значение говорит об успешном выполнении операции чтения, тогда как нулевое – о попытке чтения метки конца файла.

5. Функция **fclose** может применяться лишь к объектам типа **FILE** (аналогичное утверждение справедливо и в отношении дескрипторов файлов UNIX).

6. Операции ввода/вывода осуществляются в синхронном режиме, т. е. прежде чем программа сможет выполняться дальше, она должна дождаться завершения операции ввода/вывода.

7. Для вывода сообщений об ошибках удобно использовать входящую в библиотеку C функцию ввода/вывода **printf**, которая также будет использована в первом примере Windows-программы.

Преимуществом реализации, использующей библиотеку C, является ее переносимость на платформы UNIX, Windows, а также другие системы, которые поддерживают стандарт ANSI C. Кроме того, в том, что касается производительности, вариант, использующий функции ввода/вывода библиотеки C, ничуть не уступает другим вариантам реализации.

Как и их эквиваленты в UNIX, программы, основанные на функциях для работы с файлами, входящих в библиотеку C, способны выполнять операции произвольного доступа к файлам (с использованием функции **fseek** или, в случае текстовых файлов, функций **fsetpos** и **fgetpos**), но это является уже потолком сложности для функций ввода/вывода стандартной библиотеки C, выше которого они подняться не могут. Вместе с тем Visual C++ предоставляет нестандартные расширения, способные, например, поддерживать блокирование файлов. Наконец, библиотека C не позволяет управлять средствами защиты файлов.

Резюмируя, можно сделать вывод, что если простой синхронный файловый или консольный ввод/вывод – это все, что вам надо, то для написания переносимых программ, которые будут выполняться под управлением UNIX или Windows, следует использовать библиотеку C.

Копирование файла с использованием Windows

В программе 2 решается та же задача копирования файлов, но делается это с помощью WinAPI:

Программа 2. Копирование файлов с использованием Windows, первая реализация

```
/* Программа копирования файлов cpw. Реализация, использующая Windows.*/
/* cpw файл1 файл2: Копировать файл1 в файл2.*/
#include <windows.h>
#include <stdio.h>
#define BUF_SIZE 256
int main(int argc, LPTSTR argv[]) {
     HANDLE hIn, hOut;
     DWORD nIn, nOut;
     CHAR Buffer [BUF_SIZE];
     if(argc != 3) {
       printf("Использование: cpw файл1 файл2\n");
       return  1;
     }
     hIn = CreateFile(argv[1], GENERIC_READ, 0,
     NULL, OPEN_EXISTING, 0, NULL);
     if (hIn==INVALID_HANDLE_VALUE) {
       printf("Невозможно открыть входной
       файл.Ошибка: %x\n", GetLastError ());
       return  2;
     }
     hOut = CreateFile (argv[2], GENERIC_WRITE, 0,
     NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL,
     NULL);
     if(hOut==INVALID_HANDLE_VALUE) {
       printf("Невозможно открыть выходной файл.
       Ошибка: %x\n", GetLastError ());
       return 3;
     }
     while(ReadFile(hIn, Buffer, BUF_SIZE, &nIn,
     NULL)&&nIn > 0) {
       WriteFile (hOut, Buffer, nIn, &nOut, NULL);
```

```
        if (nIn != nOut) {
          printf("Неустранимая ошибка записи: %x\n",
          GetLastError ());
          return 4;
        }
      }
      CloseHandle (hIn);
      CloseHandle (hOut);
      return 0;
}
```

Исходный текст этого примера доступен на сайте дисциплины под именем *cpw.cpp* и иллюстрирует некоторые особенности программирования в среде Windows.

1. В программу всегда включается файл **<windows.h>**, в котором содержатся все необходимые определения функций и типов данных Windows.

2. Все объекты Windows идентифицируются переменными типа **Handle**, причем для большинства объектов можно использовать одну и ту же общую функцию **CloseHandle**.

3. Рекомендуется закрывать все ранее открытые дескрипторы, если необходимость в них отпала, чтобы освободить ресурсы. В то же время при завершении процессов относящиеся к ним дескрипторы автоматически закрываются ОС, и если не остается ни одного дескриптора, ссылающегося на какой-либо объект, то ОС уничтожает этот объект и освобождает соответствующие ресурсы. Как правило, файлы подобным способом не уничтожаются.

4. Windows определяет многочисленные символические константы и флаги. Обычно они имеют длинные имена, нередко поясняющие назначение данного объекта. В качестве типичного примера можно привести имена **INVALID_HANDLE_VALUE** и **GENERIC_READ**.

5. Функции **ReadFile** и **WriteFile** возвращают булевы значения, а не количества обработанных байтов, для передачи которых используются аргументы функций. Это определенным образом изменяет логику организации работы циклов. Нулевое значение счетчика байтов указывает на попытку чтения метки конца файла и не считается ошибкой.

6. Функция **GetLastError** позволяет получать в любой точке программы коды системных ошибок, представляемые значениями типа

DWORD. В программе 2 показано, как организовать вывод генерируемых Windows текстовых сообщений об ошибках.

7. Windows NT (и выше) обладает более мощной системой защиты файлов, чем предшествующие ей версии. В данном примере защита выходного файла не обеспечивается.

8. Такие функции, как **CreateFile**, обладают богатым набором дополнительных параметров, но в данном примере использованы значения по умолчанию.

Копирование файла с использованием вспомогательной функции Windows

Для повышения удобства работы в Windows предусмотрено множество вспомогательных функций (convenience functions), которые, объединяя в себе несколько других функций, обеспечивают выполнение часто встречающихся задач программирования. В некоторых случаях использование этих функций может приводить к повышению производительности. Например, благодаря применению функции **CopyFile** значительно упрощается программа копирования файлов (программа 3). Помимо всего прочего, это избавляет от необходимости заботиться о буфере, размер которого в двух предыдущих программах произвольно устанавливался равным 256:

Программа 3. Копирование файлов с использованием вспомогательной функции Windows

```
/* Программа копирования файлов cpCF. Реализация,
в которой для повышения удобства использования и произ-
водительности программы используется функция Windows
CopyFile. */
/* cpCF файл1 файл2: Копировать файл1 в файл2. */
#include<windows.h>
#include <stdio.h>
int main (int argc, LPTSTR argv[]) {
if (argc != 3) {
printf ("Использование: cpCF файл1 файл2\n");
return 1;
}
```

```
if(!CopyFile (argv[1], argv[2], FALSE))  {
printf("Ошибка при выполнении функции CopyFile: %x\n",
GetLastError ());
return 2;
}
return  0;
}
```

Вопросы для самопроверки

1. Перечислите основные возможности ОС семейства Windows.
2. Перечислите основные категории функций, входящих в Win32 API.
3. Перечислите основные принципы, лежащие в основе Win32 API.
4. Перечислите основные особенности программирования с применением Win32 API (на примере задачи копирования файла).

3. ФАЙЛОВЫЕ ОПЕРАЦИИ И ОТОБРАЖАЕМЫЕ НА ПАМЯТЬ ФАЙЛЫ

3.1. Файловые операции

Операции открытия, чтения, записи и закрытия файлов

Прежде всего, приложение должно открыть файл при помощи функции **CreateFile**. Ниже приведем прототип этой функции:

```
HANDLE CreateFile(
  LPCTSTR lpFileName,      // адрес строки имени
                           // файла
  DWORD   dwDesiredAccess,    // режим доступа
  DWORD   dwShareMode,      // режим совместного
                              // использования файла
LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                              // дескриптор защиты
DWORD  dwCreationDistribution,     // параметры
                                   // создания
DWORD  dwFlagsAndAttributes,     // атрибуты файла
  HANDLE hTemplateFile);         // идентификатор
                                 // файла
                                 // с атрибутами
```

Через параметр **lpFileName** этой функции передается адрес текстовой строки с завершающим нулевым символом, содержащей путь и имя файла, канала или любого другого именованного объекта, который необходимо открыть или создать. Допустимое количество символов при указании путей доступа обычно ограничивается значением **MAX_PATH (260).**

С помощью параметра **dwDesiredAccess** следует указать нужный вид доступа. Если файл будет открыт только для чтения, в этом параметре необходимо указать флаг **GENERIC_READ**. Если необходимо выполнять над файлом операции чтения и записи, следует указать логическую комбинацию флагов **GENERIC_READ** и **GENERIC_WRITE**. В том случае, когда будет указан только флаг **GENERIC_WRITE**, операция чтения из файла будет запрещена.

Если файл будет использоваться одновременно несколькими процессами, через параметр **dwShareMode** необходимо передать режимы совместного использования файла: **FILE_SHARE_READ** или **FILE_SHARE_WRITE**.

Через параметр **lpSecurityAttributes** необходимо передать указатель на дескриптор защиты или значение **NULL**, если этот дескриптор не используется.

Параметр **dwCreationDistribution** определяет действия, выполняемые функцией **CreateFile**, если приложение пытается создать файл, который уже существует. Для этого параметра можно указать одну из следующих констант (табл. 1).

Таблица 1

Параметры создания/открытия файла

Константа	Описание
CREATE_NEW	Если создаваемый файл уже существует, функция **CreateFile** возвращает код ошибки
CREATE_ALWAYS	Существующий файл перезаписывается, при этом содержимое старого файла теряется
OPEN_EXISTING	Открывается существующий файл. Если файл с указанным именем не существует, функция **CreateFile** возвращает код ошибки
OPEN_ALWAYS	Если указанный файл существует, он открывается. Если файл не существует, он будет создан
TRUNCATE_EXISTING	Если файл существует, он открывается, после чего длина файла устанавливается равной нулю. Содержимое старого файла теряется. Если же файл не существует, функция **CreateFile** возвращает код ошибки

Параметр **dwFlagsAndAttributes** задает атрибуты и флаги для файла. Всего имеется 16 флагов и атрибутов. Атрибуты являются характеристиками файла, а не открытого дескриптора, и игнорируются, если открывается существующий файл.

При этом можно использовать любые логические комбинации следующих атрибутов (табл. 2), кроме атрибута **FILE_ATTRIBUTE_NORMAL**, который можно использовать только отдельно.

Таблица 2

Атрибуты создания/открытия файла

Атрибут	Описание
FILE_ATTRIBUTE_ ARCHIVE	Файл был архивирован (выгружен)
FILE_ATTRIBUTE_ COMPRESSED	Файл, имеющий этот атрибут, динамически сжимается при записи и восстанавливается при чтении. Если этот атрибут имеет каталог, то для всех расположенных в нем файлов и каталогов также выполняется динамическое сжатие данных
FILE_ATTRIBUTE_ HIDDEN	Скрытый файл
FILE_ATTRIBUTE_ NORMAL	Остальные перечисленные в этом списке атрибуты не установлены
FILE_ATTRIBUTE_ READONLY	Файл можно только читать
FILE_ATTRIBUTE_ SYSTEM	Файл является частью операционной системы

Кроме того, существует несколько флагов, позволяющих уточнить способ обработки файла и облегчить реализации Windows оптимизацию производительности и обеспечение целостности файлов. Часть из них описана в табл. 3.

И, наконец, последний параметр, **hTemplateFile**, предназначен для доступа к файлу шаблона с расширенными атрибутами для создаваемого файла. Этот параметр здесь не рассматривается.

В случае успешного завершения функция **CreateFile** возвращает идентификатор открытого файла. При ошибке возвращается значение **INVALID_HANDLE_VALUE**. Здесь все как обычно, пока никакого отображения еще не выполняется.

Таблица 3

Флаги создания/открытия файла

Флаг	Описание
FILE_FLAG_DELETE_ ON_CLOSE	Файл будет удален сразу же после закрытия последнего из его открытых дескрипторов
FILE_FLAG_WRITE_ THROUGH	Устанавливает режим сквозной записи промежуточных данных непосредственно в файл на диске, минуя кэш
FILE_FLAG_NO_ BUFFERING	Устанавливает режим отсутствия промежуточной буферизации или кэширования, при котором обмен данными происходит непосредственно с буферами данных программы, указанными при вызове функций **ReadFile** или **WriteFile**
FILE_FLAG_RANDOM_ ACCESS	Предполагается открытие файла для произвольного доступа; Windows будет пытаться оптимизировать кэширование файла применительно к этому виду доступа
FILE_FLAG_SEQUENTI AL_SCAN	Предполагается открытие файла для последовательного доступа; Windows будет пытаться оптимизировать кэширование файла применительно к этому виду доступа

Заметим, что для одного файла могут быть одновременно открыты несколько дескрипторов, если только это разрешается атрибутами совместного доступа и защиты файла. Открытые дескрипторы могут принадлежать одному и тому же или различным процессам.

Для закрытия объектов любого типа, объявления недействительными их дескрипторов и освобождения системных ресурсов почти во всех случаях используется одна и та же универсальная функция

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

Единственный параметр **hObject** – дескриптор объекта любого типа. Возвращаемое значение: в случае успешного выполнения функции – **TRUE**, иначе – **FALSE**.

Закрытие дескриптора сопровождается уменьшением на единицу счетчика ссылок на объект, что делает возможным удаление таких не хранимых постоянно (nonpersistent) объектов, как временные файлы или события. При выходе из программы система автоматически закрывает все открытые дескрипторы, однако все же лучше, чтобы программа

самостоятельно закрывала свои дескрипторы перед тем как завершить работу.

Попытки закрытия недействительных дескрипторов или повторного закрытия одного и того же дескриптора приводят к исключениям.

Чтение из файла производится функцией

```
BOOL ReadFile (HANDLE hFile, LPVOID lpBuffer,
DWORD nNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead,
LPOVERLAPPED lpOverlapped);
```

Здесь параметр **hFile** – дескриптор считываемого файла, который должен быть создан с правами доступа **GENERIC_READ**. Параметр **lpBuffer** является указателем на буфер в памяти, куда помещаются считываемые данные. Параметр **nNumberOfBytesToRead** – количество байтов, которые должны быть считаны из файла. Параметр l**pNumberOfBytesRead** – указатель на переменную, предназначенную для хранения числа байтов, которые были фактически считаны в результате вызова функции **ReadFile**. Этот параметр может принимать нулевое значение, если перед выполнением чтения указатель файла был позиционирован в конце файла или если во время чтения возникли ошибки, а также после чтения из именованного канала, работающего в режиме обмена сообщениями (работа с каналами описана далее), если переданное сообщение имеет нулевую длину.

Параметр **lpOverlapped** – указатель на структуру **OVERLAPPED**. Используется для организации асинхронного режима чтения (записи). Если запись выполняется синхронно, в качестве этого параметра следует указать значение **NULL**. Для использования асинхронного режима файл должен быть открыт функцией **CreateFile** с использованием флага **FILE_FLAG_OVERLAPPED**. Если указан этот флаг, параметр **lpOverlapped** не может иметь значение **NULL**. Он обязательно должен содержать адрес подготовленной структуры типа **OVERLAPPED**.

Возвращаемое функцией значение в случае успешного выполнения (которое считается таковым, даже если не был считан ни один байт из-за попытки чтения с выходом за пределы файла) – **TRUE**, иначе – **FALSE**.

Если значения дескриптора файла или иных параметров, используемых при вызове функции, оказались недействительными, возникает ошибка и функция возвращает значение **FALSE**. Попытка выполнения чтения в ситуациях, когда указатель файла позиционирован в конце

файла, не приводит к ошибке; вместо этого количество считанных байтов (***lpNumberOfBytesRead**) устанавливается равным нулю.

Запись в файл производится функцией

```
BOOL WriteFile(HANDLE hFile, LPVOID lpBuffer,
        DWORD nNumberOfBytesToWrite,
LPDWORD lpNumberOfBytesWrite, LPOVERLAPPED lpOverlapped);
```

Ее параметры аналогичны параметрам функции чтения из файла. Возвращаемое значение в случае успешного выполнения – **TRUE**, иначе – **FALSE**. Успешное выполнение записи еще не говорит о том, что данные действительно оказались записанными на диск, если только при создании файла с помощью функции **CreateFile** не был использован флаг **FILE_FLAG_WRITE_THROUGH**. Если во время вызова функции указатель файла был позиционирован в конце файла, Windows увеличит длину существующего файла.

Дополнительные функции работы с файлами

Так как ввод и вывод данных на диск в операционной системе Microsoft Windows NT (и выше) буферизуются, запись данных на диск может быть отложена до тех пор, пока система не освободится от выполнения текущей работы. С помощью функции **FlushFileBuffers** можно принудительно заставить операционную систему записать на диск все изменения для файла, идентификатор которого передается этой функции через единственный параметр

```
BOOL FlushFileBuffers (HANDLE hFile);
```

В случае успешного завершения функция возвращает значение **TRUE**, при ошибке – **FALSE**. Код ошибки можно получить при помощи функции **GetLastError**.

Напомним, что при закрывании файла функцией **CloseHandle** содержимое всех буферов, связанных с этим файлом, записывается на диск автоматически. Поэтому вы должны использовать функцию **FlushFileBuffers** только в том случае, если запись содержимого буферов нужно выполнить до закрывания файла.

С помощью функции **SetFilePointer** приложение может выполнять прямой доступ к файлу, перемещая указатель текущей позиции, связанный с файлом. Сразу после открывания файла этот указатель устанавливается в начало файла. Затем он передвигается функциями

ReadFile и **WriteFile** на количество прочитанных или записанных байтов соответственно.

Функция **SetFilePointer** позволяет выполнить установку текущей позиции:

```
DWORD SetFilePointer(HANDLE hFile, LONG lDistanceToMove,
    PLONG lpDistanceToMoveHigh, DWORD dwMoveMethod);
```

Через параметр **hFile** передается дескриптор файла, для которого выполняется изменение текущей позиции.

Параметр **lDistanceToMove**, определяющий дистанцию, на которую будет передвинута текущая позиция, может принимать как положительные, так и отрицательные значения. В первом случае текущая позиция переместится по направлению к концу файла, во втором – к началу файла.

Если размер файла не превышает 2^{32} – 2 байта, для параметра **lpDistanceToMoveHigh** можно указать значение **NULL**. Если ваш файл очень большой, для указания смещения может потребоваться 64-разрядное значение. Для того чтобы указать такое очень большое смещение, необходимо записать старшее 32-разрядное слово этого 64-разрядного значения в переменную и передать функции **SetFilePointer** адрес этой переменной через параметр **lpDistanceToMoveHigh**. Младшее слово смещения следует передавать как и раньше – через параметр **lDistanceToMove**.

Параметр **dwMoveMethod** определяет способ изменения текущей позиции и может принимать одно из перечисленных в табл. 4 значений.

Таблица 4

Параметры позиционирования в файле

Значение	Описание
FILE_BEGIN	Смещение отсчитывается от начала файла, при этом значение смещения трактуется как беззнаковая величина
FILE_CURRENT	Смещение отсчитывается от текущей позиции в файле и может принимать как положительные, так и отрицательные значения
FILE_END	Смещение отсчитывается от конца файла и трактуется как отрицательная величина

В случае успешного завершения функция **SetFilePointer** возвращает младшее слово новой 64-разрядной позиции в файле. Старшее слово при этом записывается по адресу, заданному параметром **lpDistanceToMoveHigh**.

При ошибке функция возвращает значение **0xFFFFFFFF**. При этом в слово по адресу **lpDistanceToMoveHigh** записывается значение **NULL**. Код ошибки можно получить при помощи функции **GetLastError**.

При необходимости изменить длину файла (уменьшить или увеличить) можно воспользоваться функцией **SetEndOfFile**. Эта функция устанавливает новую длину файла в соответствии с текущей позицией:

```
BOOL SetEndOfFile(HANDLE hFile);
```

Для изменения длины файла вам достаточно установить текущую позицию в нужное место с помощью функции **SetFilePointer**, а затем вызвать функцию **SetEndOfFile**.

Так как операционная система Microsoft Windows NT является мультизадачной и допускает одновременную работу многих процессов, возможно возникновение ситуаций, в которых несколько задач попытаются выполнять запись или чтение для одних и тех же файлов. Например, два процесса могут попытаться изменить одни и те же записи файла, при этом третий процесс будет в то же самое время выполнять чтение этой записи.

Напомним, что если функции **CreateFile** указать режимы совместного использования файла **FILE_SHARE_READ** или **FILE_SHARE_WRITE**, несколько процессов смогут одновременно открыть файлы и выполнять операции чтения и записи, соответственно.

Если же эти режимы не указаны, совместное использование файлов будет невозможно. Первый же процесс, открывший файл, заблокирует возможность работы с этим файлом для других процессов.

Очевидно, что иногда все же необходимо обеспечить возможность одновременной работы нескольких процессов с одним и тем же файлом. В этом случае при необходимости процессы могут блокировать доступ к отдельным фрагментам файлов для других процессов. Например, процесс, изменяющий запись, перед выполнением изменения может заблокировать участок файла, содержащий эту запись, и затем, после выполнения записи, разблокировать его. Другие процессы

не смогут выполнить запись или чтение для заблокированных участков файла.

Блокировка участка файла выполняется функцией **LockFile**, прототип которой представлен ниже:

```
BOOL LockFile(HANDLE hFile, DWORD dwFileOffsetLow,
DWORD dwFileOffsetHigh, DWORD nNumberOfBytesToLockLow,
DWORD nNumberOfBytesToLockHigh);
```

Параметр **hFile** задает дескриптор файла, для которого выполняется блокировка области. Смещение блокируемой области (64-разрядное) задается при помощи параметров **dwFileOffsetLow** (младшее слово) и **dwFileOffsetHigh** (старшее слово). Размер области в байтах задается параметрами **nNumberOfBytesToLockLow** (младшее слово) и **nNumberOfBytesToLockHigh** (старшее слово). Заметим, что если в файле блокируется несколько областей, они не должны перекрывать друг друга.

В случае успешного завершения функция **LockFile** возвращает значение **TRUE**, при ошибке – **FALSE**. Код ошибки можно получить при помощи функции **GetLastError**.

После использования заблокированной области (а также перед завершением своей работы) процессы должны разблокировать все заблокированные ранее области, вызвав для этого функцию **UnlockFile**:

```
BOOL UnlockFile(HANDLE hFile, DWORD dwFileOffsetLow,
DWORD dwFileOffsetHigh, DWORD nNumberOfBytesToUnlockLow,
DWORD nNumberOfBytesToUnlockHigh);
```

Вопросы для самопроверки

1. Перечислите основные параметры функции открытия файлов.
2. Перечислите основные параметры функций чтения и записи файлов.
3. Какие существуют дополнительные функции работы с файлами?
4. Напишите вызов функции открытия нового файла для чтения и записи с возможностью использования несколькими приложениями.
5. Напишите вызов функций открытия существующего файла для записи данных в конец файла (добавления записей).
6. Напишите вызов функций для перезаписи данных из одного открытого файла в другой.

Упражнения

1. Написать алгоритм (с указанием основных функций и их параметров) для открытия в нужных режимах и перезаписи данных из одного файла в другой.

2. Написать алгоритм (с указанием основных функций и их параметров) для открытия файла и записи данных в его конец (добавления записей).

3.2. Файлы, отображаемые на память

Механизм работы с файлами, отображаемыми на память, напоминает механизм работы виртуальной памяти.

Напомним, что в операционной системе Microsoft Windows NT и выше каждому процессу выделяется 2 Гбайта адресного пространства. Любой фрагмент этого пространства может быть отображен на фрагмент файла соответствующего размера.

На рис. 2 показано отображение фрагмента адресного пространства приложения размером в 1 Гбайт на фрагмент файла, имеющего размер 5 Гбайт. С помощью соответствующей функции программного интерфейса, которую рассмотрим далее, приложение Microsoft Windows NT может выбрать любой фрагмент большого файла для отображения в адресное пространство. Поэтому, несмотря на ограничение адресного пространства величиной в 2 Гбайт, вы можете отображать (по частям) в это пространство файлы любой длины, возможной в Microsoft Windows NT. В простейшем случае при работе с относительно небольшими файлами вы можете выбрать в адресном пространстве фрагмент подходящего размера и отобразить его на начало файла.

Как выполняется отображение фрагмента адресного пространства на фрагмент файла?

Если установлено такое отображение, то операционная система обеспечивает тождественность содержимого отображаемого фрагмента памяти и фрагмента файла, выполняя при необходимости операции чтения и записи в файл (с буферизацией и кэшированием).

В процессе отображения адресного пространства память не выделяется, а только резервируется. Поэтому отображение фрагмента размером 1 Гбайт не вызовет переполнения файлов виртуальной памяти. Более того, при отображении файлы виртуальной памяти вообще не используются, так как страницы фрагмента связываются с отображаемым файлом.

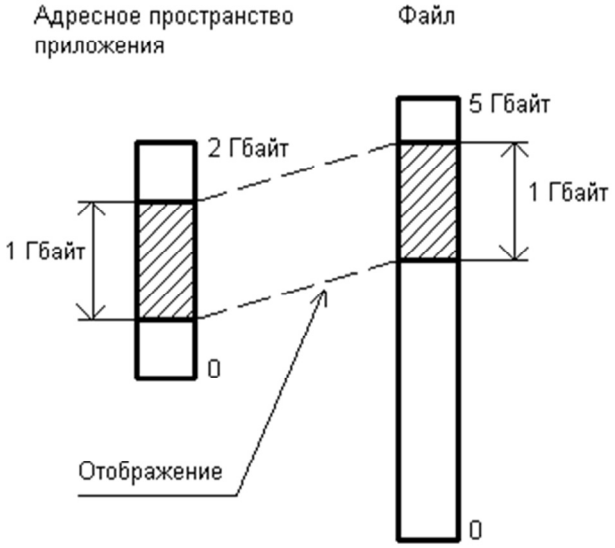

Рис. 2. Отображение фрагмента адресного пространства приложения на файл

Если приложение обращается в отображенный фрагмент для чтения, возникает исключение. Обработчик этого исключения загружает в физическую оперативную память соответствующую страницу из отображенного файла, а затем возобновляет выполнение прерванной команды. В результате в физическую оперативную память загружаются только те страницы, которые нужны приложению.

При записи происходит аналогичный процесс. Если нужной страницы нет в памяти, она подгружается из отображенного файла, затем в нее выполняется запись. Загруженная страница остается в памяти до тех пор, пока не будет вытеснена другой страницей при нехватке физической оперативной памяти. Что же касается записи измененной страницы в файл, то эта запись будет выполнена при закрытии файла, по явному запросу приложения или при выгрузке страницы из физической памяти для загрузки в нее другой страницы.

Заметим, что операционная система Microsoft Windows NT активно работает с файлами, отображаемыми в память. В частности, при загрузке исполнимого модуля приложения соответствующий exe- или dll-файл отображается на память, а затем ему передается управление. Когда пользователь запускает вторую копию приложения, для работы

используется файл, который уже отображается в память. В этом случае соответствующие страницы виртуальной памяти отображаются в адресные пространства обоих приложений.

Создание отображения файла

Рассмотрим процедуру создания отображения файла на память.

Прежде всего приложение должно открыть файл при помощи функции **CreateFile**, описанной ранее.

В случае успешного завершения функция **CreateFile** возвращает идентификатор открытого файла. При ошибке возвращается значение **INVALID_HANDLE_VALUE**. Здесь все как обычно, пока никакого отображения еще не выполняется.

Для того чтобы создать отображение файла, необходимо вызвать функцию **CreateFile Mapping**, прототип которой приведен ниже:

```
HANDLE CreateFileMapping(
  HANDLE hFile,       // идентификатор отображаемого
                      // файла
  LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
                      // дескриптор защиты
  DWORD flProtect,    // защита для отображаемого файла
  DWORD dwMaximumSizeHigh, // размер файла (старшее слово)
  DWORD dwMaximumSizeLow,  // размер файла (младшее слово)
 LPCTSTR lpName);          // имя отображенного файла
```

Через параметр **hFile** этой функции нужно передать идентификатор файла, для которого будет выполняться отображение в память, или значение **0xFFFFFFFF**. В первом случае функция **CreateFileMapping** отобразит заданный файл в память, а во втором – создаст отображение с использованием файла виртуальной памяти. Отображение с использованием файла виртуальной памяти удобно для организации передачи данных между процессами.

Заметим, что если функция **CreateFile** завершится с ошибкой и эта ошибка не будет обработана приложением, функция **CreateFileMapping** получит через параметр **hFile** значение **INVALID_HANDLE_VALUE**, численно равное **0xFFFFFFFF**. В этом случае она вместо того чтобы выполнить отображение файла в память, создаст отображение с использованием файла виртуальной памяти.

Параметр **lpFileMappingAttributes** задает адрес дескриптора защиты. В большинстве случаев для этого параметра вы можете указать значение **NULL**.

Параметр **flProtect** задает защиту для создаваемого отображения файла (табл. 5).

Таблица 5

Параметры защиты для отображаемого файла

Значение	Описание
PAGE_READONLY	К выделенной области памяти предоставляется доступ только для чтения. При создании или открывании файла необходимо указать флаг **GENERIC_READ**
PAGE_READWRITE	К выделенной области памяти предоставляется доступ для чтения и записи. При создании или открывании файла необходимо указать флаги **GENERIC_READ** и **GENERIC_WRITE**
PAGE_WRITECOPY	К выделенной области памяти предоставляется доступ для копирования при записи. При создании или открывании файла необходимо указать флаги **GENERIC_READ** и **GENERIC_WRITE**.

С помощью параметров **dwMaximumSizeHigh** и **dwMaximumSizeLow** необходимо указать 64-разрядный размер файла. Параметр **dwMaximumSizeHigh** должен содержать старшее 32-разрядное слово размера, а параметр **dwMaximumSizeLow** – малдшее 32-разрядное слово размера. Для небольших файлов, длина которых укладывается в 32 разряда, нужно указывать нулевое значение параметра **dwMaximumSizeHigh**.

Заметим, что можно указать нулевые значения для обоих этих параметров. В этом случае предполагается, что размер файла изменяться не будет.

Через параметр **lpName** можно указать имя отображения, которое будет доступно *всем работающим одновременно приложениям*. Имя должно представлять собой текстовую строку, закрытую двоичным нулем и не содержащую символов «\».

Если отображение будет использоваться только одним процессом, вы можете не задавать для него имя. В этом случае значение параметра **lpName** следует указать как **NULL**.

В случае успешного завершения функция **CreateFileMapping** возвращает идентификатор созданного отображения. При ошибке возвращается значение **NULL**.

Так как *имя отображения глобально*, возможно возникновение ситуации, когда процесс пытается создать отображение с уже существующим именем. В этом случае функция **CreateFileMapping** возвращает идентификатор существующего отображения. Такую ситуацию можно определить с помощью функции **GetLastError**, вызвав ее сразу после функции **CreateFileMapping**. Функция **GetLastError** при этом вернет значение **ERROR_ALREADY_EXISTS**.

Выполнение отображения файла в память

Получив от функции **CreateFileMapping** идентификатор объекта-отображения, необходимо выполнить само отображение, вызвав для этого функцию **MapViewOfFile**. В результате заданный фрагмент отображенного файла будет доступен в адресном пространстве процесса.

Прототип функции **MapViewOfFile**:

```
LPVOID MapViewOfFile(
  HANDLE hFileMappingObject, // идентификатор
                             // отображения
  DWORD dwDesiredAccess,     // режим доступа
  DWORD dwFileOffsetHigh,    // смещение в файле
                             // (старшее слово)
  DWORD dwFileOffsetLow,     // смещение в файле
                             // (младшее слово)
DWORD dwNumberOfBytesToMap); // количество
                             // отображаемых байт
```

Функция **MapViewOfFile** создает окно размером **dwNumberOfBytesToMap** байт, которое смещено относительно начала файла на количество байт, заданное параметрами **dwFileOffsetHigh** и **dwFileOffsetLow**. Если задать значение параметра **dwNumberOfBytesToMap**, равное нулю, будет выполнено отображение всего файла.

Смещение нужно задавать таким образом, чтобы оно попадало на границу минимального пространства памяти, которое можно зарезервировать. Значение «64 Кбайта» подходит в большинстве случаев.

Параметр **dwDesiredAccess** определяет требуемый режим доступа к отображению, т. е. режимы доступа для страниц виртуальной памяти, используемых для отображения (табл. 6).

Таблица 6

Режимы доступа к отображенному файлу

Значение	Описание
FILE_MAP_WRITE	Доступ на запись и чтение. При создании отображения функции **CreateFileMapping** необходимо указать тип защиты
FILE_MAP_READ	Доступ только на чтение. При создании отображения необходимо указать тип защиты **PAGE_READWRITE** или **PAGE_READ**
FILE_MAP_ALL_ACCESS	Аналогично **FILE_MAP_WRITE**
FILE_MAP_COPY	Доступ для копирования при записи. При создании отображения необходимо указать атрибут **PAGE_WRITECOPY**

В случае успешного выполнения отображения функция **MapViewOfFile** возвращает адрес отображенной области памяти. При ошибке возвращается значение **NULL**.

Приложение может создавать несколько отображений для разных или одинаковых фрагментов одного и того же файла.

Открывание отображения

Если несколько процессов используют совместно одно и то же отображение, первый процесс создает это отображение с помощью функции **CreateFileMapping**, указав имя отображения, а остальные должны открыть его, вызвав функцию **OpenFileMapping**:

```
HANDLE OpenFileMapping(
DWORD dwDesiredAccess,        // режим доступа
BOOL bInheritHandle,          // флаг наследования
```

```
LPCTSTR lpName);              // адрес имени отображения
                              // файла
```

Через параметр **lpName** этой функции следует передать имя открываемого отображения. Имя должно быть задано точно так же, как при создании отображения функцией **CreateFileMapping**.

Параметр **dwDesiredAccess** определяет требуемый режим доступа к отображению и указывается точно так же, как и для описанной выше функции **MapViewOfFile**.

Параметр **bInheritHandle** определяет возможность наследования идентификатора отображения. Если он равен **TRUE**, порожденные процессы могут наследовать идентификатор, если **FALSE** – то нет.

Отмена отображения файла

Если созданное отображение больше не нужно, его следует отменить с помощью функции **UnmapViewOfFile**:

```
BOOL UnmapViewOfFile(LPVOID lpBaseAddress);
```

Через единственный параметр этой функции необходимо передать адрес области отображения, полученный от функций **MapViewOfFile**.

В случае успеха функция возвращает значение **TRUE**. При этом гарантируется, что все измененные страницы оперативной памяти, расположенные в отменяемой области отображения, будут записаны на диск в отображаемый файл. При ошибке функция возвращает значение **FALSE**.

Если приложение создало несколько отображений для файла, перед завершением работы с файлом все они должны быть отменены с помощью функции **UnmapViewOfFile**. Далее с помощью функции **CloseHandle** следует закрыть идентификаторы отображений, полученных от функции **CreateFile**.

Принудительная запись измененных данных

Как было отмечено ранее, после отмены отображения все измененные страницы памяти записываются в отображаемый файл. Если это потребуется, приложение может в любое время выполнить принудительную запись измененных страниц в файл при помощи функции **FlushViewOfFile**:

```
BOOL FlushViewOfFile(
  LPCVOID lpBaseAddr,   // начальный адрес сохраняемой
                        // области
DWORD dwNumberOfBytesToFlush);    // размер области
                                  // в байтах
```

С помощью параметров **lpBaseAddr** и **dwNumberOfBytesToFlush** вы можете выбрать любой фрагмент внутри области отображения, для которого будет выполняться сохранение измененных страниц на диске. Если задать значение параметра **dwNumberOfBytesToFlush** равным нулю, будут сохранены все измененные страницы, принадлежащие области отображения.

Примеры приложений

Для примера на сайте дисциплины в папке **Mapping** имеется исходный текст приложения *mapfile.cpp*, которое читает и пишет не сами файлы, а их отображения. Текстовый файл и символ, который нужно из него удалить, задаются через аргументы командной строки. Единственная особенность – отображение выходного файла задается, когда известен размер его содержимого.

Вопросы для самопроверки

1. Каковы особенности создания отображения файла?
2. Как выполняется отображение файла в память?
3. Каковы особенности операций открывания отображения, отмены отображения файла и принудительной записи измененных данных?
4. Запишите функции, необходимые для перезаписи данных из одного файла в другой с использованием отображения.

5. Запишите функции, необходимые для создания в режиме записи отображения несуществующего файла, записи в него данных, его открытия в другом приложении на чтение и чтения из него.

Упражнения

1. Написать алгоритм (с указанием основных функций и их параметров) для перезаписи данных из одного файла в другой с использованием отображения.

2. Написать алгоритм (с указанием основных функций и их параметров) для создания в режиме записи отображения несуществующего файла, записи в него данных, его открытия в другом приложении на чтение и чтения из него.

4. ДИНАМИЧЕСКИЕ БИБЛИОТЕКИ

Библиотеки динамической компоновки DLL (Dynamic Link Libraries) являются стержневым компонентом операционной системы Windows NT и многих приложений Windows. Без преувеличения можно сказать, что вся операционная система Windows, все ее драйверы, а также другие расширения есть не что иное, как набор библиотек динамической компоновки. Редкое крупное приложение Windows не имеет собственных библиотек динамической компоновки, и ни одно приложение не может обойтись без вызова функций, расположенных в таких библиотеках. В частности, все функции программного интерфейса WinAPI находятся именно в библиотеках динамической компоновки DLL.

Статическая и динамическая компоновка

Прежде чем приступить к изучению особенностей использования библиотек динамической компоновки в операционной системе Microsoft Windows, напомним кратко, чем отличаются друг от друга статическая и динамическая компоновка [5].

При использовании статической компоновки готовится исходный текст приложения, затем он транслируется для получения объектного модуля. После этого редактор связей компонует объектные модули, полученные в результате трансляции исходных текстов, и модули из библиотек объектных модулей в один исполнимый exe-файл. В процессе запуска файл программы полностью загружается в оперативную память и ему передается управление.

Таким образом, при использовании статической компоновки редактор связей записывает в файл программы все модули, необходимые для работы. В любой момент времени в оперативной памяти компьютера находится весь код, необходимый для работы запущенной программы.

В среде мультизадачной операционной системы статическая компоновка неэффективна, так как приводит к неэкономному использованию очень дефицитного ресурса – оперативной памяти. Представьте себе, что в системе одновременно работают пять приложений и все они вызывают такие функции, как **`sprintf`**, **`memcpy`**, **`strcmp`** и т. д. Если приложения были собраны с использованием статической компоновки, в памяти будут находиться одновременно пять копий функции **`sprintf`**, пять копий функции **`memcpy`** и т. д (рис. 3).

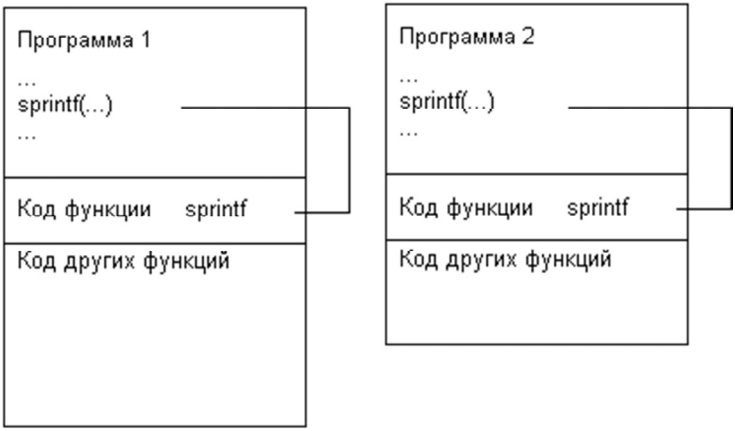

Рис. 3. Вызов функций при использовании статической компоновки

Очевидно, использование оперативной памяти было бы намного эффективнее, если бы в памяти находилось только по одной копии функций, а все работающие параллельно программы могли бы их вызывать.

Практически в любой многозадачной операционной системе для любого компьютера используется именно такой способ обращения к функциям, нужным одновременно большому количеству работающих параллельно программ.

При использовании динамической компоновки загрузочный код нескольких (или нескольких десятков) функций объединяется в отдельные файлы, загружаемые в оперативную память в единственном экземпляре. Программы, работающие параллельно, вызывают функции, загруженные в память из файлов библиотек динамической компоновки, а не из файлов программ.

Таким образом, используя механизм динамической компоновки, в загрузочном файле программы можно расположить только те функции, которые являются специфическими для данной программы. Те же функции, которые нужны всем (или многим) программам, работающим параллельно, можно вынести в отдельные файлы – библиотеки динамической компоновки и хранить в памяти в единственном экземпляре (рис. 4). Эти файлы можно загружать в память только при необходимости, например, когда какая-нибудь программа захочет вызвать функцию, код которой расположен в библиотеке.

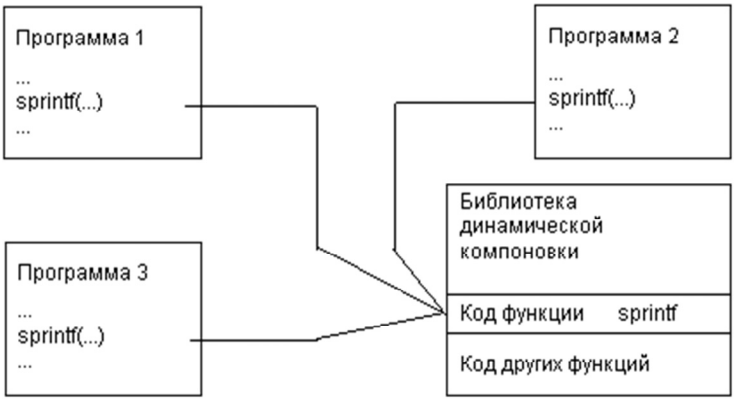

Рис. 4. Вызов функции при использовании динамической компоновки

В операционной системе Windows файлы библиотек динамической компоновки имеют расширение имени *dll*, хотя можно использовать любое другое, например exe. В первых версиях Windows DLL-библиотеки располагались в файлах с расширением имени *exe*. Возможно поэтому файлы *krnl286.exe*, *krnl386.exe*, *gdi.exe* и *user.exe* имели расширение имени exe, а не dll, несмотря на то что перечисленные выше файлы, составляющие ядро операционной системы Windows версии 3.1, есть не что иное, как DLL-библиотеки. Наиболее важные компоненты операционной системы Microsoft Windows расположены в библиотеках с именами *kernel32.dll* (ядро операционной системы), *user32.dll* (функции пользовательского интерфейса), *gdi32.dll* (функции для рисования изображений и текста).

Механизм динамической компоновки был изобретен задолго до появления операционных систем Windows и OS/2 (которая также

активно использует механизм динамической компоновки). Например, в мультизадачных многопользовательских операционных системах VS1, VS2, MVS, VM, созданных для компьютеров IBM-370 и аналогичных, код функций, нужных параллельно работающим программам, располагается в отдельных библиотеках и может загружаться при необходимости в специально выделенную общую область памяти.

Отображение страниц DLL-библиотеки

В среде Microsoft Windows DLL-библиотека загружается в страницы виртуальной памяти, которые отображаются в адресные пространства всех «заинтересованных» приложений, которым нужны функции из этой библиотеки. При этом используется механизм, аналогичный отображению файлов на память, рассмотренный ранее.

На рис. 5 схематически показано отображение кода и данных DLL-библиотеки в адресные пространства двух приложений.

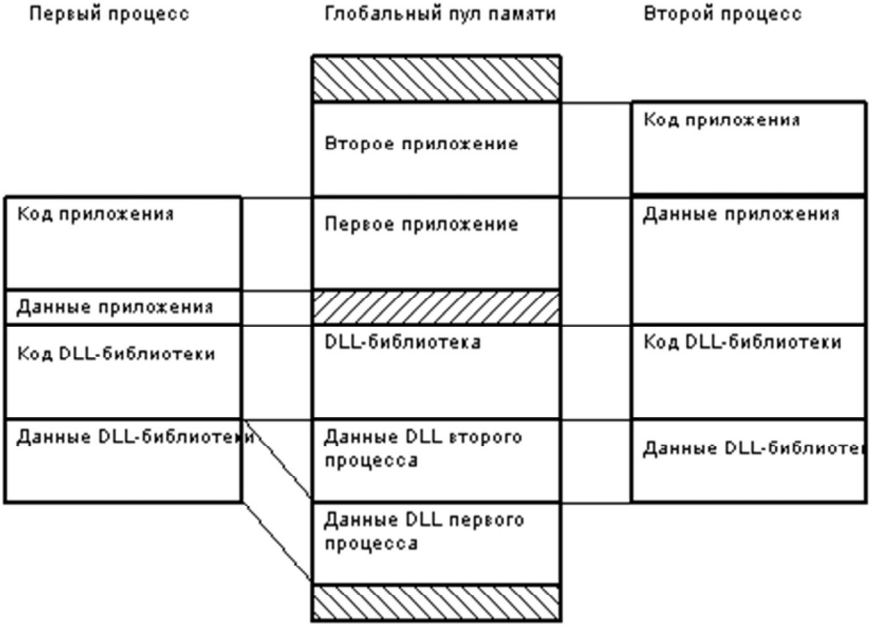

Рис. 5. Отображение DLL-библиотеки в адресные пространства двух процессов

На этом рисунке показано, что в глобальном пуле памяти находится один экземпляр кода DLL-библиотеки, который отображается в адресные пространства приложений (процессов). Что же касается данных DLL-библиотеки, то для каждого приложения в глобальном пуле создается отдельная область. Таким образом, различные приложения не могут в этом случае передавать друг другу данные через общую область данных DLL-библиотеки.

Тем не менее принципиальная возможность создания глобальных областей памяти DLL-библиотеки, доступных разным процессам, существует. Для этого необходимо при редактировании описать область данных DLL-библиотеки как SHARED.

Заметим, что одна и та же функция DLL-библиотеки может отображаться на разные адреса в различные адресные пространства приложений. Это же относится к глобальным и статическим переменным DLL-библиотеки – они будут отображаться на разные адреса для различных приложений.

Когда первое приложение загрузит DLL-библиотеку в память (явно или неявно), эта библиотека (точнее говоря, страницы памяти, в которые она загружена) будет отображена в адресное пространство этого приложения. Если теперь другое приложение попытается загрузить ту же самую библиотеку еще раз, то для него будет создано новое отображение тех же самых страниц. На этот раз страницы могут быть отображены уже на другие адреса.

Кроме того, для каждой DLL-библиотеки система ведет счетчик использования (usage count). Содержимое этого счетчика увеличивается при очередной загрузке библиотеки в память и уменьшается при освобождении библиотеки.

Когда содержимое счетчика использования DLL-библиотеки станет равным нулю, библиотека будет выгружена из памяти.

Обмен данными между приложениями через DLL-библиотеку

Напомним, что DLL-библиотеки не имеют собственных областей данных, а отображаются в адресные пространства приложений, загружающих эти библиотеки. Как результат, приложения не могут получать адреса статических и глобальных переменных и использовать эти переменные для обмена данными, – ведь адрес, верный в контексте одного приложения, не будет иметь никакого смысла для другого приложения.

Приложение также может сделать попытку изменить содержимое статической или глобальной переменной, с тем чтобы другие приложения могли прочитать новое значение. Однако этот способ передачи данных между приложениями не будет работать.

Попытка изменения данных будет зафиксирована операционной системой, которая создаст для этого приложения копию страницы памяти, в которой находятся изменившиеся данные, с использованием механизма копирования при записи (copy-on-write).

Если по какой-либо причине нельзя использовать для обмена данными между приложениями файлы, отображаемые на память, можно создать DLL-библиотеку с данными, имеющими атрибут **SHARED**.

Для этого нужно задать имя секции данных, которая будет использована для передачи данных между процессами. Это можно сделать с помощью прагмы транслятора **data_seg**:

```
#pragma data_seg (".shar")
```

После этого в файле определения модуля для DLL-библиотеки необходимо указать атрибуты секции данных, как это показано ниже:

```
SECTIONS .shar READ WRITE SHARED
```

Можно также указать ключ редактору связей:

```
-SECTION:.shar,RWS
```

Строка **RWS** определяет атрибуты секции: R – **READ**, W – **WRITE**, S – **SHARED**.

При обращении к глобальным переменным, расположенным в секции с атрибутом **SHARED**, процессы должны выполнять взаимную синхронизацию с использованием таких средств, как критические секции, объекты-события, семафоры, которые будут описаны позже.

Инициализация DLL-библиотеки в среде Microsoft Windows

В 32-разрядных DLL-библиотеках операционной системы Microsoft Windows используется функция **DLLEntryPoint**, которая выполняет все необходимые задачи по инициализации библиотеки и при необходимости освобождает заказанные ранее ресурсы (имя функции инициализации может быть любым). Функция **DLLEntryPoint** вызывается всякий раз, когда выполняется инициализация процесса или задачи,

обращающихся к функциям библиотеки, а также при явной загрузке и выгрузке библиотеки функциями **LoadLibrary**и **FreeLibrary**.

Ниже приведен прототип функции **DLLEntryPoint**:

```
BOOL WINAPI DllEntryPoint(
  HINSTANCE hinstDLL,   // идентификатор модуля
                        // DLL-библиотеки
  DWORD     fdwReason,  // код причины вызова функции
LPVOID    lpvReserved);  // зарезервировано
```

Через параметр **hinstDLL** функции **DLLEntryPoint** передается идентификатор модуля DLL-библиотеки, который можно использовать при обращении к ресурсам, расположенным в файле этой библиотеки.

Что же касается параметра **fdwReason**, то он зависит от причины, по которой произошел вызов функции **DLLEntryPoint**. Этот параметр может принимать следующие значения (табл. 7).

Таблица 7

Причины вызовы функции DLLEntryPoint

Значение	Описание
DLL_PROCESS_ATTACH	Библиотека отображается в адресное пространство процесса в результате запуска процесса или вызова функции **LoadLibrary**
DLL_THREAD_ATTACH	Текущий процесс создал новый поток, после чего система вызывает функции **DLLEntryPoint** всех DLL-библиотек, подключенных к процессу
DLL_THREAD_DETACH	Этот код причины передается функции **DLLEntryPoint**, когда поток завершает свою работу нормальным (не аварийным) способом
DLL_PROCESS_DETACH	Отображение DLL-библиотеки в адресное пространство отменяется в результате нормального завершения процесса или вызова функции **FreeLibrary**

Параметр **lpvReserved** зарезервирован. В документации тем не менее сказано, что значение параметра **lpvReserved** равно **NULL** во всех случаях, кроме двух следующих:

- когда параметр **fdwReason** равен **DLL_PROCESS_ATTACH** и используется статическая загрузка DLL-библиотеки;

• когда параметр **fdwReason** равен **DLL_PROCESS_DETACH** и функция **DLLEntryPoint** вызвана в результате завершения процесса, а не вызова функции **FreeLibrary.**

В процессе инициализации функция **DLLEntryPoint** может отменить загрузку DLL-библиотеки. Если код причины вызова равен **DLL_PROCESS_ATTACH**, функция **DLLEntryPoint** отменяет загрузку библиотеки, возвращая значение **FALSE**. Если же инициализация выполнена успешно, функция должна возвратить значение **TRUE**.

В том случае, когда приложение пыталось загрузить DLL-библиотеку функцией **LoadLibrary**, а функция **DLLEntryPoint** отменила загрузку, функция **LoadLibrary** возвратит значение **NULL**. Если же приложение выполняет инициализацию DLL-библиотеки неявно, при отмене загрузки библиотеки приложение также не будет загружено для выполнения.

Приведем пример функции инициализации DLL-библиотеки:

```
BOOL WINAPI DLLEntryPoint(
  HMODULE hModule,      // идентификатор модуля
  DWORD   fdwReason,    // причина вызова функции
                        // DLLEntryPoint
LPVOID  lpvReserved)    // зарезервировано
{
  switch(fdwReason)
  {
    // Подключение нового процесса
    case DLL_PROCESS_ATTACH:
    {
      // Обработка подключения процесса
      . . .
      break;
    }
    // Подключение нового потока
    case DLL_THREAD_ATTACH:
    {
      // Обработка подключения нового потока
. . .
break;
    }
    // Отключение процесса
case DLL_PROCESS_DETACH:
{
```

```
// Обработка отключения процесса
        . . .
        break;
    }
    // Отключение потока
    case DLL_THREAD_DETACH:
    {
      // Обработка отключения потока
      . . .
      break;
    }
  }
  return TRUE;
}
```

Экспортирование функций и глобальных переменных

Хотя существуют DLL-библиотеки без исполнимого кода и предназначенные для хранения ресурсов, подавляющее большинство DLL-библиотек экспортирует функции для их совместного использования несколькими приложениями.

Кроме функции **DLLEntryPoint** в 32-разрядных библиотеках операционных систем Microsoft Windows могут быть определены *экспортируемые* и *неэкспортируемые* функции.

Экспортируемые функции доступны для вызова приложениям Windows. Неэкспортируемые являются локальными для DLL-библиотеки, они доступны только для функций библиотеки.

При необходимости можно экспортировать из 32-разрядных DLL-библиотек не только функции, но и глобальные переменные.

Самый простой способ сделать функцию экспортируемой – перечислить все экспортируемые функции в файле определения модуля при помощи оператора **EXPORTS**:

```
   EXPORTS
ИмяТочкиВхода [=ВнутрИмя] [@Номер] [NONAME]
[CONSTANT]
   . . .
```

Здесь **ИмяТочкиВхода** задает имя, под которым экспортируемая из DLL-библиотеки функция будет доступна для вызова.

Внутри DLL-библиотеки эта функция может иметь другое имя. В этом случае необходимо указать ее внутреннее имя **ВнутрИмя**.

С помощью параметра **@Номер** можно задать порядковый номер экспортируемой функции.

Если не указать порядковые номера экспортируемых функций, при компоновке загрузочного файла DLL-библиотеки редактор связей создаст свою собственную нумерацию, которая может изменяться при внесении изменений в исходные тексты функций и последующей повторной компоновке.

Заметим, что ссылка на экспортируемую функцию может выполняться двумя различными способами – по имени функции и по ее порядковому номеру. Если функция вызывается по имени, ее порядковый номер не имеет значения. Однако вызов функции по порядковому номеру выполняется быстрее, поэтому использование порядковых номеров предпочтительнее.

Если при помощи файла определения модуля DLL-библиотеки задается фиксированное распределение порядковых номеров экспортируемых функций, при внесении изменений в исходные тексты DLL-библиотеки это распределение не изменится. В этом случае все приложения, ссылающиеся на функции из этой библиотеки по их порядковым номерам, будут работать правильно. Если же не определены порядковые номера функций, у приложений могут возникнуть проблемы с правильной адресацией функции из-за возможного изменения этих номеров.

Указав флаг **NONAME** и порядковый номер, можно сделать имя экспортируемой функции невидимым. При этом экспортируемую функцию можно будет вызвать только по порядковому номеру, так как имя такой функции не попадет в таблицу экспортируемых имен DLL-библиотеки.

Флаг **CONSTANT** позволяет экспортировать из DLL-библиотеки не только функции, но и данные. При этом параметр **ИмяТочкиВхода** задает имя экспортируемой глобальной переменной, определенной в DLL-библиотеке.

Приведем пример экспортирования функций и глобальных переменных из DLL-библиотеки:

```
EXPORTS
  DrawBitmap=MyDraw  @4
  ShowAll
  HideAll
  MyPoolPtr    @5 CONSTANT
  GetMyPool    @8 NONAME
FreeMyPool  @9 NONAME
```

В приведенном выше примере в разделе **EXPORTS** перечислены имена нескольких экспортируемых функций **DrawBitmap**, **ShowAll**, **HideAll**, **GetMyPool**, **FreeMyPool** и глобальной переменной **MyPoolPtr**.

Функция **MyDraw**, определенная в DLL-библиотеке, экспортируется под именем **DrawBitmap**. Она также доступна под номером 4.

Функции **ShowAll** и **HideAll** экспортируются под своими «настоящими» именами, с которыми они определены в DLL-библиотеке. Для них не заданы порядковые номера.

Функции **GetMyPool** и **FreeMyPool** экспортируются с флагом **NONAME**, поэтому к ним можно обращаться только по их порядковым номерам, которые равны, соответственно, 8 и 9.

Имя **MyPoolPtr** экспортируется с флагом **CONSTANT**, поэтому оно является именем глобальной переменной, определенной в DLL-библиотеке и доступной для приложений, загружающих эту библиотеку.

Импортирование функций

Когда используется статическая компоновка, то в файл проекта приложения включается соответствующий lib-файл, содержащий нужную библиотеку объектных модулей. Такая библиотека содержит исполняемый код модулей, который на этапе статической компоновки включается в exe-файл загрузочного модуля.

Если используется динамическая компоновка, в загрузочный exe-файл приложения записывается не исполнимый код функций, а ссылка на соответствующую DLL-библиотеку и функцию внутри нее. Эта ссылка может быть организована с использованием либо имени функции, либо ее порядкового номера в DLL-библиотеке.

Откуда при компоновке приложения редактор связей узнает имя DLL-библиотеки, имя или порядковый номер экспортируемой функции? Для динамической компоновки функции из DLL-библиотеки можно использовать различные способы.

Библиотека импорта

Для того чтобы редактор связей мог создать ссылку, в файл проекта приложения необходимо включить так называемую *библиотеку импорта* (import library). Эта библиотека создается автоматически системой разработки Microsoft Visual C++.

Следует заметить, что стандартные библиотеки систем разработки приложений Windows содержат как обычные объектные модули, предназначенные для статической компоновки, так и ссылки на различные стандартные DLL-библиотеки, экспортирующие функции программного интерфейса операционной системы Windows.

Динамический импорт функций во время выполнения приложения

Приложение может в любой момент времени загрузить любую DLL-библиотеку, вызвав специально предназначенную для этого функцию программного интерфейса Windows с именем **`LoadLibrary`**. Приведем ее прототип:

```
HINSTANCE WINAPI LoadLibrary (LPCSTR lpszLibFileName);
```

Параметр функции является указателем на текстовую строку, закрытую двоичным нулем. В эту строку перед вызовом функции следует записать путь к файлу DLL-библиотеки или имя этого файла. Если путь к файлу не указан, при поиске выполняется последовательный просмотр следующих каталогов:

- каталог, из которого запущено приложение;
- текущий каталог;
- 32-разрядный системный каталог Microsoft Windows;
- каталог, в котором находится операционная система Windows;
- каталоги, перечисленные в переменной описания среды PATH.

Если файл DLL-библиотеки найден, функция **`LoadLibrary`** возвращает идентификатор модуля библиотеки. В противном случае

возвращается значение **NULL**. При этом код ошибки можно получить при помощи функции **GetLastError**.

Функция **LoadLibrary** может быть вызвана разными приложениями для одной и той же DLL-библиотеки несколько раз. В этом случае загрузка DLL-библиотеки выполняется только один раз. Последующие вызовы функции **LoadLibrary** приводят только к увеличению счетчика использования DLL-библиотеки. При многократном вызове функции **LoadLibrary** различными процессами функция инициализации DLL-библиотеки получает несколько раз управление с кодом причины вызова, равным значению **DLL_PROCESS_ATTACH.**

В качестве примера приведем фрагмент исходного текста приложения, загружающего DLL-библиотеку из файла DLLDEMO.DLL:

```
typedef HWND (WINAPI *MYDLLPROC)(LPSTR);
MYDLLPROC    GetAppWindow;
HANDLE       hDLL;

hDLL = LoadLibrary("DLLDEMO.DLL");
if(hDLL != NULL)
{
  GetAppWindow = (MYDLLPROC)GetProcAddress(hDLL,
    "FindApplicationWindow");
if(GetAppWindow != NULL)
  {
if(GetAppWindow(szWindowTitle) != NULL)
MessageBox(NULL, "Application window was found",
szAppTitle, MB_OK | MB_ICONINFORMATION);
else
MessageBox(NULL, "Application window was not found",
szAppTitle, MB_OK | MB_ICONINFORMATION);
  }
FreeLibrary(hDLL);
}
```

Здесь в начале с помощью функции **LoadLibrary** выполняется попытка загрузки DLL-библиотеки DLLDEMO.DLL.

В случае успеха приложение получает адрес точки входа для функции с именем **FindApplicationWindow**, для чего используется функция **GetProcAddress**. Эту функцию обсудим немного позже.

Если точка входа получена, функция вызывается через указатель **GetAppWindow**.

После использования DLL-библиотека освобождается при помощи функции **FreeLibrary**, прототип который показан ниже:

```
void WINAPI FreeLibrary(HINSTANCE hLibrary);
```

В качестве параметра этой функции следует передать идентификатор освобождаемой библиотеки.

При освобождении DLL-библиотеки ее счетчик использования уменьшается. Если этот счетчик становится равным нулю (что происходит, когда все приложения, работавшие с библиотекой, освободили ее или завершили свою работу), DLL-библиотека выгружается из памяти.

Каждый раз при освобождении DLL-библиотеки вызывается функция **DLLEntryPoint** с параметрами **DLL_PROCESS_DETACH** или **DLL_THREAD_DETACH**, выполняющая все необходимые завершающие действия.

Теперь о функции **GetProcAddress**.

Для того чтобы вызвать функцию из библиотеки, зная ее идентификатор, необходимо получить значение дальнего указателя на эту функцию, вызвав функцию **GetProcAddress**:

```
FARPROC WINAPI GetProcAddress(HINSTANCE hLibrary,
                    LPCSTR lpszProcName);
```

Через параметр **hLibrary** необходимо передать функции идентификатор DLL-библиотеки, полученный ранее от функции **LoadLibrary**.

Параметр **lpszProcName** является дальним указателем на строку, содержащую имя функции или ее порядковый номер, преобразованный макрокомандой **MAKEINTRESOURCE**.

Приведем фрагмент кода, в котором определяются адреса двух функций. В первом случае используется имя функции, а во втором – ее порядковый номер:

```
FARPROC lpMsg;
FARPROC lpTellMe;
lpMsg = GetProcAddress(hLib, "Msg");
lpTellMe = GetProcAddress(hLib, MAKEINTRESOURCE(8));
```

Перед тем как передать управление функции по полученному адресу, следует убедиться в том, что этот адрес не равен **NULL**:

```
if(lpMsg != (FARPROC)NULL)
{
  (*lpMsg)((LPSTR)"My message");
}
```

Для того чтобы включить механизм проверки типов передаваемых параметров, можно определить свой тип – указатель на функцию и затем использовать его для преобразования типа адреса, полученного от функции **GetProcAddress**:

```
typedef int (*LPGETZ)(int x, int y);
LPGETZ lpGetZ;
lpGetZ = (LPGETZ)GetProcAddress(hLib, "GetZ");
```

А что произойдет, если приложение при помощи функции **LoadLibrary** попытается загрузить DLL-библиотеку, которой нет на диске? В этом случае операционная система Microsoft Windows выведет на экран диалоговую панель с сообщением о том, что она не может найти нужную DLL-библиотеку. В некоторых случаях появление такого сообщения нежелательно, например, по логике работы приложения описанная ситуация является нормальной.

Для того чтобы отключить режим вывода диалоговой панели с сообщением о невозможности загрузки DLL-библиотеки, можно использовать функцию **SetErrorMode**, передав ей в качестве параметра значение **SEM_FAILCRITICALERRORS**:

```
UINT nPrevErrorMode;
nPrevErrorMode = SetErrorMode(SEM_FAILCRITICALERRORS);
hDLL = LoadLibrary("DLLDEMO.DLL");
if(hDLL != NULL)
{
  // Работа с DLL-библиотекой
  . . .
}
SetErrorMode(nPrevErrorMode);
```

Примеры динамической библиотеки и приложения

Для примера на сайте дисциплины в папке **dll** имеется исходный текст приложения *testdll.cpp*, которое загружает библиотеку с именем *testfunc.dll* и вызывает из нее функцию **TestHello.** Приложение про-

веряет доступность библиотеки и наличие функции в ней. Функция точки входа DllMain библиотеки уведомляет о ее загрузке и выгрузке, а собственно вызываемая функция TestHello выводит сообщение «Hello, you're calling a function in this DLL». Библиотека создана с помощью шаблона *Мастера приложений* Win32 среды программирования VisualStudio и содержит файлы исходного кода, содержащие две указанные функции, и файл определения модуля Source.def, содержащий перечень экспортируемых функций.

Вопросы для самопроверки

1. Что такое статическая и динамическая компоновка программ, в чем их отличие?
2. Как выполняется отображение страниц DLL-библиотеки?
3. Как и когда производится инициализация DLL-библиотеки? Какие события при этом могут быть обработаны?
4. Как и где описать экспортируемые и неэкспортируемые функции DLL-библиотеки?
5. Как задать внешнее имя экспортируемой функции, ее номер для вызова?
6. Как определить глобальную переменную DLL-библиотеки, доступную для экспорта только по номеру?
7. Как и откуда можно загрузить DLL-библиотеку? Как ее выгрузить?
8. Как убедиться, что DLL-библиотека загружена приложением?
9. Как импортировать функцию из DLL-библиотеки по имени, по номеру?
10. Как убедиться, что импорт функции из DLL-библиотеки прошел успешно?
11. Как вызвать импортированную из DLL-библиотеки функцию?

Упражнения

1. Написать функцию инициализации DLL-библиотеки.
2. Написать алгоритм (с указанием основных функций и их параметров) для загрузки DLL-библиотеки, импорта функции, ее вызова, выгрузки библиотеки.

5. МНОГОЗАДАЧНОЕ ПРОГРАММИРОВАНИЕ В WINDOWS

Процесс есть объект, обладающий собственным независимым виртуальным адресным пространством, в котором могут размещаться код и данные, защищенные от других процессов. Внутри каждого процесса могут независимо выполняться один или несколько потоков. Поток может создавать новые потоки и новые независимые процессы, управлять взаимодействием объектов между собой и их синхронизацией.

Внутри каждого процесса могут выполняться один или несколько потоков, и именно поток является базовой единицей выполнения в Windows. Выполнение потоков планируется системой на основе обычных факторов: наличие таких ресурсов, как CPU и физическая память, приоритеты, равнодоступность ресурсов и г. д. Начиная с Windows NT4 поддерживается симметричная многопроцессорная обработка (Symmetric Multiprocessing, SMP), позволяющая распределять выполнение потоков между отдельными процессорами.

Каждому процессу принадлежат следующие ресурсы:

- один или несколько потоков;
- виртуальное адресное пространство, отличное от адресных пространств других процессов;
- один или несколько сегментов кода, включая код DLL;
- один или несколько сегментов данных, содержащих глобальные переменные;
- строки, содержащие информацию об окружении, например, информацию о текущем пути доступа к файлам;
- область нераспределенной памяти (куча, heap) процесса;
- различного рода ресурсы (например, дескрипторы открытых файлов).

Поток разделяет вместе с процессом код, глобальные переменные, строки окружения и другие ресурсы. Каждый поток планируется независимо от других и располагает следующими элементами:

• стек, используемый для вызова процедур, прерываний и обработчиков исключений, а также хранения автоматических переменных;

• локальные области хранения потока (Thread Local Storage, TLS) – массивы указателей, используя которые каждый поток может создавать собственную уникальную информационную среду;

• аргумент в стеке, получаемый от создающего потока, который обычно является уникальным для каждого потока;

• структура контекста, поддерживаемая ядром системы и содержащая значения машинных регистров.

На рис. 6 показан процесс с несколькими потоками.

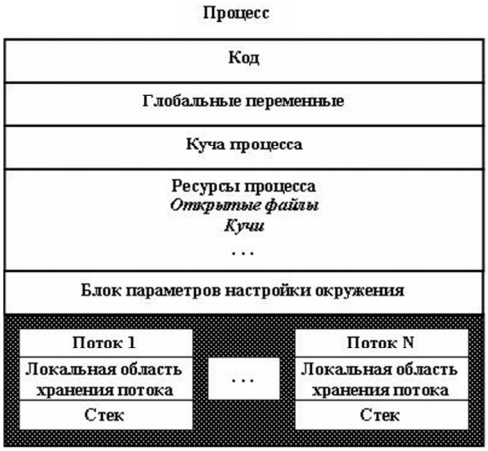

Рис. 6. Процесс и его потоки

Создание процесса

Функция **CreateProcess** создает новый процесс с единственным потоком. При вызове этой функции требуется указать имя файла исполняемой программы. Для большинства параметров можно использовать значения, заданные по умолчанию (**NULL**):

```
BOOL CreateProcess(lpApplicationName,
LPTSTR lpCommandLine,
LPSECURITY_ATTRIBUTES lpsaProcess,
LPSECURITY_ATTRIBUTES lpsaThread,
```

```
BOOL bInheritHandles,
DWORD dwCreationFlags,
LPVOID lpEnvironment,
LPCTSTR lpCurDir,
LPSTARTUPINFO lpStartupInfo,
LPPROCESS_INFORMATION lpProcInfo);
```

Функция возвращает два дескриптора, по одному для процесса и потока, передавая их в структуре типа **LPROCESS_INFORMATION**. Эти дескрипторы относятся к создаваемому функцией **CreateProcess** новому процессу и его *основному* потоку. Во избежание утечки ресурсов в процессе работы необходимо закрывать оба дескриптора, когда они больше не нужны. Возвращаемое значение: в случае успешного создания процесса и потока – **TRUE**, иначе – **FALSE**.

Параметры

lpApplicationName и **lpCommandLine** (последний указатель имеет тип **LPTSTR**, а не **LPCTSTR**)

Указатели используются для указания исполняемой программы и аргументов командной строки. При этом действуют следующие правила.

- Указатель **lpApplicationName**, если его значение не равно **NULL**, указывает на строку, содержащую имя файла исполняемого модуля. Если имя модуля содержит пробелы, его заключают в кавычки.
- Если значение указателя **lpApplicationName** равно **NULL**, то имя модуля определяется первой из лексем в параметре **lpCommandLine**.

Обычно задается только параметр **lpCommandLine**, в то время как параметр **lpApplicationName** полагается равным **NULL**. Новый процесс может получить командную строку посредством обычного **argv**-механизма или путем вызова функции **GetCommandLine** для получения командной строки в виде одиночной строки символов.

lpsaProcess и **lpsaThread** – указатели на структуры атрибутов защиты процесса и потока. Значениям **NULL** соответствует использование атрибутов защиты, заданных по умолчанию.

bInheritHandles – режим наследования открытых дескрипторов файлов и т. д. из вызывающего процесса. Для наследования значение параметра устанавливают равным **TRUE**, иначе **FALSE**.

dwCreationFlags – может объединять в себе несколько флаговых значений, включая следующие:

- **CREATESUSPENDED** – указывает на то, что основной поток будет создан в приостановленном состоянии и начнет выполняться лишь после вызова функция **ResumeThread**;
- **DETACHED_PROCESS** и **CREATE_NEW_CONSOLE** – взаимоисключающие флаги: первый означает создание нового процесса, у которого консоль отсутствует, а второй – процесса, у которого имеется собственная консоль. Если ни один из этих флагов не указан, то новый процесс наследует консоль родительского процесса;
- **Create_New_Process_Group** – указывает на то, что создаваемый процесс является корневым для новой группы процессов. Если все процессы, принадлежащие данной группе, разделяют общую консоль, то все они будут получать управляющие сигналы консоли (Ctrl-C);
- флаги управления приоритетами потоков нового процесса. Можно использовать приоритет родительского процесса (устанавливается по умолчанию) или указывать значение **NORMAL_PRIORITY_CLASS**.

lpEnvironment – блок параметров настройки окружения нового процесса. Если задано значение **NULL**, то новый процесс будет использовать значения параметров окружения родительского процесса.

lpCurDir – указатель на строку, содержащую путь к текущему каталогу нового процесса. Если задано значение **NULL**, то будет использоваться рабочий каталог родительского процесса.

lpStartupInfo – указатель на структуру, которая описывает внешний вид основного окна и содержит дескрипторы стандартных устройств нового процесса. Соответствующую информацию из родительского процесса можно получить при помощи функции **GetStartupInfo**. Можно обнулить структуру **STARTUPINFO** перед вызовом функций **CreateProcess**.

lpProcInfo – указатель на структуру, в которую помещаются значения дескрипторов и идентификаторов процесса и потока.

Структура **PROCESS_INFORMATION** имеет следующий вид:

```
typedef struct PROCESS_INFORMATION {
HANDLE hProcess;HANDLE hThread;
DWORD dwProcessId;DWORD dwThreadld;}
PROCESS_INFORMATION;
```

Процессам и потокам нужны и дескрипторы, и идентификаторы, поскольку одним функциям управления требуются идентификаторы, а другим – дескрипторы. Дескрипторы процессов и потоков должны закрываться после того как необходимость в них отпала.

Завершение и прекращение выполнения процесса

После того как процесс завершил свою работу, он может вызвать функцию **ExitProcess**, указав в качестве параметра код завершения:

```
VOID ExitProcess (UINT uExitCode)
```

Эта функция не осуществляет возврата. Она завершает вызывающий процесс и все его потоки. Выполнение оператора **return** в основной программе с использованием кода возврата равносильно вызову функции **ExitProcess**, в котором этот код возврата указан в качестве кода завершения. Другой процесс может определить код завершения, вызвав функцию **GetExitCodeProcess**:

```
BOOL GetExitCodeProcess(HANDLE hProcess,
                        LPDWORD lpExitCode)
```

Процесс, идентифицируемый дескриптором **hProcess**, должен обладать правами доступа **PROCESS_QUERY_INFORMATION** (см. описание функции **OpenProcess**). **lpExitCode** указывает на переменную типа **DWORD**, которая принимает значение кода завершения. Одним из ее возможных значений является **STILL_ACTIVE**, означающее, что данный процесс еще не завершился.

Один процесс может прекратить выполнение другого процесса, если у дескриптора завершаемого процесса имеются права доступа **PROCESS_TERMINATE**. При вызове функции завершения процесса указывается код завершения:

```
BOOL TerminateProcess (HANDLE hProcess, UINT uExitCode)
```

Прежде чем завершить выполнение процесса, все ресурсы, которые он мог разделять с другими процессами, должны быть освобождены.

Ожидание завершения процесса

Простейшим методом синхронизации с другим процессом является ожидание его завершения. Представленные ниже стандартные функции ожидания Windows обладают рядом интересных свойств.

Функции ожидания могут работать с самыми различными типами объектов (процессами и потоками).

Функции могут ожидать завершения одного процесса, первого из нескольких указанных, или всех процессов, образующих группу.

Есть возможность устанавливать конечный интервал ожидания.

```
DWORD WaitForSingleObject (
HANDLE hObject,
DWORD dwMilliseconds);
DWORD WaitForMultipleObjects (
DWORD nCount,
CONST HANDLE *lpHandles,
BOOLfWaitAll,
DWORDdwMilliseconds);
```

Возвращаемое значение указывает причину завершения ожидания или, в случае ошибки, равно **0xFFFFFFFF** (для получения более подробной информации используйте функцию **GetLastError**).

В аргументах этих функций указывается либо дескриптор одиночного процесса (**hObject**), либо дескрипторы ряда отдельных объектов, хранящиеся в массиве, на который указывает указатель **lpHandles**. Значение параметра **nCount**, определяющего размер массива, не должно превышать 64 (определено в файле WINNT.H).

dwMilliseconds – число миллисекунд интервала ожидания. Если число равно нулю, то возврат из функции осуществляется сразу же, что полезно при опросе состояния процессов. Если же значение параметра равно **INFINITE**, то ожидание длится до завершения процесса.

fWaitAll – параметр, указывающий (если его значение равно **TRUE**) на необходимость ожидания завершения всех процессов.

Возможные возвращаемые значения функций:

- **WAIT_OBJECT_0** – указанный объект завершен или остановлен (в случае функции **WaitForSingleObject**) или одновременно все

nCount объектов завершены или остановлены (в случае функции **WaitForMultipleObjects**, когда параметр **fWaitAll** равен **TRUE**);

- **WAIT_OBJECT_0+n**, где 0 <n<**nCount** – вычтя значение **WAIT_OBJECT_0** из возвращенного значения, можно определить, выполнение какого процесса завершилось, если ожидается завершение выполнения любого из группы процессов. Если завершено несколько объектов, возвращается наименьшее из возможных значений;
- **WAIT_TIMEOUT** – указывает на то, что в течение отведенного периода ожидания объект (объекты) не завершены;
- **WAIT_FAILED** – неудачное завершение функции, вызванное, например, тем, что у дескриптора отсутствовали права доступа.

Примеры родительского и дочернего приложений

Для примера на сайте дисциплины имеется родительское приложение *spaces_new.cpp*, которое для каджого аргумента командной строки порождает дочерний процесс *file_new.cpp*, который, в свою очередь, получает один аргумент командной строки и интерпретирует его как имя файла, подлежащего обработке. Если файл существует и доступен для дочернего приложения, то он открывается на чтение и в нем подсчитывается количество пробелов, которое выводится на экран вместе с идентификатором текущего дочернего приложения и имени обработанного файла и возвращается, как результат работы дочернего приложения. Если указанный файл не может быть открыт, дочернее приложение выводит на экран сообщение об ошибке и возвращает значение –1 как признак ошибки. Родительский процесс дожидается завершения всех дочерних, получает коды их завершения и выводит на экран имена файлов и количество пробелов в них.

Приложения созданы как консольные с помощью шаблона *Мастера приложений* Win32 среды программирования VisualStudio.

Вопросы для самопроверки

1. Дайте определение процесса, потока.
2. Перечислите ресурсы, принадлежащие процессу, потоку.
3. Перечислите параметры, передаваемые в функцию порождения процесса.
4. Перечислите параметры, возвращаемые из функции порождения процесса.

5. Перечислите функции завершения и прекращения выполнения процесса.

6. Какие существуют функции ожидания завершения процесса? В чем их отличия?

7. Как получить код завершения процесса?

Упражнение

Написать алгоритм (с указанием основных функций и их параметров) для порождения процесса, ожидания его завершения в течение заданного времени, получения кода завершения, если процесс завершился, и прекращения выполнения, если он еще работает.

6. WINDOWS IPC

Введение в Windows IPC

Ранее было показано, как создавать процессы и управлять ими, но в качестве средства взаимодействия между процессами использовались аргументы командной строки и возвращаемые дочерними процессами значения, а также простые текстовые файлы.

Далее рассматриваются средства последовательного межпроцессного взаимодействия (Interprocess Communication, IPC), в котором используются объекты, подобные файлам.

Два основных механизма Windows, реализующие IPC, это анонимные и именованные каналы, доступ к которым осуществляется с помощью известных функций **`ReadFile`** и **`WriteFile`**. Простые анонимные каналы являются символьными и работают в полудуплексном режиме. Эти свойства делают их удобными для перенаправления выходных данных одной программы на вход другой, как это обычно делается в UNIX. Схема взаимодействия родственных процессов с использованием анонимных каналов приведена на рис. 7.

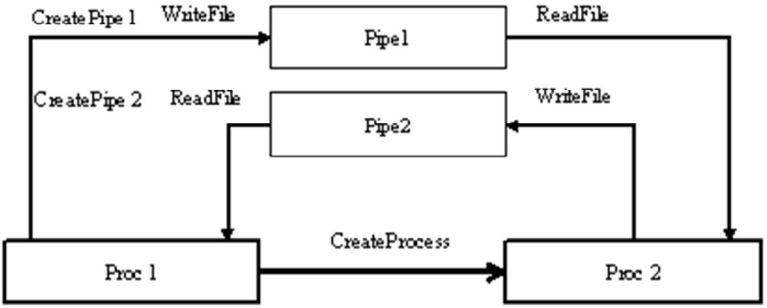

Рис. 7. Взаимодействие родственных процессов с использованием анонимных каналов

По сравнению с анонимными каналами возможности именованных каналов гораздо богаче. Они являются дуплексными, ориентированы на обмен сообщениями и обеспечивают взаимодействие через сеть. Кроме того, один именованный канал может иметь несколько открытых дескрипторов. В сочетании с удобными, ориентированными на выполнение транзакций, функциями эти возможности делают именованные каналы пригодными для создания клиент-серверных систем. Схема взаимодействия процессов с использованием именованных каналов приведена на рис. 8.

В том случае, когда требуется обеспечить передачу данных только в одном направлении, можно использовать так называемые почтовые ящики Mailslot. Они удобны тем, что их можно использовать для организации широковещательной передачи данных между процессами, запущенными на различных рабочих станциях сети. Схема взаимодействия процессов с использованием почтовых ящиков приведена на рис. 9.

Кроме того, можно организовать передачу данных между процессами, работающими в разных адресных пространствах, с использованием файлов, отображенных на память. Методика использования файлов, отображенных на память, для передачи данных между процессами заключается в следующем.

Один из процессов создает такой файл, задавая при этом имя отображения. Данное имя является глобальным и доступно для всех процессов, запущенных в системе. Другие процессы могут воспользоваться именем отображения, открыв созданный ранее файл. В результате оба процесса могут получить указатели на область памяти, для которой выполнено отображение, и эти указатели будут ссылаться на одни и те же страницы виртуальной памяти.

Обмениваясь данными через эту область, процессы должны обеспечить синхронизацию своей работы, например, с помощью событий, объектов взаимного исключения (mutual exclusion, mutex) или семафоров (в зависимости от логики процесса обмена данными). Мьютексы удобнее использовать в многопоточных приложениях, потому они будут рассмотрены позже. Использование же событий (events) и семафоров для блокировки доступа к отображениям (последнее характерно и для UNIX-подобных ОС при работе с разделяемой памятью) будет рассмотрено в рамках данной главы. Схема взаимодействия процессов с использованием отображаемых файлов и перечисленных методов их синхронизации приведена на рис. 10.

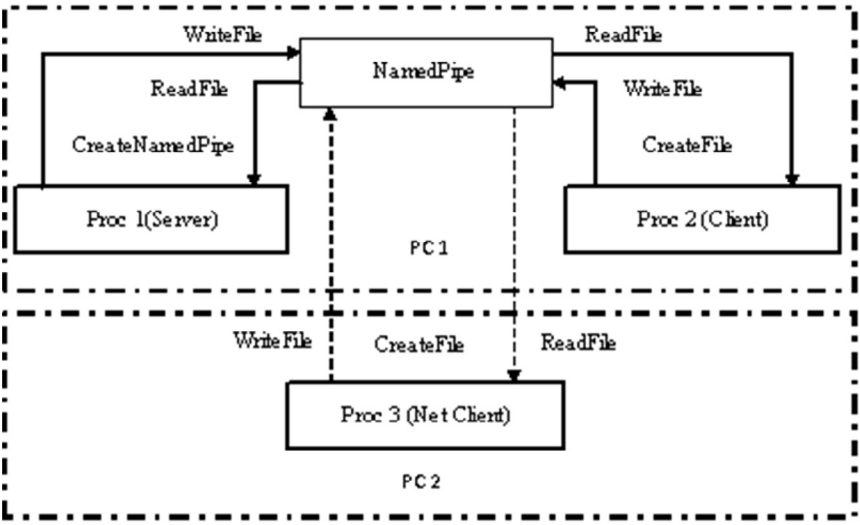

Рис. 8. Взаимодействие процессов с использованием именованных каналов

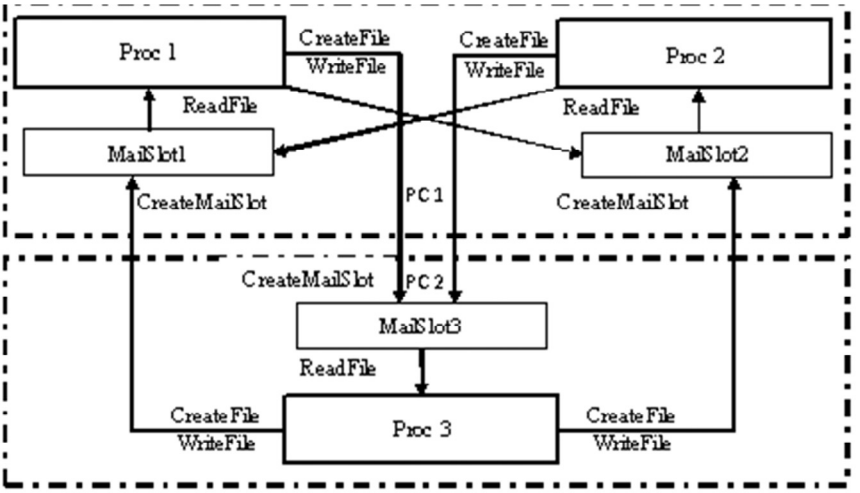

Рис. 9. Взаимодействие процессов с использованием почтовых ящиков

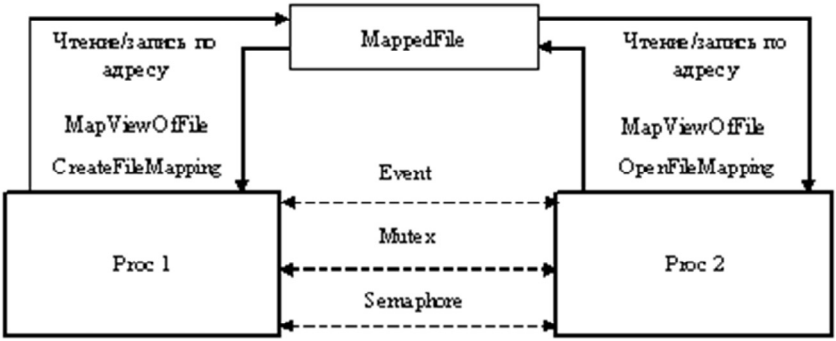

Рис. 10. Взаимодействие процессов с использованием отображений файлов

Вопросы для самопроверки

1. Что такое IPC?
2. Перечислите основные средства взаимодействия процессов.
3. Перечислите основные средства синхронизации процессов.

6.1. Каналы передачи данных

В среде операционных систем Microsoft Windows (от NT 4.0 и выше) доступно такое удобное средство передачи данных между параллельно работающими процессами, как каналы. Это средство позволяет организовать передачу данных между локальными процессами, а также между процессами, запущенными на различных рабочих станциях в сети. Каналы больше всего похожи на файлы, поэтому они достаточно просты в использовании.

Через канал можно передавать данные только между двумя процессами. Один из процессов создает канал, другой открывает его. После этого оба процесса могут передавать данные через канал в одну или обе стороны, используя для этого хорошо известные функции, предназначенные для работы с файлами, такие как **`ReadFile`** и **`WriteFile`**. Заметим, что приложения могут выполнять над каналами синхронные или асинхронные операции, аналогично тому, как это можно делать с файлами. В случае использования асинхронных операций необходимо отдельно побеспокоиться об организации синхронизации.

Существуют две разновидности каналов: анонимные (Anonymous Pipes) и именованные (Named Pipes).

Анонимные каналы обычно используются для организации передачи данных между родительскими и дочерними процессами, запущенными на одной рабочей станции или на «отдельно стоящем» компьютере.

Как видно из названия, именованным каналам при создании присваивается имя, которое доступно для других процессов. Зная имя какой-либо рабочей станции в сети, процесс может получить доступ к каналу, созданному на этой рабочей станции.

6.1.1. Анонимные каналы

Анонимные каналы Windows обеспечивают однонаправленное (полудуплексное) посимвольное межпроцессное взаимодействие. Каждый канал имеет два дескриптора: дескриптор чтения (read handle) и дескриптор записи (write handle).

Дескрипторы каналов часто бывают наследуемыми; причины этого станут понятными из приведенного ниже примера. Чтобы канал можно было использовать для IPC, должен существовать еще один процесс, и для этого процесса требуется один из дескрипторов канала. Предположим, например, что родительскому процессу, создавшему канал, необходимо вывести в него данные, которые нужны дочернему процессу. Тогда возникает вопрос о том, как передать дочернему процессу дескриптор чтения (**hRead**). Родительский процесс осуществляет это, устанавливая дескриптор стандартного ввода в структуре **STARTUPINFO** для дочерней процедуры равным ***hRead**.

Чтение с использованием дескриптора чтения канала блокируется, если канал пуст. В противном случае в процессе чтения будет воспринято столько байтов, сколько имеется в канале, вплоть до количества, указанного при вызове функции **ReadFile**. Операция записи в заполненный канал, которая выполняется с использованием буфера в памяти, также будет блокирована.

Наконец, анонимные каналы обеспечивают только однонаправленное взаимодействие. Для двухстороннего взаимодействия необходимы два канала.

Для создания анонимных каналов используется функция **CreatePipe**, имеющая следующий прототип:

```
BOOL CreatePipe(
  PHANDLE hReadPipe,     // адрес переменной, в которую
                         // будет записан
                         // идентификатор канала
                         // для чтения данных
  PHANDLE hWritePipe,    // адрес переменной, в которую
                         // будет записан
                         // идентификатор канала
                         // для записи данных
  LPSECURITY_ATTRIBUTES lpPipeAttributes,
                         // адрес переменной для атрибутов
                         // защиты количество байт памяти,
                         // зарезервированной для канала
DWORD nSize);
```

Канал может использоваться как для записи в него данных, так и для чтения. Поэтому при создании канала функция **CreatePipe** возвращает два идентификатора, записывая их по адресу, заданному в параметрах **hReadPipe** и **hWritePipe**.

Идентификатор, записанный по адресу **hReadPipe**, можно передавать в качестве параметра функции **ReadFile** для выполнения операции чтения. Идентификатор, записанный по адресу **hWritePipe**, передается функции **WriteFile** для выполнения операции записи.

Через параметр **lpPipeAttributes** передается адрес переменной, содержащей атрибуты защиты для создаваемого канала. В наших приложениях мы будем указывать этот параметр как **NULL**. В результате канал будет иметь атрибуты защиты, принятые по умолчанию.

И наконец, параметр **nSize** определяет размер буфера для создаваемого канала. Если этот размер указан как нуль, будет создан буфер с размером, принятым по умолчанию.

Заметим, что при необходимости система может изменить указанный вами размер буфера.

В случае успеха функция **CreatePipe** возвращает значение **TRUE**, при ошибке – **FALSE**. В последнем случае для уточнения причины возникновения ошибки вы можете воспользоваться функцией **GetLastError**.

Запись данных в канал

Запись данных в открытый канал выполняется с помощью функции **WriteFile**, аналогично записи в обычный файл:

```
HANDLE hNamedPipe;
DWORD  cbWritten;
char   szBuf[256];
WriteFile(hNamedPipe, szBuf, strlen(szBuf) + 1,
&cbWritten, NULL);
```

Через первый параметр функции **WriteFile** передается идентификатор реализации канала. Через второй параметр передается адрес буфера, данные из которого будут записаны в канал. Размер этого буфера указывается при помощи третьего параметра. Предпоследний параметр используется для определения количества байтов данных, действительно записанных в канал. И наконец, последний параметр задан как **NULL**, поэтому запись будет выполняться в синхронном режиме.

Учтите, что если канал был создан для работы в блокирующем режиме и функция **WriteFile** работает синхронно (без использования вывода с перекрытием), то эта функция не вернет управление до тех пор, пока данные не будут записаны в канал.

Чтение данных из канала

Как и следовало ожидать, для чтения данных из канала можно воспользоваться функцией **ReadFile**, например, так:

```
HANDLE hNamedPipe;
DWORD  cbRead;
char   szBuf[256];
ReadFile(hNamedPipe, szBuf, 512, &cbRead, NULL);
```

Данные, прочитанные из канала **hNamedPipe**, будут записаны в буфер **szBuf**, имеющий размер 512 байт. Количество действительно прочитанных байт данных будет сохранено функцией **ReadFile** в переменной **cbRead**. Так как последний параметр функции указан как **NULL**, используется синхронный режим работы без перекрытия.

Закрытие идентификатора канала

Если канал больше не нужен, процессы должны закрыть его идентификатор функцией **CloseHandle**:

```
CloseHandle(hNamedPipe);
```

Пример применения анонимных каналов

В примере ниже (рис. 11) представлен родительский процесс, который создает дочерний процесс и соединяет его с каналом. Родительский процесс устанавливает канал и осуществляет перенаправление стандартного ввода/вывода. Обратите внимание на то, каким образом задается свойство наследования дескрипторов анонимного канала и как организуется перенаправление стандартного ввода/вывода в дочернем процессе.

Дескрипторы каналов и потоков должны закрываться при первой же возможности. Родительский процесс должен закрыть дескриптор устройства стандартного вывода сразу же после создания дочернего процесса, чтобы тот мог распознать метку конца файла. В случае существования открытого дескриптора первого процесса второй процесс не смог бы завершиться, поскольку система не обозначила бы конец файла.

Обратите внимание на то, каким образом задается свойство наследования дескрипторов анонимного канала:

```
/* Перенаправить стандартный ввод/вывод. */
STARTUPINFO StartInfoChild;
GetStartupInfo(&StartInfoChild);
StartInfoChild.hStdInput =
hReadPipe1;//GetStdHandle(hReadPipe);
StartInfoChild.hStdError =
GetStdHandle(STD_ERROR_HANDLE);
StartInfoChild.hStdOutput = hWritePipe2;
StartInfoChild.dwFlags = STARTF_USESTDHANDLES;
```

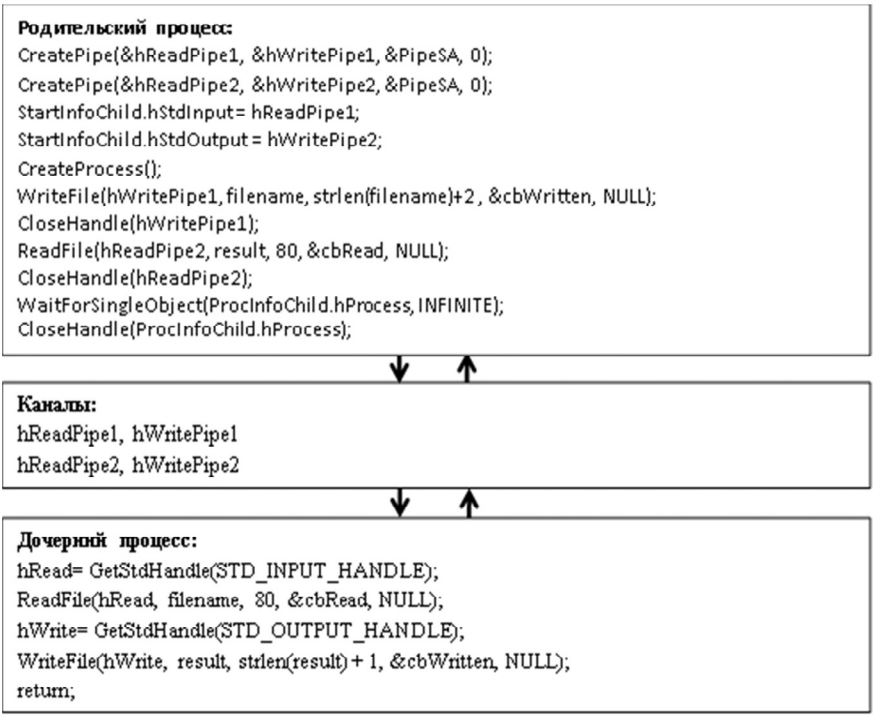

Рис. 11. Межпроцессное взаимодействие с использованием анонимного канала

Дочерний процесс при запуске должен унаследовать потоки ввода/вывода:

```
CreateProcess(NULL, (LPTSTR)Command, NULL, NULL, TRUE /*
Унаследовать дескрипторы. */, 0, NULL, NULL,
&StartInfoChild, &ProcInfoChild);
```

А вот как организуется перенаправление стандартного ввода/вывода в дочернем процессе:

```
// чтение из канала
hRead=GetStdHandle(STD_INPUT_HANDLE);
ReadFile(hRead, filename, 80, &cbWritten, NULL);
. . .
```

```
   // сообщение в консоль ошибок
   hError=GetStdHandle(STD_ERROR_HANDLE);
   WriteFile(hError,      message,      strlen(message),
&cbWritten, NULL);
   // запись в канал
   hWrite= GetStdHandle(STD_OUTPUT_HANDLE);
   WriteFile(hWrite,  message,  strlen(message)  +  1,
&cbWritten, NULL);
```

Примеры родительского и дочернего приложений

Пример родительской (*pipe_parent.cpp*) и дочерней (*pipe_child.cpp*) программ, использующих анонимные каналы, представлен на сайте дисциплины в папке **Winpipes**.

Вопросы для самопроверки

1. Каковы особенности анонимных каналов?
2. Как производится наследование дескрипторов анонимных каналов дочерним процессом?
3. Каковы параметры функции создания анонимного канала?
4. Какие функции применяются для чтения и записи данных в анонимные каналы?

Упражнения

1. Написать алгоритм (с указанием основных функций и их параметров) для обмена данными через анонимные каналы со стороны родительского процесса.
2. Написать алгоритм (с указанием основных функций и их параметров) для обмена данными через анонимные каналы со стороны дочернего процесса.

6.1.2. Именованные каналы

Именованные каналы (Named Pipes) предлагают ряд возможностей, которые делают их полезными в качестве универсального механизма реализации приложений на основе IPC, включая приложения, требующие сетевого доступа к файлам, и клиент-серверные системы. К числу упомянутых возможностей (часть которых обеспечивается дополнительно) относятся следующие.

- Именованные каналы ориентированы на обмен сообщениями, поэтому процесс, выполняющий чтение, может считывать сообщения переменной длины именно в том виде, в каком они были посланы процессом, выполняющим запись.
- Именованные каналы являются двунаправленными, что позволяет осуществлять обмен сообщениями между двумя процессами посредством единственного канала.
- Допускается существование нескольких независимых экземпляров канала, имеющих одинаковые имена. Например, с единственной серверной системой могут связываться одновременно несколько клиентов, использующих каналы с одним и тем же именем. Каждый клиент может иметь собственный экземпляр именованного канала, и сервер может использовать этот же канал для отправки ответа клиенту.
- Каждая из систем, подключенных к сети, может обратиться к каналу, используя его имя. Взаимодействие посредством именованного канала осуществляется одинаковым образом для процессов, выполняющихся как на одной и той же, так и на разных машинах.
- Имеется несколько вспомогательных и связных функций, упрощающих обслуживание взаимодействия «запрос/ответ» и клиент-серверных соединений.

Во всех случаях, когда требуется, чтобы канал связи был двунаправленным, ориентированным на обмен сообщениями или доступным для нескольких клиентских процессов, следует применять именованные каналы.

Использование именованных каналов

Как видно из названия, именованным каналам при создании присваивается имя, которое доступно для других процессов. Зная имя какой-либо рабочей станции в сети, процесс может получить доступ к каналу, созданному на этой рабочей станции.

Имена каналов в общем случае имеют следующий вид:

\\ИмяСервера\pipe\ИмяКанала

Если процесс открывает канал, созданный на другой рабочей станции, он должен указать имя сервера. Если же процесс создает канал или открывает канал на своей рабочей станции, вместо имени указывается символ точки:

\\.\pipe\ИмяКанала

В любом случае процесс может создать канал только на той рабочей станции, где он запущен, поэтому при создании канала имя сервера никогда не указывается.

Серверами именованных каналов могут быть только системы на основе Windows NT 4.0,2000, 2003, 2008, 2012 Server; системы на базе рабочих станций Windows (9x и выше) могут выступать только в роли клиентов.

В простейшем случае один серверный процесс создает один канал (точнее говоря, одну реализацию канала) для работы с одним клиентским процессом.

Однако часто требуется организовать взаимодействие одного серверного процесса с несколькими клиентскими. Например, сервер базы данных может принимать от клиентов запросы и рассылать ответы на них.

В случае такой необходимости серверный процесс может создать несколько реализаций канала, по одной реализации для каждого клиентского процесса.

Функция **CreateNamedPipe** создает первый экземпляр именованного канала и возвращает дескриптор. При вызове этой функции указывается также максимально допустимое количество экземпляров каналов, а следовательно, и количество клиентов, одновременная поддержка которых может быть обеспечена.

Как правило, создающий процесс рассматривается в качестве *сервера. Клиентские процессы,* которые могут выполняться и на других системах, открывают канал с помощью функции **CreateFile**.

Для создания именованного канала необходимо использовать функцию **CreateNamedPipe**. Вот прототип этой функции:

```
HANDLE CreateNamedPipe(
LPCTSTR lpName,             // адрес строки имени канала
DWORD dwOpenMode,           // режим открытия канала
```

```
DWORD dwPipeMode,          // режим работы канала
DWORD nMaxInstances,     // максимальное количество
                         // реализаций канала
DWORD nOutBufferSize,    // размер выходного буфера
                         // в байтах
DWORD nInBufferSize,     // размер входного буфера в байтах
DWORD nDefaultTimeOut,   // время ожидания в миллисекундах
LPSECURITY_ATTRIBUTES lpSecurityAttributes);
                         // адрес атрибутов защиты
```

Через параметр **lpName** передается адрес строки имени канала в форме **\\.\pipe\ИмяКанала** (напомним, что при создании канала имя сервера не указывается, так как канал можно создать только на той рабочей станции, где запущен процесс, создающий канал).

Параметр **dwOpenMode** задает режим, в котором открывается канал. Остановимся на этом параметре подробнее. Канал может быть ориентирован либо на передачу потока байтов, либо на передачу сообщений. В первом случае данные через канал передаются по байтам, во втором – отдельными блоками заданной длины.

Режим работы канала (ориентированный на передачу байтов или сообщений) задается соответственно константами **PIPE_TYPE_BYTE** или **PIPE_TYPE_MESSAGE**, которые указываются в параметре **dwOpenMode**. По умолчанию используется режим **PIPE_TYPE_BYTE**.

Помимо способа передачи данных через канал, с помощью параметра **dwOpenMode** можно указать, будет ли данный канал использован только для чтения данных, только для записи или одновременно для чтения и записи. Способ использования канала задается указанием одной из следующих констант (табл. 8).

Таблица 8

Способы использования именованного канала

Константа	Использование канала
PIPE_ACCESS_INBOUND	Только для чтения
PIPE_ACCESS_OUTBOUND	Только для записи
PIPE_ACCESS_DUPLEX	Для чтения и записи

Перечисленные выше параметры должны быть одинаковы для всех реализаций канала (о реализациях канала мы говорили выше). Далее

мы перечислим параметры, которые могут отличаться для разных реализаций канала (табл. 9).

Таблица 9

Параметры реализации именованного канала

Константа	Использование канала
PIPE_READMODE_BYTE	Канал открывается на чтение в режиме последовательной передачи отдельных байтов
PIPE_READMODE_ MESSAGE	Канал открывается на чтение в режиме передачи отдельных сообщений указанной длины
PIPE_WAIT	Канал будет работать в блокирующем режиме, когда процесс переводится в состояние ожидания до завершения операций в канале
PIPE_NOWAIT	Неблокирующий режим работы канала. Если операция не может быть выполнена немедленно, в неблокирующем режиме функция завершается с ошибкой
FILE_FLAG_ OVERLAPPED	Использование асинхронных операций (ввод и вывод с перекрытием). Данный режим позволяет процессу выполнять полезную работу параллельно с проведением операций в канале
FILE_FLAG_WRITE_ THROUGH	В этом режиме функции, работающие с каналом, не возвращают управление до тех пор, пока не будет полностью завершена операция на удаленном компьютере. Используется только с каналом, ориентированном на передачу отдельных байт, и только в том случае, когда канал создан между процессами, запущенными на различных станциях сети

Дополнительно к перечисленным выше флагам через параметр **dwOpenMode** можно передавать следующие флаги защиты (табл. 10).

Подробное описание этих флагов выходит за рамки курса. При необходимости следует обратиться к документации, входящей в состав SDK.

Теперь перейдем к параметру **dwPipeMode**, определяющему режим работы канала. В этом параметре вы можете указать перечисленные выше константы **PIPE_TYPE_BYTE**, **PIPE_TYPE_MESSAGE**, **PIPE_ READMODE_BYTE**, **PIPE_READMODE_MESSAGE**, **PIPE_WAIT** и **PIPE_NOWAIT**.

Для всех реализаций канала необходимо указывать один и тот же набор констант.

Таблица 10

Флаги защиты именованного канала

Флаг	Описание
WRITE_DAC	Вызывающий процесс должен иметь права доступа на запись к произвольному управляющему списку доступа именованного канала access control list (ACL)
WRITE_OWNER	Вызывающий процесс должен иметь права доступа на запись к процессу, владеющему именованным каналом
ACCESS_SYSTEM_SECURITY	Вызывающий процесс должен иметь права доступа на запись к управляющему списку доступа именованного канала access control list (ACL)

Параметр **nMaxInstances** определяет максимальное количество реализаций, которые могут быть созданы для канала. Вы можете указывать здесь значения от единицы до **PIPE_UNLIMITED_INSTANCES**. В последнем случае максимальное количество реализаций ограничивается только наличием свободных системных ресурсов.

Заметим, что если один серверный процесс использует несколько реализаций канала для связи с несколькими клиентскими, то общее количество реализаций может быть меньше, чем потенциальное максимальное количество клиентов. Это связано с тем, что клиенты могут использовать реализации по очереди, если только они не пожелают связаться с серверным процессом все одновременно.

Параметры **nOutBufferSize** и **nInBufferSize** определяют соответственно размер буферов, используемых для записи в канал и чтения из канала. При необходимости система может использовать буферы других, по сравнению с указанными, размеров.

Параметр **nDefaultTimeOut** определяет время ожидания для реализации канала. Для всех реализаций необходимо указывать одинаковое значение этого параметра (в миллисекундах).

Через параметр **lpPipeAttributes** передается адрес переменной, содержащей атрибуты защиты для создаваемого канала. В примерах этот параметр указан как **NULL**. В результате канал будет иметь атрибуты защиты, принятые по умолчанию.

В случае успеха функция **CreateNamedPipe** возвращает идентификатор созданной реализации канала, который можно использовать

в операциях чтения и записи, выполняемых с помощью таких функций, как **ReadFile** и **WriteFile**.

При ошибке функция **CreateNamedPipe** возвращает значение **INVALID_HANDLE_VALUE**. Код ошибки вы можете уточнить, вызвав функцию **GetLastError**.

Приведем пример использования функции **CreateNamedPipe** для создания именованного канала с именем **$MyPipe$**, предназначенным для чтения данных, работающем в блокирующем режиме и допускающем создание неограниченного количества реализаций:

```
HANDLE hNamedPipe;
LPSTR  lpszPipeName = "\\\\.\\pipe\\$MyPipe$";
hNamedPipe = CreateNamedPipe(
lpszPipeName,
    PIPE_ACCESS_DUPLEX,
    PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE |
    PIPE_WAIT,
    PIPE_UNLIMITED_INSTANCES,
512, 512, 5000, NULL);
```

Через создаваемый канал передаются сообщения (так как указана константа **PIPE_TYPE_MESSAGE**). Данная реализация предназначена только для чтения (константа **PIPE_READMODE_MESSAGE**).

При создании канала мы указали размер буферов ввода и вывода, равный 512 байт. Время ожидания операций выбрано равным пять секунд. Атрибуты защиты не указаны, поэтому используются значения, принятые по умолчанию.

Установка соединения с каналом со стороны сервера

После того как серверный процесс создал канал, он может перейти в режим соединения с клиентским процессом. Соединение со стороны сервера выполняется с помощью функции **ConnectNamedPipe**. Ее прототип представлен ниже:

```
BOOL ConnectNamedPipe(
HANDLE hNamedPipe,          // идентификатор именованного
                            // канала
LPOVERLAPPED lpOverlapped); // адрес структуры OVERLAPPED
```

Через первый параметр серверный процесс передает этой функции идентификатор канала, полученный от функции **CreateNamedPipe**.

Второй параметр используется только для организации асинхронного обмена данными через канал. Если вы используете только синхронные операции, в качестве значения для этого параметра можно указать **NULL**.

В случае успеха функция **ConnectNamedPipe** возвращает значение **TRUE**, а при ошибке – **FALSE**. Код ошибки можно получить с помощью функции **GelLastError**.

В зависимости от различных условий функция **ConnectNamedPipe** может вести себя по-разному. Если параметр **lpOverlapped** указан как **NULL**, функция выполняется в синхронном режиме. В противном случае используется асинхронный режим.

Для канала, созданного в синхронном блокирующем режиме (с использованием константы **PIPE_WAIT**), функция **ConnectNamedPipe** переходит в состояние ожидания соединения с клиентским процессом. Именно этот режим мы будем использовать в наших примерах программ, исходные тексты которых вы найдете ниже.

Для канала, созданного в синхронном неблокирующем режиме, функция **ConnectNamedPipe** немедленно возвращает управление с кодом **TRUE**, если только клиент был отключен от данной реализации канала и возможно подключение этого клиента. В противном случае возвращается значение **FALSE**. Дальнейший анализ необходимо выполнять с помощью функции **GetLastError**. Эта функция может вернуть значение **ERROR_PIPE_LISTENING** (если к серверу еще не подключен ни один клиент), **ERROR_PIPE_CONNECTED** (если клиент уже подключен) или **ERROR_NO_DATA** (если предыдущий клиент отключился от сервера, но клиент еще не завершил соединение).

Ниже приведен пример использования функции **ConnectNamedPipe**:

```
HANDLE hNamedPipe;
LPSTR  lpszPipeName = "\\\\.\\pipe\\$MyPipe$";
hNamedPipe = CreateNamedPipe(
lpszPipeName,
    PIPE_ACCESS_DUPLEX,
    PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE |
    PIPE_WAIT,
```

```
    PIPE_UNLIMITED_INSTANCES,
    512, 512, 5000, NULL);
fConnected = ConnectNamedPipe(hNamedPipe, NULL);
```

В данном случае функция **ConnectNamedPipe** перейдет в состояние ожидания, так как канал был создан для работы в синхронном блокирующем режиме.

Установка соединения с каналом со стороны клиента

Для создания канала клиентский процесс может воспользоваться функцией **CreateFile**. Эта функция предназначена для работы с файлами, однако с ее помощью можно также открыть канал, указав его имя вместо имени файла. Забегая вперед, скажем, что функция **CreateFile** позволяет открывать не только файлы или каналы, но и другие системные ресурсы, например устройства и каналы Mailslot.

В случае успешного завершения функция **CreateFile** возвращает идентификатор созданного или открытого файла (или каталога), а при работе с каналом – идентификатор реализации канала.

При ошибке возвращается значение **INVALID_HANDLE_VALUE** (а не **NULL**, как можно было бы предположить). Код ошибки можно определить при помощи функции **GetLastError**.

В том случае, если файл уже существует и были указаны константы **CREATE_ALWAYS** или **OPEN_ALWAYS**, функция **CreateFile** не возвращает код ошибки. В то же время в этой ситуации функция **GetLastError** возвращает значение **ERROR_ALREADY_EXISTS**.

Приведем фрагмент исходного текста клиентского приложения, открывающего канал с именем **$MyPipe$** при помощи функции **CreateFile**:

```
char   szPipeName[256];
HANDLE hNamedPipe;
strcpy(szPipeName, "\\\\.\\pipe\\$MyPipe$");
hNamedPipe = CreateFile(
szPipeName, GENERIC_READ | GENERIC_WRITE,
0, NULL, OPEN_EXISTING, 0, NULL);
```

Здесь канал открывается как для записи, так и для чтения.

Отключение серверного процесса от клиентского процесса

Если сервер работает с несколькими клиентскими процессами, то он может использовать для этого несколько реализаций канала, причем одни и те же реализации могут применяться по очереди.

Установив канал с клиентским процессом при помощи функции **ConnectNamedPipe**, серверный процесс может затем разорвать канал, вызвав для этого функцию **DisconnectNamedPipe**. После этого реализация канала может быть вновь использована для соединения с другим клиентским процессом.

Прототип функции **DisconnectNamedPipe** приведен ниже:

```
BOOL DisconnectNamedPipe(HANDLE hNamedPipe);
```

Через параметр **hNamedPipe** функции передается идентификатор реализации канала Pipe, полученный от функции **CreateNamedPipe**.

В случае успеха функция возвращает значение **TRUE**, а при ошибке – **FALSE**. Код ошибки можно получить от функции **GetLastError**.

Закрывание идентификатора канала

Если канал больше не нужен, после отключения от клиентского процесса серверный и клиентский процессы должны закрыть его идентификатор функцией **CloseHandle**:

```
CloseHandle(hNamedPipe);
```

Запись данных в канал

Запись данных в открытый канал выполняется с помощью функции **WriteFile**, аналогично записи в обычный файл:

```
HANDLE hNamedPipe;
DWORD  cbWritten;
char   szBuf[256];
WriteFile(hNamedPipe, szBuf, strlen(szBuf) + 1,
&cbWritten, NULL);
```

Через первый параметр функции **WriteFile** передается идентификатор реализации канала. Через второй параметр передается адрес

буфера, данные из которого будут записаны в канал. Размер этого буфера указывается при помощи третьего параметра. Предпоследний параметр используется для определения количества байтов данных, действительно записанных в канал. И наконец, последний параметр задан как **NULL**, поэтому запись будет выполняться в синхронном режиме.

Если канал был создан для работы в блокирующем режиме и функция **WriteFile** работает синхронно (без использования вывода с перекрытием), то эта функция не вернет управление до тех пор, пока данные не будут записаны в канал.

Чтение данных из канала

Как и следовало ожидать, для чтения данных из канала можно воспользоваться функцией **ReadFile**, например, так:

```
HANDLE hNamedPipe;
DWORD  cbRead;
char   szBuf[256];
ReadFile(hNamedPipe, szBuf, 512, &cbRead, NULL);
```

Данные, прочитанные из канала **hNamedPipe**, будут записаны в буфер **szBuf**, имеющий размер 512 байт. Количество действительно прочитанных байтов данных будет сохранено функцией **ReadFile** в переменной **cbRead**. Так как последний параметр функции указан как **NULL**, используется синхронный режим работы без перекрытия.

Другие функции

Среди других функций, предназначенных для работы с каналами, перечислим функции **CallNamedPipe**, **TransactNamedPipe**, **PeekNamedPipe**, **WaitNamedPipe**, **SetNamedPipeHandleState**, **GetNamedPipeInfo**, **GetNamedPipeHandleState**.

Обычно сценарий взаимодействия клиентского процесса с серверным заключается в выполнении следующих операций:

- подключение к каналу с помощью функции **CreateFile**;
- чтение или запись такими функциями как **ReadFile** или **WriteFile**;
- отключение от канала функцией **CloseHandle**.

Функция **CallNamedPipe** позволяет выполнить эти операции за один прием при условии, что канал открыт в режиме передачи сообщений и что клиент посылает одно сообщение серверу и в ответ также получает от сервера одно сообщение.

Функция **TransactNamedPipe**, как и функция **CallNamedPipe**, предназначена для выполнения передачи и приема данных от клиентского процесса серверному. Однако эта функция более гибкая в использовании, чем функция **CallNamedPipe**. Прежде всего перед использованием функции **TransactNamedPipe** клиентский процесс должен открыть канал с сервером, воспользовавшись для этого, например, функцией **CreateFile**. Кроме того, клиентский процесс может выполнять обмен данными с сервером, вызывая функцию **TransactNamedPipe** много раз. При этом не будет происходить многократного открытия и закрытия канала.

Чтение данных из канала функцией **ReadFile** вызывает удаление прочитанных данных. В противоположность этому функция **PeekNamedPipe** позволяет получить данные из именованного или анонимного канала без удаления, так что при последующих вызовах этой функции или функции **ReadFile** будут получены все те же данные, что и при первом вызове функции **PeekNamedPipe**. Еще одно отличие заключается в том, что функция **PeekNamedPipe** никогда не переходит в состояние ожидания, сразу возвращая управление вне зависимости от того, есть данные в канале или нет.

С помощью функции **WaitNamedPipe** процесс может выполнять ожидание момента, когда канал будет доступен для соединения:

```
BOOL WaitNamedPipe(
LPCTSTR lpszPipeName, // адрес имени канала Pipe
DWORD dwTimeout);     // время ожидания в миллисекундах
```

При необходимости можно изменить режимы работы для уже созданного канала. Для этого предназначена функция **SetNamedPipeHandleState**, прототип которой приведен ниже:

```
BOOL SetNamedPipeHandleState(
HANDLE hNamedPipe,      // идентификатор канала
LPDWORD lpdwMode,       // адрес переменной, в которой
                        // указан новый режим канала
```

```
LPDWORD lpcbMaxCollect, // адрес переменной, в которой
                        // указывается максимальный
                        // размер пакета, передаваемого
                        // в канал
LPDWORD lpdwCollectDataTimeout); //адрес максимальной
                       // задержки перед передачей данных
```

С помощью функции **GetNamedPipeHandleState** процесс может определить состояние канала, зная его идентификатор.

Еще одна функция, позволяющая получить информацию об именованном канале по его идентификатору, называется **GetNamedPipeInfo**.

Примеры приложений

Далее представлен пример из двух приложений, которые передают друг другу данные через именованный канал. Заметим, что приложения способны установить канал связи между различными рабочими станциями через сеть.

Первое из этих приложений называется **namedpipeserver**. Оно выполняет роль сервера, который получает команды от клиентского приложения **namedpipeclient** и отображает их в консольном окне.

Сразу после запуска приложение-сервер переходит в состояние ожидания соединения с клиентским приложением. При этом в его окне отображается строка **Waiting for connect**. При запуске клиентского приложения можно дополнительно указать параметр – имя рабочей станции, например:

```
>namedpipeclient gun
```

Если имя рабочей станции не указано, приложение будет пытаться установить канал с серверным приложением, запущенным на том же компьютере, что и клиентское.

Как только клиентское приложение установит канал связи с сервером, в окно серверного приложения выводятся строки **Connected** и **Waiting for command**. Команды, посылаемые серверу, также выводятся в его окне. Принятые команды посылаются сервером обратно клиентскому приложению и отображаются в его окне в строке **Receivedback**.

После успешного создания канала клиентское приложение выводит сообщение **`Connected`** и предлагает вводить команды в приглашении **`cmd>.`** В качестве команд можно вводить имена текстовых файлов, которые будут передаваться серверу, а в ответ клиент будет получать количество пробелов в них. При невозможности открыть указанный файл для чтения сервер будет возвращать клиенту сообщение об ошибке. Команда **`exit`** используется для завершения работы и клиентского и серверного приложений.

После того как команда посылается серверу, она возвращается обратно и отображается в окне клиентского приложения для контроля в строке **`Receivedback`**.

Исходные тексты приложений *namedpipeserver.cpp* и *namedpipeclient.cpp* приведены на сайте дисциплины в папке **Winpipes**.

Вопросы для самопроверки

1. Каковы особенности именованных каналов?
2. Каковы возможные форматы имен именованных каналов?
3. Каковы параметры функции создания именованного канала?
4. Каковы параметры функции установки соединения с каналом со стороны сервера?
5. Как производится установка соединения с каналом со стороны клиента?
6. Какие функции применяются для чтения и записи данных в именованные каналы?
7. Зачем нужна функция отключения сервера от канала?
8. Какие существуют дополнительные функции для работы с именованными каналами?

Упражнения

1. Написать алгоритм (с указанием основных функций и их параметров) для обмена данными через именованные каналы со стороны серверного процесса.
2. Написать алгоритм (с указанием основных функций и их параметров) для обмена данными через именованные каналы со стороны клиентского процесса.

6.2. Почтовые ящики (Mailslots)

Как и именованные каналы, почтовые ящики (Mailslots) Windows снабжаются именами, которые могут быть использованы для обеспечения взаимодействия между независимыми процессами. Почтовые ящики представляют собой широковещательный механизм, основанный на дейтаграммах (описаны в главе 8), и ведут себя иначе по сравнению с именованными каналами, что делает их весьма полезными в ряде ограниченных ситуаций, которые тем не менее представляют большой интерес. Из наиболее важных свойств почтовых ящиков можно отметить следующие.

- Почтовые ящики являются однонаправленными.
- С одним почтовым ящиком может быть связано несколько записывающих программ (writers) и несколько считывающих программ (readers), но они часто связаны между собой отношениями «один ко многим» в той или иной форме.
- Записывающей программе (клиенту) не известно достоверно, все ли, только некоторые или какая-то одна из программ считывания (сервер) получили сообщение.
- Почтовые ящики могут находиться в любом месте сети.
- Размер сообщений ограничен.

Использование почтовых ящиков требует выполнения следующих операций.

- Каждый сервер создает дескриптор почтового ящика с помощью функции **`CreateMailSlot`**.
- После этого сервер ожидает получения почтового сообщения, используя функцию **`ReadFile`**.
- Клиент, обладающий только правами записи, должен открыть почтовый ящик, вызвав функцию **`CreateFile`**, и записать сообщения, используя функцию **`WriteFile`**. В случае отсутствия ожидающих программ считывания попытка открытия почтового ящика завершится ошибкой (наподобие «имя не найдено»).

Сообщение клиента может быть прочитано *всеми* серверами; все серверы получают одно и то же сообщение.

Существует еще одна возможность. В вызове функции **`CreateFile`** клиент может указать имя почтового ящика в следующем виде:

```
\\*\mailslot\mailslotname
```

При этом символ звездочки (*) действует в качестве группового символа (wildcard) и клиент может обнаружить любой сервер в пределах *имени домена* – группы систем, объединенных общим именем, которое назначается администратором сети.

Использование почтовых ящиков

Сервер приложения (application server), действуя в качестве *почтового клиента* (mailslot client), периодически осуществляет широковещательную рассылку своего имени и имени канала. Любой *клиент приложения* (application client), которому требуется найти сервер, может получить это имя, действуя в качестве *сервера почтовых ящиков* (mailslot server). Это соответствует ситуации, в которой имеется одна записывающая программа (почтовый клиент) и несколько считывающих программ (почтовых серверов). Если бы почтовых клиентов (т. е. серверов приложения) было несколько, то ситуация описывалась бы отношением типа «многие ко многим».

Возможен и другой вариант, когда одна считывающая программа получает сообщения от многочисленных записывающих программ, которые, например, предоставляют информацию о своем состоянии. Этот вариант, соответствующий, например, электронной доске объявлений, оправдывает использование термина *почтовый ящик.* Оба описанных варианта использования – широковещательная рассылка имени и информации о состоянии – могут быть объединены, чтобы клиент мог выбирать наиболее подходящий сервер.

Обмен ролями терминов *клиент* и *сервер* в данном контексте может несколько сбивать с толку, однако заметьте, что почтовый сервер выполняет вызовы функции **`CreateMailSlot`**, тогда как клиент почтового ящика создает соединение, используя функцию **`CreateFile`**. Кроме того, в обоих случаях первый вызов функции **`WriteFile`** выполняется клиентом, а первый вызов функции **`ReadFile`** выполняется сервером.

Создание почтового ящика

Для создания почтового ящика и получения дескриптора, который можно будет использовать в операциях **`ReadFile`**, почтовые серверы (программы считывания) вызывают функцию **`CreateMailslot`**.

На одном компьютере может находиться только один почтовый ящик с данным именем, но один и тот же почтовый ящик может использоваться несколькими системами в сети, что обеспечивает возможность работы с ним нескольких программ считывания:

```
HANDLE CreateMailslot(LPCTSTR lpName, DWORD cbMaxMsg,
DWORD dwReadTimeout, LPSECURITY_ATTRIBUTES lpsa)
```

Здесь **lpName** – указатель на строку с именем почтового ящика, которая должна иметь следующий вид:

\\.\mailslot\[путь]*имя*

Имя должно быть уникальным. Точка «.» указывает на то, что почтовый ящик создается на локальном компьютере.

cbMaxMsg – максимальный размер сообщения (в байтах), которые может записывать клиент. Значению ноль соответствует отсутствие ограничений.

dwReadTimeOut – длительность интервала ожидания (в миллисекундах) для операции чтения. Значению ноль соответствует немедленный возврат, а значению **MAILSLOT_WAIT_FOREVER** – неопределенный период ожидания (который может длиться сколь угодно долго).

Во время открытия почтового ящика с помощью функции **CreateFile** клиент (записывающая программа) может указывать его имя в следующем виде:

- **\\ .\mailslot\ [путь]имя** – определяет локальный почтовый ящик;
- **\\имя_компьютера\mailslot\[путь]имя** – определяет почтовый ящик, расположенный на компьютере с заданным именем;
- **\\имя_домена\mailslot\[путь]имя** – определяет все почтовые ящики с данным именем, расположенные на компьютерах, принадлежащих данному домену. В этом случае максимальный размер сообщения составляет 424 байта;
- ***\mailslot\[путь]имя** – определяет все почтовые ящики с данным именем, расположенные на компьютерах, принадлежащих главному домену системы. В этом случае максимальный размер сообщения составляет 424 байта.

Средства, сопоставимые с почтовыми ящиками, в UNIX отсутствуют. Однако для этой цели могут быть использованы широковеща-

тельные (broadcast) или групповые (multicast) дейтаграммы протокола TCP/IP.

Параметр **lpSecurityAttributes** задает адрес структуры защиты, который в примерах будет задан как **NULL**.

При ошибке функцией **CreateMailslot** возвращается значение **INVALID_HANDLE_VALUE**. Код ошибки можно определить при помощи функции **GetLastError**.

Ниже приведен пример использования функции **CreateMailslot** в серверном приложении:

```
LPSTR lpszMailslotName =
"\\\\.\\mailslot\\$MailslotName$";
hMailslot = CreateMailslot(lpszMailslotName, 0,
MAILSLOT_WAIT_FOREVER, NULL);
```

В этом примере задается максимальный размер сообщения, поэтому на эту величину нет ограничений (кроме ограничения в 424 байта для сообщений, передаваемых всем компьютерам домена в широковещательном режиме).

Время ожидания указано как **MAILSLOT_WAIT_FOREVER**, поэтому функции, работающие с данным каналом Mailslot, будут работать в блокирующем режиме.

Открытие канала Mailslot

Прежде чем приступить к работе с каналом Mailslot, клиентский процесс должен его открыть. Для выполнения этой операции следует использовать функцию **CreateFile**, например, так:

```
LPSTR  lpszMailslotName =
"\\\\.\\mailslot\\$MailslotName$";
hMailslot = CreateFile( lpszMailslotName, GENERIC_WRITE,
FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
```

Здесь в качестве первого параметра функции **CreateFile** передается имя канала Mailslot. Заметим, что вы можете открыть канал Mailslot, созданный на другой рабочей станции в сети. Для этого строка

имени канала, передаваемая функции **CreateFile**, должна иметь следующий вид:

\\ИмяРабочейСтанции\mailslot\[Путь]ИмяКанала

Можно открыть канал для передачи сообщений всем рабочим станциям заданного домена. Для этого необходимо задать имя по следующему образцу:

\\ИмяДомена\mailslot\[Путь]ИмяКанала

Для передачи сообщений одновременно всем рабочим станциям сети первичного домена имя задается следующим образом:

***\mailslot\[Путь]ИмяКанала**

В качестве второго параметра функции **CreateFile** мы передаем константу **GENERIC_WRITE**. Эта константа определяет, что над открываемым каналом будет выполняться операция записи. Напомним, что клиентский процесс может только посылать сообщения в канал Mailslot, но не читать их оттуда. Чтение сообщений из канала Mailslot – задача для серверного процесса.

Третий параметр указан как **FILE_SHARE_READ**, и это тоже необходимо, так как сервер может читать сообщения, посылаемые одновременно несколькими клиентскими процессами.

Обратите также внимание на константу **OPEN_EXISTING**. Она используется потому, что функция **CreateFile** открывает существующий канал, а не создает новый.

Запись сообщений в канал Mailslot

Запись сообщений в канал Mailslot выполняет клиентский процесс, вызывая для этого функцию **WriteFile**:

```
HANDLE hMailslot;
char   szBuf[512];
DWORD  cbWritten;
WriteFile(hMailslot, szBuf, strlen(szBuf) + 1,
&cbWritten, NULL);
```

В качестве первого параметра этой функции необходимо передать идентификатор канала Mailslot, полученный от функции **CreateFile**.

Второй параметр определяет адрес буфера с сообщением, третий – размер сообщения. В нашем случае сообщения передаются в виде текстовой строки, закрытой двоичным нулем, поэтому для определения длины сообщения мы воспользовались функцией **strlen**.

Чтение сообщений из канала Mailslot

Серверный процесс может читать сообщения из созданного им канала Mailslot при помощи функции **ReadFile**, как это показано ниже:

```
HANDLE hMailslot;
char   szBuf[512];
DWORD  cbRead;
ReadFile(hMailslot, szBuf, 512, &cbRead, NULL);
```

Через первый параметр функции **ReadFile** передается идентификатор созданного ранее канала Mailslot, полученный от функции **CreateMailslot**. Второй и третий параметры задают соответственно адрес буфера для сообщения и его размер.

Заметим, что перед выполнением операции чтения следует проверить состояние канала Mailslot. Если в нем нет сообщений, то функцию **ReadFile** вызывать не следует. Для проверки состояния канала вы должны воспользоваться функцией **GetMailslotInfo**, описанной ниже.

Определение состояния канала Mailslot

Серверный процесс может определить текущее состояние канала Mailslot по его идентификатору с помощью функции **GetMailslotInfo**. Прототип этой функции мы привели ниже:

```
BOOL GetMailslotInfo(
HANDLE hMailslot,        // идентификатор канала Mailslot
LPDWORD lpMaxMessageSize,     // адрес максимального
                              // размера сообщения
LPDWORD lpNextSize,           // адрес размера следующего
                              // сообщения
```

```
LPDWORD lpMessageCount,    // адрес количества сообщений
LPDWORD lpReadTimeout);    // адрес времени ожидания
```

Через параметр **hMailslot** функции передается идентификатор канала Mailslot, состояние которого необходимо определить.

Остальные параметры задаются как указатели на переменные типа **DWORD**, в которые будут записаны параметры состояния канала Mailslot.

В переменную, адрес которой передается через параметр **lpMaxMessageSize**, после возвращения из функции **GetMailslotInfo** будет записан максимальный размер сообщения. Вы можете использовать это значение для динамического получения буфера памяти, в который это сообщение будет прочитано функцией **ReadFile**.

В переменную, адрес которой указан через параметр **lpNextSize**, записывается размер следующего сообщения, если оно есть в канале. Если же в канале больше нет сообщений, в эту переменную будет записана константа **MAILSLOT_NO_MESSAGE**.

С помощью параметра **lpMessageCount** вы можете определить количество сообщений, записанных в канал клиентскими процессами. Если это количество равно нулю, вам не следует вызывать функцию **ReadFile** для чтения несуществующего сообщения.

И наконец, в переменную, адрес которой задается в параметре **lpReadTimeout**, записывается текущее время ожидания, установленное для канала (в миллисекундах).

Если вам не нужна вся информация, которую можно получить с помощью функции **GetMailslotInfo**, некоторые из ее параметров (кроме, разумеется, первого) можно указать как **NULL**.

В случае успешного завершения функция **GetMailslotInfo** возвращает значение **TRUE**, а при ошибке – **FALSE**. Код ошибки можно получить, вызвав функцию **GetLastError**.

Ниже приведен пример использования функции **GetMailslotInfo**:

```
BOOL   fReturnCode;
DWORD  cbMessages;
DWORD  cbMsgNumber;
fRetCode = GetMailslotInfo(hMailslot, NULL, &cbMessages,
&cbMsgNumber, NULL);
```

Изменение состояния канала Mailslot

С помощью функции **SetMailslotInfo** серверный процесс может изменить время ожидания для канала Mailslot уже после его создания. Прототип функции **SetMailslotInfo** приведен ниже:

```
BOOL SetMailslotInfo(
HANDLE hMailslot,        // идентификатор канала Mailslot
DWORD dwReadTimeout);    // время ожидания
```

Через параметр **hMailslot** функции **SetMailslotInfo** передается идентификатор канала Mailslot, для которого нужно изменить время ожидания.

Новое значение времени ожидания в миллисекундах задается через параметр **dwReadTimeout**. Вы также можете указать здесь константы «0» или **MAILSLOT_WAIT_FOREVER**. В первом случае функции, работающие с каналом, вернут управление немедленно, во втором – будут находиться в состоянии ожидания до тех пор, пока не завершится выполняемая операция.

Примеры приложений

Для примера имеются исходные тексты двух приложений – **mslotserver** и **mslotclient**, которые обмениваются информацией через канал Mailslot. Заметим, что приложения способны установить канал связи между различными рабочими станциями через сеть.

Первое из этих приложений называется **mslotserver**. Оно выполняет роль сервера, который получает команды от клиентского приложения **mslotclient** и отображает их в консольном окне.

Сразу после запуска приложение-сервер переходит в состояние ожидания соединения с клиентским приложением. При этом в его окне отображается строка **Waiting for connect**. При запуске клиентского приложения можно дополнительно указать параметр – имя рабочей станции, например:

```
>mslotclient gun
```

Если имя рабочей станции не указано, приложение будет пытаться установить канал с серверным приложением, запущенным на том же компьютере, что и клиентское.

Далее выполняются действия, аналогичные описанным в пункте *Примеры приложений* раздела 6.1.2.

Исходные тексты приложений *mslotserver.cpp* и *mslotclient.cpp* приведены на сайте дисциплины в папке **Mailslots**.

Вопросы для самопроверки

1. Каковы особенности почтовых ящиков как средства межзадачной коммуникации?
2. Каковы параметры функции создания почтового ящика?
3. Каковы возможные форматы имен почтовых ящиков?
4. Каковы особенности создания почтового ящика на сервере?
5. Каковы особенности открытия почтового ящика клиентом?
8. Какие функции применяются для чтения и записи данных в почтовые ящики?
7. Как определить состояние почтового ящика?
8. Как и что можно изменить в состоянии почтового ящика?

Упражнения

1. Написать алгоритм (с указанием основных функций и их параметров) для обмена данными через почтовые ящики со стороны серверного процесса.
2. Написать алгоритм (с указанием основных функций и их параметров) для обмена данными через почтовые ящики со стороны клиентского процесса.

6.3. Средства синхронизации процессов

В Microsoft Windows, начиная с версии NT, для обмена данными между приложениями можно пользоваться относительно медленными механизмами DDE (Dynamic Data Exchange, динамический обмен данными), OLE (Object Linking and Embedding, связывание и встраивание объектов) и Clipboard (универсальный буфер обмена), однако прямое создание глобальных областей памяти, доступных всем приложениям, невозможно. Причина этого лежит в том, что адресные пространства 32-разрядных приложений, работающих под управлением операционных систем Microsoft Windows NT и выше, полностью изолированы.

Вместо этого можно использовать передачу данных между процессами, работающими в разных адресных пространствах, с использованием файлов, отображенных на память.

Методика использования файлов, отображенных на память, для передачи данных между процессами заключается в следующем.

Один из процессов создает такой файл, задавая при этом имя отображения. Это имя является глобальным и доступно для всех процессов, запущенных в системе. Другие процессы могут воспользоваться именем отображения, открыв созданный ранее файл. В результате оба процесса могут получить указатели на область памяти, для которой выполнено отображение, и эти указатели будут ссылаться на одни и те же страницы виртуальной памяти. Обмениваясь данными через эту область, процессы должны обеспечить синхронизацию своей работы, например, с помощью событий, мьютексов или семафоров (в зависимости от логики процесса обмена данными). События, мьютексы и семафоры являются объектами ядра, могут иметь имена и дескрипторы, поэтому активно применяются в многозадачных приложениях.

События используются для того, чтобы сигнализировать другим процессам/потокам о наступлении какого-либо события, например, о появлении нового сообщения. Важной дополнительной возможностью, обеспечиваемой объектами событий, является то, что переход в сигнальное состояние единственного объекта события способен вывести из состояния ожидания одновременно несколько процессов/потоков.

Мьютексы также можно использовать для синхронизации потоков, принадлежащих различным процессам. Так, два процесса, разделяющие общую память посредством отображения файлов, могут использовать мьютексы в своих потоках для синхронизации доступа к разделяемым областям памяти. Однако в каждый момент времени иметь права владельца мьютекса может только один поток. Соответственно рассмотрение мьютексов будет отложено до рассмотрения многопоточного программирования.

Для семафоров понятие владения неприменимо. Доступ разрешен одновременно нескольким потокам (в том числе из разных процессов), число которых ограничено максимальным значением счетчика.

6.3.1. События

События – самая примитивная разновидность объектов ядра. События содержат счетчик числа пользователей (как и все объекты ядра) и две булевы переменные: одна сообщает тип данного объекта-события, другая – его состояние (свободен или занят).

События используются для того, чтобы сигнализировать другим процессам/потокам о наступлении какого-либо события, например о появлении нового сообщения. Важной дополнительной возможностью, обеспечиваемой объектами событий, является то, что переход в сигнальное состояние единственного объекта события способен вывести из состояния ожидания одновременно несколько потоков. Объекты события делятся на сбрасываемые вручную и автоматически сбрасываемые, и это их свойство устанавливается при вызове функции **`CreateEvent`**.

- Сбрасываемые вручную события (manual-reset events) могут сигнализировать одновременно всем потокам, ожидающим наступления этого события, и переводятся в несигнальное состояние программно.
- Автоматически сбрасываемые события (auto-reset event) сбрасываются самостоятельно после освобождения одного из ожидающих потоков, тогда как другие ожидающие потоки продолжают ожидать перехода события в сигнальное состояние.

Схема использования событий достаточно проста. Один из процессов создает объект-событие, вызывая для этого функцию **`CreateEvent`**. При этом событие имеет имя, которое доступно всем потокам активных процессов. При создании или позже событие устанавливается в исходное состояние (отмеченное или неотмеченное).

Вызывая функции **`WaitForSingleObject`** или **`WaitForMultipleObjects`**, поток может выполнять ожидание момента, когда событие перейдет в отмеченное состояние.

Другой поток, принадлежащий тому же самому или другому процессу, может получить идентификатор события по его имени, например, с помощью функции **`OpenEvent`**. Далее, пользуясь функциями **`SetEvent`**, **`ResetEvent`** или **`PulseEvent`**, этот поток может изменить состояние события.

На рис. 12 приведен пример использования события для синхронизации двух задач, работающих одновременно. Представим себе, что первый поток занимается отображением данных, которые готовятся второй задачей небольшими порциями (например, читаются с диска).

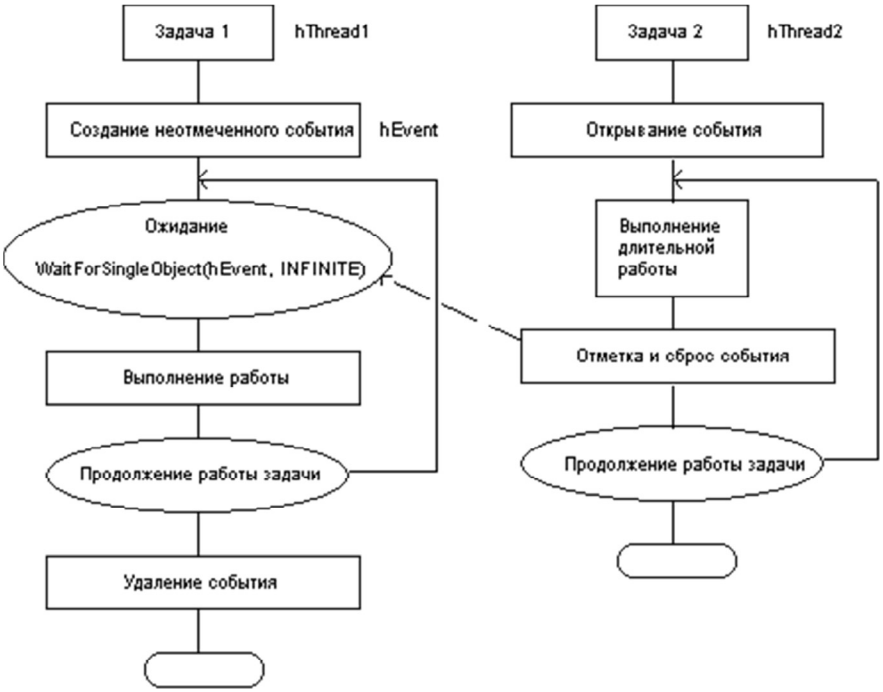

Рис. 12. Пример использования события для синхронизации двух потоков

После создания неотмеченного события первый поток переходит в состояние ожидания, пока второй поток не подготовит для нее данные. Как только это произойдет, второй поток отмечает и затем сбрасывает событие, что приводит к завершению ожидания первым процессом.

Отобразив подготовленные данные, первый поток опять входит в состояние ожидания, пока второй поток не подготовит данные и не отметит событие. Таким образом, два потока синхронизуют свою работу с помощью объекта-события.

Создание события

Для создания события задача должна вызваться функцией **CreateEvent**:

```
HANDLE CreateEvent(
LPSECURITY_ATTRIBUTES lpsa,     // атрибут безопасности
BOOL bManualReset,              // тип события
BOOL bInitialState,             // начальное состояние
LPTCSTR lpEventName);           // адрес глобального
                                // имени события
```

Чтобы создать событие, сбрасываемое вручную, необходимо установить значение параметра **bManualReset** равным **True**. Точно так же, чтобы сделать начальное состояние события сигнальным, установите равным **True** значение параметра **bInitialState**. В том случае, когда событие используется задачами, работающими в рамках одного процесса, имя события можно не задавать, указав параметр **lpEventName** как **NULL**. При этом создается безымянное событие.

В случае успешного завершения функция **CreateEvent** возвращает идентификатор события, которым нужно будет пользоваться при выполнении всех операций над объектом-событием. При ошибке возвращается значение **NULL** (код ошибки можно получить при помощи функции **GetLastError**).

Если при создании события указано имя уже существующего в системе события, созданного ранее другим процессом, то функция **GetLastError**, вызванная сразу после вызова функции **CreateEvent**, возвращает значение **ERROR_ALREADY_EXISTS**.

Открытие события

Для открытия именованного объекта события в другом процессе используется функция **OpenEvent**:

```
HANDLE OpenEvent(
  DWORD fdwAccess,    // флаги доступа
  BOOL fInhent,       // флаг наследования
LPCTSTR lpEventName);// адрес глобального имени события
```

Флаги доступа, передаваемые через параметр **fdwAccess**, определяют требуемый уровень доступа к объекту-событию. Этот параметр может быть комбинацией следующих значений (табл. 11).

Таблица 11

Флаги доступа к объекту-событию

Значение	Описание
EVENT_ALL_ACCESS	Указаны все возможные флаги доступа
EVENT_MODIFY_STATE	Полученный идентификатор можно будет использовать для функций **SetEvent** и **ResetEvent**
SYNCHRONIZE	Полученный идентификатор можно будет использовать в любых функциях ожидания события

Параметр **fInherit** определяет возможность наследования полученного идентификатора. Если этот параметр равен **TRUE**, идентификатор может наследоваться дочерними процессами. Если же он равен **FALSE**, наследование не допускается.

И наконец, через параметр **lpszEventName** необходимо передать функции адрес символьной строки, содержащей имя объекта-события.

Заметим, что с помощью функции **OpenEvent** несколько процессов могут открыть один и тот же объект-событие и затем выполнять одновременное ожидание для этого объекта.

Установка события

Для установки объекта-события в отмеченное состояние используется функция **SetEvent**:

```
BOOL SetEvent(HANDLE hEvent);
```

В качестве единственного параметра этой функции необходимо передать идентификатор объекта-события, полученного от функции **CreateEvent** или **OpenEvent**. При успешном завершении возвращается значение **TRUE**, при ошибке – **FALSE**. В последнем случае можно получить код ошибки при помощи функции **GetLastError**.

Сброс события

Сброс события (т. е. установка его в неотмеченное состояние) выполняется функцией **ResetEvent**:

```
BOOL ResetEvent(HANDLE hEvent);
```

Если задача создала событие, работающее в автоматическом режиме, оно будет сбрасываться и без помощи этой функции, если только какая-либо задача выполняла ожидание этого события и событие произошло.

Функция PulseEvent

Функция **PulseEvent** выполняет установку объекта-события в отмеченное состояние с последующим сбросом события в неотмеченное состояние:

```
BOOL PulseEvent(HANDLE hEvent);
```

Если эта функция вызвана для события, работающего в ручном режиме, то все задачи, ожидающие это событие, завершат ожидание и продолжат свою работу. Событие при этом будет установлено в неотмеченное состояние.

Для автоматических объектов-событий выполняются аналогичные действия, однако функция возвращает управление сразу, как только одна из ожидающих задач перейдет в активное состояние.

Примеры приложений

Для примера имеются исходные тексты двух приложений – **mfe_server** и **mfe_client** (mfe – mapping file with events), которые обмениваются информацией через отображаемый на память файл и синхронизируются событиями.

Первое из этих приложений (**mfe_server**) выполняет роль сервера, который получает команды от клиентского приложения **mfe_client** и отображает их в консольном окне. Сразу после запуска приложение-сервер создает отображение файла в виртуальной памяти и ожидает события записи в него команды от приложения-клиента.

Как только клиентское приложение запишет команду в отображаемый файл, имя отображения которого заранее известно, оно установит

соответствующее событие в сигнальное состояние. В качестве команд можно вводить имена текстовых файлов, которые будут передаваться серверу, а в ответ клиент будет получать количество пробелов в них. При невозможности открыть указанный файл для чтения сервер будет возвращать клиенту сообщение об ошибке. Команда **`exit`** используется для завершения работы и клиентского, и серверного приложений.

Сервер, дождавшись события, прочитает команду из отображения и выведет в своем окне. Результаты обработки команды будут записаны в отображение, после чего в сигнальное состояние будет переведено другое событие.

После получения второго события клиентское приложение выводит сообщение сервера – результат обработки файла или сообщение об ошибке. Команда **`exit`**, отправленная клиентом, устанавливает третье событие, которое используется для завершения работы серверного приложения.

Исходные тексты приложений *mfe_server.cpp* и *mfe_client.cpp* приведены на сайте дисциплины в папке **Mapping**.

Вопросы для самопроверки

1. Каковы особенности событий как средства синхронизации процессов?
2. Каковы параметры функции создания события сервером?
3. Каковы параметры функции открытия события клиентом?
4. Как осуществляется ожидание события приложением?
5. Когда и как применяются функции установки и сброса события?

Упражнения

1. Написать алгоритм (с указанием основных функций и их параметров) для обмена данными через отображаемые на память файлы при синхронизации событиями со стороны серверного процесса.
2. Написать алгоритм (с указанием основных функций и их параметров) для обмена данными через отображаемые на память файлы при синхронизации событиями со стороны клиентского процесса.

6.3.2. Семафоры

В отличие от других объектов IPC семафоры позволяют обеспечить доступ к ресурсу для заранее определенного, ограниченного приложением количества задач. Все остальные задачи, пытающиеся получить доступ сверх установленного лимита, будут переведены при этом в состояние ожидания до тех пор, пока какая-либо задача, получившая доступ к ресурсу раньше, не освободит ресурс, связанный с данным семафором.

Как и другие объекты IPC, семафор может находиться в отмеченном или неотмеченном состоянии. Приложение выполняет ожидание для семафора при помощи таких функций, как **WaitForSingleObject** или **WaitForMultipleObjects** (как и для объекта-события или мьютекса). Если семафор находится в неотмеченном состоянии, задача, вызвавшая для него функцию **WaitForSingleObject**, находится в состоянии ожидания. Когда же состояние семафора становится отмеченным, работа задачи возобновляется.

В отличие от других объектов IPC с каждым семафором связывается счетчик, начальное и максимальное значения которого задаются при создании семафора. Значение этого счетчика уменьшается, когда задача вызывает для семафора функцию **WaitForSingleObject** или **WaitForMultipleObjects**, и увеличивается при вызове другой, специально предназначенной для этого функции.

Если значение счетчика семафора равно нулю, он находится в неотмеченном состоянии. Если же это значение больше нуля, семафор переходит в отмеченное состояние.

Создание семафора

Для создания семафора приложение должно вызвать функцию **CreateSemaphore**, прототип которой приведен ниже:

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
                        // атрибуты защиты
  LONG lInitialCount,   // начальное значение счетчика
                        // семафора
LONG lMaximumCount,     // максимальное значение
                        // счетчика семафора
LPCTSTR lpName);     // адрес строки с именем семафора
```

В качестве атрибутов защиты можно передать значение **NULL**. Через параметры **lInitialCount** и **lMaximumCount** передаются соответственно начальное и максимальное значения счетчика, связанного с создаваемым семафором. Начальное значение счетчика **lInitialCount** должно быть больше или равно нулю и не должно превосходить максимального значения счетчика, передаваемого через параметр **lMaximumCount**.

Имя семафора указывается аналогично имени рассмотренного ранее объекта Event с помощью параметра **lpName**.

В случае удачного создания семафора функция **CreateSemaphore** возвращает его идентификатор. В случае возникновения ошибки возвращается значение **NULL**, при этом код ошибки можно узнать при помощи функции **GetLastError**.

Так как имена семафоров доступны всем приложениям в системе, возможно возникновение ситуации, когда приложение пытается создать семафор с уже использованным именем. При этом новый семафор не создается, а приложение получает идентификатор для уже существующего семафора. Если возникла такая ситуация, функция **GetLastError**, вызванная сразу после функции **CreateSemaphore**, возвращает значение **ERROR_ALREADY_EXISTS**.

Уничтожение семафора

Для уничтожения семафора необходимо передать его идентификатор функции **CloseHandle**. Заметим, что при завершении процесса все созданные им семафоры уничтожаются автоматически.

Открывание семафора

Если семафор используется только для синхронизации задач, созданных в рамках одного приложения, вы можете создать безымянный семафор, указав в качестве параметра **lpName** функции **CreateSemaphore** значение **NULL**. В том случае, когда необходимо синхронизовать задачи разных процессов, следует определить имя семафора. При этом один процесс создает семафор с помощью функции **CreateSemaphore**, а второй открывает его, получая идентификатор для уже существующего семафора.

Существующий семафор можно открыть функцией **OpenSemaphore**, прототип которой приведен ниже:

```
HANDLE OpenSemaphore(
  DWORD   fdwAccess,           // требуемый доступ
  BOOL    fInherit,            // флаг наследования
  LPCTSTR lpszSemaphoreName ); // адрес имени семафора
```

Флаги доступа, передаваемые через параметр **fdwAccess**, определяют требуемый уровень доступа к семафору. Этот параметр может быть комбинацией следующих значений (табл. 12).

Таблица 12

Флаги доступа к объекту-семафору

Значение	Описание
SEMAPHORE_ALL_ ACCESS	Указаны все возможные флаги доступа
SEMAPHORE_MODIFY_ STATE	Возможно изменение значение счетчика семафора функцией **ReleaseSemaphore**
SYNCHRONIZE	Полученный идентификатор можно будет использовать в любых функциях ожидания события

Параметр **fInherit** определяет возможность наследования полученного идентификатора. Если этот параметр равен **TRUE**, идентификатор может наследоваться дочерними процессами. Если же он равен **FALSE**, наследование не допускается.

Через параметр **lpszSemaphoreName** необходимо передать функции адрес символьной строки, содержащей имя семафора.

Если семафор открыт успешно, функция **OpenSemaphore** возвращает его идентификатор. При ошибке возвращается значение **NULL**. Код ошибки можно определить при помощи функции **GetLastError**.

Увеличение значения счетчика семафора

Для увеличения значения счетчика семафора приложение должно использовать функцию **ReleaseSemaphore**:

```
BOOL ReleaseSemaphore(
HANDLE hSemaphore,          // идентификатор семафора
LONG cReleaseCount,         // значение инкремента
LPLONG lplPreviousCount); // адрес переменной для записи
                            // предыдущего значения
                            // счетчика семафора
```

Функция **ReleaseSemaphore** увеличивает значение счетчика семафора, идентификатор которого передается ей через параметр **hSemaphore**, на значение, указанное в параметре **cReleaseCount**.

Заметим, что через параметр **cReleaseCount** можно передавать только положительное значение, большее нуля. При этом если в результате увеличения новое значение счетчика должно будет превысить максимальное значение, заданное при создании семафора, функция **ReleaseSemaphore** возвращает признак ошибки и не изменяет значение счетчика.

Предыдущее значение счетчика, которое было до использования функции **ReleaseSemaphore**, записывается в переменную типа **LONG**. Адрес этой переменной передается функции через параметр **lplPreviousCount**.

Если функция **ReleaseSemaphore** завершилась успешно, она возвращает значение **TRUE**. При ошибке возвращается значение **FALSE**. Код ошибки в этом случае можно определить как обычно, при помощи функции **GetLastError**.

Функция используется обычно для решения двух задач.

Во-первых, с помощью этой функции задачи освобождают ресурс, доступ к которому регулируется семафором. Они могут делать это после использования ресурса или перед своим завершением.

Во-вторых, эта функция может быть использована на этапе инициализации мультизадачного приложения. Создавая семафор с начальным значением счетчика, равным нулю, главная задача блокирует работу задач, выполняющих ожидание этого семафора. После завершения инициализации главная задача с помощью функции **ReleaseSema-**

phore может увеличить значение счетчика семафора до максимального, в результате чего известное количество ожидающих задач будет активизировано.

Уменьшение значения счетчика семафора

В программном интерфейсе ОС Microsoft Windows нет функции, специально предназначенной для уменьшения значения счетчика семафора. Этот счетчик уменьшается, когда задача вызывает функции ожидания, такие как **WaitForSingleObject** или **WaitForMultipleObjects**. Если задача вызывает несколько раз функцию ожидания для одного и того же семафора, содержимое его счетчика каждый раз будет уменьшаться.

Определение текущего значения счетчика семафора

Единственная возможность определения текущего значения счетчика семафора заключается в увеличении этого значения функцией **ReleaseSemaphore**. Значение счетчика, которое было до увеличения, будет записано в переменную, адрес которой передается функции **ReleaseSemaphore** через параметр **lplPreviousCount**.

Заметим, что в ОС Microsoft Windows не предусмотрено средств, с помощью которых можно было бы определить текущее значение семафора, не изменяя его. В частности, нельзя задать функции **ReleaseSemaphore** нулевое значение инкремента, так как в этом случае указанная функция просто вернет соответствующий код ошибки.

Примеры приложений

Для примера имеются исходные тексты двух приложений – *mfs_server* и *mfs_client* (mfs – mapping file with semaphores), которые обмениваются информацией через отображаемый на память файл и синхронизируются событиями.

Первое из этих приложений (*mfs_server*) выполняет роль сервера, который получает команды от клиентского приложения *mfs_client* и отображает их в консольном окне. Сразу после запуска приложение-сервер создает отображение файла в виртуальной памяти и ожидает события записи в него команды от приложения-клиента.

Далее выполняются действия, аналогичные описанным в пункте *Примеры приложений* раздела 6.3.1.

Исходные тексты приложений *mfs_server.cpp* и *mfs_client.cpp* приведены на сайте дисциплины в папке **Mapping**.

Вопросы для самопроверки

1. Каковы особенности семафоров как средства синхронизации процессов?
2. Каковы параметры функция создания семафора сервером?
3. Каковы параметры функция открытия семафора клиентом?
4. Как осуществляется ожидание семафора приложением?
5. Когда и как применяются функции увеличения и уменьшения значения семафора?

Упражнения

1. Написать алгоритм (с указанием основных функций и их параметров) для обмена данными через отображаемые на память файлы при синхронизации семафорами со стороны серверного процесса.
2. Написать алгоритм (с указанием основных функций и их параметров) для обмена данными через отображаемые на память файлы при синхронизации семафорами со стороны клиентского процесса.

6.3.3. Мьютексы

Если необходимо обеспечить последовательное использование ресурсов задачами, созданными в рамках разных процессов, вместо событий и семафоров возможно использовать объекты синхронизации Mutex (мьютекс). Свое название они получили от выражения «mutually exclusive», что означает «взаимно исключающий».

Как и объект-событие, объект мьютекс может находиться в отмеченном или неотмеченном состоянии. Когда какая-либо задача (поток), принадлежащая любому процессу, становится владельцем объекта мьютекс, последний переключается в неотмеченное состояние. Если же задача «отказывается» от владения объектом мьютекс, его состояние становится отмеченным.

Организация последовательного доступа к ресурсам с использованием объектов мьютекс возможна потому, что в каждый момент только

одна задача может владеть этим объектом. Все остальные задачи (для того чтобы завладеть объектом, который уже захвачен) должны ждать, например, с помощью известной функции **WaitForSingleObject**.

Для того чтобы объект мьютекс был доступен задачам, принадлежащим различным процессам, при создании необходимо присвоить ему имя (аналогично тому, как это делалось для объекта-события).

Создание объекта Mutex

Для создания объекта мьютекс используется функция **CreateMutex**:

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
                                  // атрибуты защиты
  BOOL bInitialOwner,             // начальное состояние
  LPCTSTR lpName);                // имя объекта Mutex
```

В качестве первого параметра (атрибуты защиты) можно указать значение **NULL** (как и во всех других примерах).

Параметр **bInitialOwner** определяет начальное состояние объекта мьютекс. Если он имеет значение **TRUE**, задача, создающая объект мьютекс, будет им владеть сразу после создания. Если же значение этого параметра равно **FALSE**, после создания объект мьютекс не будет принадлежать ни одной задаче, пока не будет захвачен какой-либо из них явным образом.

Через параметр **lpName** необходимо передать указатель на имя объекта мьютекс, для которого действуют те же правила, что и для имени объекта-события. Это имя не должно содержать символа ‘\’ и его длина не должна превышать значения **MAX_PATH**.

Если объект мьютекс будет использован только задачами одного процесса, вместо адреса имени можно указать значение **NULL**. В этом случае будет создан «безымянный» объект мьютекс.

Функция **CreateMutex** возвращает идентификатор созданного объекта мьютекс или **NULL** при ошибке.

Возможно возникновение такой ситуации, когда приложение пытается создать объект мьютекс с именем, которое уже используется в системе другим объектом мьютекс. В этом случае функция **Cre-**

ateMutex вернет идентификатор существующего объекта мьютекс, а функция **GetLastError**, вызванная сразу после вызова функции **CreateMutex**, вернет значение **ERROR_ALREADY_EXISTS**. Заметим, что функция создания объектов-событий **CreateEvent** ведет себя в данной ситуации аналогичным образом.

Освобождение идентификатора объекта Mutex

Если объект Mutex больше не нужен, необходимо освободить его идентификатор при помощи универсальной функции **CloseHandle**. Заметим тем не менее, что при завершении процесса освобождаются идентификаторы всех объектов Mutex, созданных для этого процесса.

Открытие объекта Mutex

Зная имя объекта Mutex, задача может его открыть с помощью функции **OpenMutex**:

```
HANDLE OpenMutex(
  DWORD   fdwAccess, // требуемый доступ
  BOOL    fInherit,  // флаг наследования
  LPCTSTR lpszMutexName ); // адрес имени объекта Mutex
```

Флаги доступа, передаваемые через параметр **fdwAccess**, определяют требуемый уровень доступа к объекту Mutex. Этот параметр может быть комбинацией следующих значений (табл. 13).

Таблица 13

Флаги доступа к объекту-мьютексу

Значение	Описание
MUTEX_ALL_ACCESS	Указаны все возможные флаги доступа
SYNCHRONIZE	Полученный идентификатор можно будет использовать в любых функциях ожидания события

Параметр **fInherit** определяет возможность наследования полученного идентификатора. Если этот параметр равен **TRUE**, идентификатор может наследоваться дочерними процессами. Если же он равен **FALSE**, наследование не допускается.

Через параметр **lpszEventName** в функцию передается адрес символьной строки, содержащей имя объекта Mutex.

С помощью функции **OpenMutex** несколько задач могут открыть один и тот же объект Mutex и затем выполнять одновременное ожидание для этого объекта.

Как завладеть объектом Mutex

Зная идентификатор объекта Mutex, полученный от функций **CreateMutex** или **OpenMutex**, задача может завладеть объектом при помощи функций ожидания событий, например, при помощи функций **WaitForSingleObject** или **WaitForMultipleObjects**.

Напомним, что функция **WaitForSingleObject** возвращает управление, как только идентификатор объекта, передаваемый ей в качестве параметра, перейдет в отмеченное состояние. Если объект Mutex не принадлежит ни одной задаче, его состояние будет отмеченным.

Когда происходит вызов функции **WaitForSingleObject** для объекта Mutex, который никому не принадлежит, она сразу возвращает управление. При этом задача, вызвавшая функцию **WaitForSingleObject**, становится владельцем объекта Mutex. Если теперь другая задача вызовет функцию **WaitForSingleObject** для этого же объекта Mutex, то она будет переведена в состояние ожидания до тех пор, пока первая задача не «откажется от своих прав» на данный объект Mutex. Освобождение объекта Mutex выполняется функцией **ReleaseMutex**.

Освобождение объекта Mutex

Для отказа от владения объектом Mutex (т. е. для его освобождения) необходимо использовать функцию **ReleaseMutex**:

```
BOOL ReleaseMutex(HANDLE hMutex);
```

Через единственный параметр этой функции необходимо передать идентификатор объекта Mutex. Функция возвращает значение **TRUE** при успешном завершении и **FALSE** – при ошибке.

Рекурсивное использование объектов Mutex

Объекты Mutex допускают рекурсивное использование. Задача может выполнять рекурсивные попытки завладеть одним и тем же объектом Mutex и при этом она не будет переводиться в состояние ожидания.

В случае рекурсивного использования каждому вызову функции ожидания должен соответствовать вызов функции освобождения объекта Mutex **`ReleaseMutex`**.

Примеры приложений

Для примера имеются исходные тексты двух приложений – *mfm_server* и *mfm_client* (mfm – mapping file with semaphores), которые обмениваются информацией через отображаемый на память файл и синхронизируются событиями.

Первое из этих приложений (*mfm_server*) выполняет роль сервера, который получает команды от клиентского приложения *mfm_client* и отображает их в консольном окне. Сразу после запуска приложение-сервер создает отображение файла в виртуальной памяти и ожидает события записи в него команды от приложения-клиента.

Как только клиентское приложение запишет команду в отображаемый файл, имя отображения которого заранее известно, оно освободит соответствующий мьютекс.

Далее выполняются действия, аналогичные описанным в пункте *Примеры приложений* раздела 6.3.1.

Исходные тексты приложений *mfm_server.cpp* и *mfm_client.cpp* приведены на сайте дисциплины в папке **Mapping**.

Вопросы для самопроверки

1. Каковы особенности мьютексов как средства синхронизации процессов?
2. Каковы параметры функции создания мьютекса сервером?
3. Каковы параметры функции открытия мьютекса клиентом?
4. Как осуществляется ожидание мьютекса приложением?
5. Когда и как применяются функции захвата и освобождения мьютекса?
6. Что такое «покинутый мьютекс»? Каковы его особенности?

Упражнения

1. Написать алгоритм (с указанием основных функций и их параметров) для обмена данными через отображаемые на память файлы при синхронизации мьютексами со стороны серверного процесса.

2. Написать алгоритм (с указанием основных функций и их параметров) для обмена данными через отображаемые на память файлы при синхронизации мьютексами со стороны клиентского процесса.

7. МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ В WINDOWS

Потоки, принадлежащие одному процессу, разделяют общие данные и код, поэтому важно, чтобы каждый поток имел также собственную область памяти, относящуюся только к нему. В Windows удовлетворение этого требования обеспечивается несколькими способами.

- У каждого потока имеется собственный стек, который он использует при вызове функций и обработке некоторых данных.
- При создании потока вызывающий процесс может передать ему аргумент (Arg на рис. 13), который обычно является указателем. На практике этот аргумент помещается в стек потока.

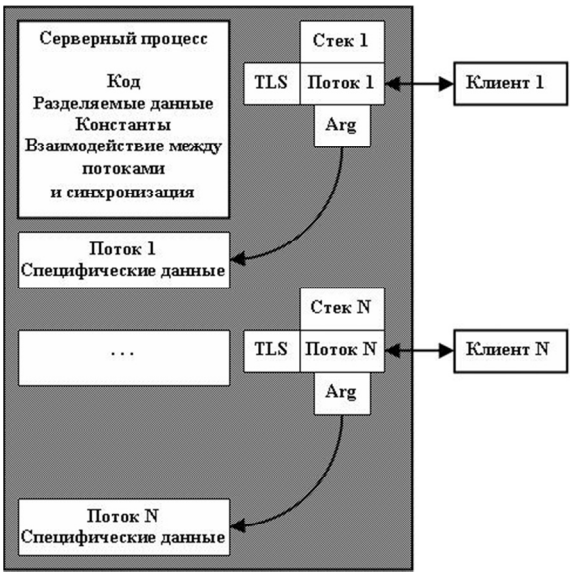

Рис. 13. Потоки в среде сервера

• Каждый поток может распределять индексы собственных локальных областей хранения (TLS), считывать и устанавливать значения TLS. TLS предоставляют в распоряжение потоков небольшие массивы данных. TLS обеспечивают защиту данных одного потока от воздействия со стороны других потоков.

Создание потоков

Для создания потоков в адресном пространстве процесса предусмотрен системный вызов **CreateThread**. Он требует указания:

• начального адреса потока в коде процесса;

• размера стека (из виртуального адресного пространства процесса) по умолчанию равен размеру стека основного потока;

• указателя на аргумент, передаваемый потоку.

Функция возвращает значение идентификатора (ID) и дескриптор потока. В случае ошибки возвращаемое значение равно **NULL**:

```
HANDLE CreateThread (
LPSECURITY_ATTRIBUTES lpsa,
DWORD dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddr,
LPVOID lpThreadParm,
DWORD dwCreationFlags,
LPDWORD lpThreadId);
```

Параметры

lpsa – указатель на структуру атрибутов защиты.

dwStackSize – размер стека нового потока в байтах. Значению «0» соответствует размер стека по умолчанию.

lpStartAddr – указатель на функцию (в коде процесса), которая должна выполняться. Функция принимает единственный аргумент в виде указателя и возвращает 32-битовый код завершения. Функция потока (**ThreadFunc**) имеет следующую сигнатуру:

```
DWORD WINAPI ThreadFunc (LPVOID);
```

lpThreadParm – указатель, передаваемый в функцию потока.

dwCreationFlags – если значение этого параметра установлено равным нулю, то поток запускается сразу после вызова функции **CreateThread**. Установка значения **CREATE_SUSPENDED** приведет к запуску потока в приостановленном состоянии, из которого поток может быть переведен в состояние готовности вызовом функции **ResumeThread**.

lpThreadId – указатель на переменную типа **DWORD**, которая получает идентификатор нового потока.

Завершение потоков

Поток процесса может завершить свое выполнение, вызвав функцию **ExitThread:**

```
VOID ExitThread (DWORD dwExitCode);
```

но обычным способом завершения потока является возврат из функции потока с использованием кода завершения в качестве возвращаемого значения. По завершении выполнения потока память, занимаемая его стеком, освобождается. Когда завершается выполнение последнего потока, завершается и выполнение процесса.

Выполнение потока также может быть завершено другим потоком с помощью функции **TerminateThread**, однако освобождения ресурсов потока при этом не происходит.

Поток, выполнение которого было завершено, продолжает существовать до тех пор, пока посредством функции **CloseHandle** не будет закрыт его последний дескриптор. Любой другой поток может получить код завершения потока: **BOOL GetExitCodeThread (HANDLE hThread, LPDWORD lpExitCode)**. Если поток еще не завершен, то значение этой переменной будет равно **STILL_ACTIVE.**

Дополнительные функции

Функции, используемые для получения идентификаторов и дескрипторов потоков, напоминают те, которые используются для аналогичных целей при взаимодействии процессов:

- **GetCurrentThread** – возвращает ненаследуемый дескриптор вызывающего потока;
- **GetCurrentThreadId** – возвращает идентификатор потока;

• **GetThreadId** — возвращает идентификатор потока по дескриптору;

• **OpenThread** — создает дескриптор потока по идентификатору.

Для каждого потока поддерживается *счетчик приостановок* (suspend count), и выполнение потока может быть продолжено, если значение этого счетчика равно нулю. Поток может увеличивать или уменьшать значение счетчика приостановок другого потока функциями **SuspendThread** и **ResumeThread**:

```
DWORD ResumeThread (HANDLE hThread);
DWORD SuspendThread (HANDLE hThread)
```

В случае успешного выполнения обе функции возвращают предыдущее значение счетчика приостановок, иначе – **0xFFFFFFFF**.

Ожидание завершения потоков

Поток может дожидаться завершения выполнения другого потока. При вызове функций ожидания (**WaitForSingleObject** и **WaitForMultipleObjects**) следует использовать дескрипторы потоков.

Допустимое количество объектов, одновременно ожидаемых функцией **WaitForMultipleObjects**, ограничено значением **MAXIMUM_WAIT_OBJECTS** (64), но при большом количестве потоков можно воспользоваться серией вызовов функций ожидания.

Функция ожидания дожидается, пока объект, указанный дескриптором, не перейдет в *сигнальное* состояние. Поток переводится в сигнальное состояние при помощи функций **ExitThread** и **TerminateThread**. Функция **ExitProcess** переводит в сигнальное состояние как сам процесс, так и все его потоки.

Вопросы для самопроверки

1. В чем отличие процессов и потоков?
2. Каковы особенности функции создания потока?
3. Какими функциями производятся завершение и прекращение выполнения потока, получение кода его завершения?
4. Какие существуют дополнительные функции для работы с потоками?

Упражнение

Написать алгоритм (с указанием основных функций и их параметров) работы многопоточного приложения без синхронизации потоков.

7.1. Средства синхронизации потоков в Windows

Windows предоставляет четыре объекта, предназначенных для синхронизации потоков и процессов. Три из них – мьютексы, семафоры и события – являются объектами ядра, имеющими дескрипторы. Четвертый объект – критические участки кода. В силу своей простоты и предоставляемых ими преимуществ в отношении производительности объекты критических участков кода являются предпочтительным механизмом, если их возможностей достаточно для удовлетворения требований программиста.

В то же время неправильное применение объектов критических участков кода порождает риски, такие, например, как риск взаимной блокировки потоков. Мьютексы (из-за того, что могут принадлежать захватившим их потокам) уменьшают риски взаимных блокировок. Кроме того, мьютекс всегда можно сделать покинутым, прекратив выполнение захватившего его потока.

Примеры приложений

Пример синхронизации потоков с применением мьютексов представлен в программе *mthreads.cpp*, доступной на сайте дисциплины в папке **WinThreads**.

Вопросы для самопроверки

1. Какие объекты Windows можно применять для синхронизации потоков?
2. Как можно осуществлять обмен данными между потоками?
3. Каковы особенности мьютексов как средства синхронизации потоков?

7.2. Критические участки кода

Объект критического участка кода – это участок программного кода, который каждый раз должен выполняться только одним потоком; параллельное выполнение этого участка несколькими потоками может приводить к непредсказуемым или неверным результатам.

Объекты **CRITICAL_SECTION** (CS) можно инициализировать и удалять, но они не имеют дескрипторов и не могут совместно использоваться другими процессами. Объекты должны объявляться как переменные типа **CRITICAL_SECTION**. Потоки входят в объекты CS и покидают их, но выполнение кода отдельного объекта CS каждый раз разрешено только одному потоку. Вместе с тем один и тот же поток может входить в несколько отдельных объектов CS и покидать их, если они расположены в разных местах программы.

Для инициализации и удаления переменной типа **CRITICAL_SECTION** используются соответственно следующие функции:

```
VOID InitializeCriticalSection (
LPCRITICAL_SECTION lpCriticalSection);
VOID DeleteCriticalSection (
LPCRITICAL_SECTION lpCriticalSection);
```

Функция **EnterCriticalSection** блокирует поток, если на данном критическом участке кода присутствует другой поток. Ожидающий поток разблокируется после того как другой поток выполнит функцию **LeaveCriticalSection**. Говорят, что поток *получил права владения* объектом CS, если произошел возврат из функции **EnterCriticalSection**, тогда как для уступки прав владения используется функция **LeaveCriticalSection**:

```
VOID EnterCriticalSection (
LPCRITICAL_SECTION lpCriticalSection);
VOID LeaveCriticalSection (
LPCRITICAL_SECTION lpCriticalSection);
```

Поток, владеющий объектом CS, может повторно войти в этот же CS без его блокирования; таким образом, объекты CS являются *рекурсивными*. Поддерживается счетчик вхождений в объект CS, и поэтому поток должен покинуть CS столько раз, сколько было вхождений

в него, чтобы разблокировать этот объект для других потоков. Выход из объекта CS, которым данный поток не владеет, может привести к непредсказуемым результатам, включая блокирование самого потока.

Для возврата из функции **EnterCriticalSection** нет конечного интервала ожидания; другие потоки будут блокированы на неопределенное время, пока поток, владеющий объектом CS, не покинет его. Используя функцию **TryEnterCriticalSection**, можно тестировать (опросить) CS, чтобы проверить, не владеет ли им другой поток:

```
BOOL TryEnterCriticalSection
(LPCRITICAL_SECTION lpCriticalSection)
```

Возврат функцией **TryEnterCriticalSection** значения **True** означает, что вызывающий поток приобрел права владения критическим участком кода, тогда как возврат значения **False** говорит о том, что данный критический участок кода уже принадлежит другому потоку.

Объекты CS обладают тем преимуществом, что они не являются объектами ядра и поддерживаются в пользовательском пространстве.

Одним из наиболее распространенных способов применения объектов CS является обеспечение доступа потоков к разделяемым глобальным переменным. Рассмотрим пример:

```
CRITICAL_SECTION csl;
Volatile DWORD N = 0, M;
/* N – глобальная переменная */
InitializeCriticalSection (&csl);
EnterCriticalSection (&csl);
if (N < N_MAX) { M = N; M += 1; N = M; }
LeaveCriticalSection (&csl);
DeleteCriticalSection (&csl);
```

Примеры приложений

Полный пример применения критических секций реализован в программе *csthreads.cpp*, доступной на сайте в папке **WinThreads**.

Вопросы для самопроверки

1. Каковы особенности критических секций как средства синхронизации потоков?
2. Как инициализируются и удаляются критические секции?
3. Как поток приобретает и уступает права владения критической секцией?
4. Как проверить, не владеет ли критической секцией другой поток?

Упражнения

1. Написать алгоритм (с указанием основных функций и их параметров) работы многопоточного приложения с синхронизацией через мьютексы.
2. Написать алгоритм (с указанием основных функций и их параметров) работы многопоточного приложения с синхронизацией через критические секции.

8. СЕТЕВОЕ ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ WINDOWS

Именованные каналы пригодны для организации межпроцессного взаимодействия как в случае процессов, выполняющихся на одной и той же системе, так и в случае процессов, выполняющихся на компьютерах, связанных друг с другом локальной или глобальной сетью. Однако как именованные каналы, так и почтовые ящики обладают тем недостатком, что они не являются промышленным стандартом. Это обстоятельство усложняет перенос программ в системы, не принадлежащие семейству Windows, хотя именованные каналы не зависят от протоколов и могут выполняться поверх многих стандартных промышленных протоколов, например TCP/IP.

Возможность взаимодействия с другими системами обеспечивается в Windows поддержкой сокетов (sockets) Windows Sockets – совместимого и почти точного аналога сокетов Berkeley Sockets, де-факто играющих роль промышленного стандарта. Использование API Windows Sockets (или Winsock) позволяет серверам и клиентам принимать и отправлять запросы приложениям UNIX или каких-либо других, отличных от Windows систем.

Поскольку интерфейс Winsock должен соответствовать промышленным стандартам, принятые в нем соглашения о правилах присвоения имен и стилях программирования несколько отличаются от тех, с которыми мы сталкивались в процессе работы с описанными ранее функциями Windows. Строго говоря [6], Winsock API не является частью Win32/64. Кроме того, Winsock предоставляет дополнительные функции, не подчиняющиеся стандартам; эти функции используются лишь в случае крайней необходимости. Среди других преимуществ, обеспечиваемых Winsock, следует отметить улучшенную переносимость результирующих программ на другие системы.

Сокеты, датаграммы и каналы связи

В локальных и глобальных сетях существует два принципиально разных способа передачи данных.

Первый из них предполагает посылку пакетов данных от одного узла другому (или сразу нескольким узлам) без получения подтверждения о доставке и даже без гарантии того, что передаваемые пакеты будут получены в правильной последовательности. Примером такого протокола может служить протокол UDP (User Datagram Protocol), который используется в сетях TCP/IP, или протокол IPX, который является базовым в сетях Novell NetWare.

Основные преимущества датаграмных протоколов заключаются в высоком быстродействии и возможности широковещательной передачи данных, когда один узел отправляет сообщения, а другие их получают, причем все одновременно.

Второй способ передачи данных предполагает создание канала передачи данных между двумя различными узлами сети. При этом канал создается средствами датаграммных протоколов, однако доставка пакетов в канале является гарантированной. Пакеты всегда доходят в целости и сохранности, причем в правильном порядке, хотя быстродействие получается в среднем ниже за счет посылки подтверждений. Примерами протоколов, использующих каналы связи, могут служить протоколы **TCP** и **SPX** (протокол **NETBIOS** допускает передачу данных с использованием как датаграмм, так и каналов связи).

Для передачи данных с использованием любого из перечисленных выше способов каждое приложение должно создать объект, который называется сокетом.

По своему назначению сокет больше всего похож на идентификатор файла **(file handle)**, который нужен для выполнения над файлом операций чтения или записи. Прежде чем приложение, запущенное на узле сети, сможет выполнять передачу или прием данных, оно должно создать сокет и проинициализировать его, указав некоторые параметры.

Для сокета необходимо указать три параметра. Это IP-адрес, связанный с сокетом, номер порта, для которого будут выполняться операции передачи данных, а также тип сокета. Существуют сокеты двух типов. Первый тип предназначен для передачи данных в виде датаграмм, второй – с использованием каналов связи.

Сокеты Windows

Winsock API разрабатывался как расширение Berkley Sockets API для среды Windows, поэтому поддерживается всеми системами Windows. К преимуществам Winsock можно отнести следующее:

- перенос уже имеющегося кода, написанного для Berkeley Sockets API, осуществляется непосредственно;
- системы Windows легко встраиваются в сети, использующие как версию IPv4 протокола TCP/IP, так и постепенно распространяющуюся версию IPv6;
- сокеты можно рассматривать как дескрипторы (типа **`HANDLE`**) файлов при использовании функций **`ReadFile`** и **`WriteFile`** и, с некоторыми ограничениями, при использовании других функций, точно так же, как в качестве дескрипторов файлов сокеты применяются в UNIX. Эта возможность оказывается удобной в тех случаях, когда требуется использование асинхронного ввода/вывода;
- существуют также дополнительные, непереносимые расширения.

Инициализация Winsock

Winsock API поддерживается библиотекой DLL (WS2_32.DLL), для получения доступа к которой следует подключить к программе библиотеку WS_232.LIB. В среде VisualStudio подключение выполняется через команды меню *Свойства проекта → Компоновщик → Ввод → Дополнительные зависимости:ws2_32.lib*. В заголовке программы необходимо также указать подключение (**`#include`**) файла <**`winsock2.h`**>, причем до подключения файла <**`windows.h`**>!

Эту DLL следует инициализировать с помощью нестандартной, специфической для Winsock функции **`WSAStartup`**, которая должна быть первой из функций Winsock, вызываемых программой:

```
int WSAStartup(WORD wVersionRequired,
        LPWSADATA ipWSAData);
```

Здесь параметр **`wVersionRequired`** указывает старший номер версии библиотеки DLL, который требуется использовать. Младший байт параметра **`wVersionRequired`** указывает основной номер версии, а старший байт – дополнительный. Рекомендуется использовать

версия Winsock 2.0; **ipWSAData** – указатель на структуру **WSADATA**, которая возвращает информацию о конфигурации DLL, включая старший доступный номер версии. Пример инициализации библиотеки выглядит так:

```
WSAStartup(0x202, (WSADATA *)&WS));
```

Функция возвращает ненулевое значение, если запрошенная версия данной DLL не поддерживается. Чтобы получить более подробную информацию об ошибках, можно воспользоваться функцией **WSAGetLastError**.

Когда необходимость в использовании функциональных возможностей Winsock отпадает, следует вызывать функцию **WSACleanup**:

```
WSACleanup();
```

чтобы библиотека WS_32.DLL, обслуживающая сокеты, могла освободить ресурсы, распределенные для этого процесса.

Префикс WSA в именах перечисленных выше функций означает «Windows Sockets asynchronous…» («Асинхронный Windows Sockets…»). Средства асинхронного режима Winsock рассматриваться здесь не будут.

Создание сокета

Инициализировав Winsock DLL, можно использовать стандартные (Berkeley Sockets) функции для создания сокетов и соединений, обеспечивающих взаимодействие серверов с клиентами или взаимодействие равноправных узлов сети между собой.

Используемый в Winsock тип данных **SOCKET** аналогичен типу данных **HANDLE** в Windows, и его даже можно применять совместно с функцией **ReadFile** и другими функциями Windows, требующими использования дескрипторов типа **HANDLE**. Для создания (или открытия) сокета служит функция socket:

```
SOCKET socket(int af, int type, int protocol);
```

Параметр **af** обозначает семейство адресов, или протокол; для указания протокола IP (компонент протокола TCP/IP, отвечающий за протокол Internet) следует использовать значение **PF_INET** (или **AF_INET**, которое имеет то же самое числовое значение, но обычно используется при вызове функции **bind**).

Параметр **type** указывает тип взаимодействия: ориентированное на установку соединения (connection-oriented communication), или потоковое (**SOCK_STREAM**), и дейтаграммное (datagram communication) (**SOCK_DGRAM**), что в определенной степени сопоставимо соответственно с именованными каналами и почтовыми ящиками.

Параметр **protocol** является излишним; если параметр **af** установлен равным **AF_INET**, то можно использовать значение ноль.

В случае неудачного завершения функция **socket** возвращает значение **INVALID_SOCKET**.

Winsock можно использовать совместно с протоколами, отличными от TCP/IP, указывая различные значения параметра **protocol**; далее же будет использован только протокол TCP/IP.

Как и в случае всех остальных стандартных функций, имя функции **socket** не должно содержать прописных букв. Это является отходом от соглашений, принятых в Windows, и продиктовано необходимостью соблюдения промышленных стандартов.

Пример создания сокета:

```
SOCKET sockTCP;
sockTCP = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

Серверные функции сокета

В следующем ниже обсуждении под *сервером* будет пониматься процесс, который принимает запросы на образование соединения через заданный порт. Несмотря на то что сокеты, подобно именованным каналам, могут использоваться для создания соединений между равноправными узлами сети, введение указанного различия между узлами является весьма удобным и отражает различия в способах, используемых обеими системами для соединения друг с другом.

Связывание сокета

Следующий шаг заключается в привязке сокета к его адресу и *конечной точке* (endpoint) (направление канала связи от приложения к службе). Вызов **socket**, за которым следует вызов **bind**, аналогичен созданию именованного канала. Однако не существует имен, используя которые можно было бы различать сокеты данного компьютера.

Вместо этого в качестве конечной точки службы используется *номер порта* (port number). Любой заданный сервер может иметь несколько конечных точек. Прототип функции **bind** приводится ниже:

```
int bind(SOCKET s, const struct sockaddr *saddr,
                   int namelen);
```

Здесь **s** – несвязанный сокет, возвращенный функцией **socket**, структура **saddr** заполняется перед вызовом и задает протокол и специфическую для протокола информацию, как описано ниже. Кроме всего прочего, в этой структуре содержится номер порта. Параметр **namelen** должен быть равен значению **sizeof (sockaddr)**.

В случае успешного выполнения функция возвращает значение «0», иначе – **SOCKET_ERROR**. Структура **sockaddr** определяется следующим образом:

```
Struct sockaddr {
u_short sa_family;
char sa_data[14] ;
};
```

Первый член этой структуры, **sa_family**, обозначает протокол. Второй член, **sa_data**, зависит от протокола. Internet-версией структуры **sa_data** является структура **sockaddr_in**:

```
Struct sockaddr_in {
short sin_family; /* AF_INET */
u_short sin_port;
struct in_addr sin_addr; /* 4-байтовый IP-адрес */
char sin_zero[8];
};
```

Обратите внимание на использование типа данных **short integer** для номера порта. Кроме того, номер порта и иная информация должны храниться с соблюдением сетевого порядка следования байтов, при котором старший байт помещается в крайней позиции справа (big-endian), чтобы обеспечивалась двоичная совместимость с другими системами. Для смены порядка следования байтов из обычного (порядок хоста, little-endian) в сетевой формат (big-endian) применяются функции **htons()** (Host toNetwork short) и **htonl()** (Host toNetwork

long). Обратное преобразование выполняется соответственно функциями **ntohs()**, **ntohl()**.

В структуре **sin_addr** содержится подструктура **s_addr**, заполняемая 4-байтовым IP-адресом, например 127.0.0.1, указывающим системе, чей запрос на образование соединения должен быть принят. Обычно удовлетворяются запросы любых систем, в связи с чем следует использовать значение **INADDR_ANY**.

Датаграммный протокол UDP позволяет посылать пакеты данных одновременно всем рабочим станциям в широковещательном режиме. Для этого вы должны указать адрес как **INADDR_BROADCAST**.

Для преобразования текстовой строки с конкретным IP-адресом к требуемому формату можно использовать функцию **inet_addr**, поэтому член **sin_addr.s_addr** переменной **sockaddr_in** инициализируется следующим образом:

```
sa.sin_addr.s_addr = inet_addr("192.168.0.1");
```

В случае ошибки функция возвращает значение **INADDR_NONE**, что можно использовать для проверки. Обратное преобразование адреса IP в текстовую строку можно при необходимости легко выполнить с помощью функции **inet_ntoa**, имеющей следующий прототип:

```
char* inet_ntoa (struct in_addr in);
```

При ошибке эта функция возвращает значение **NULL**.

Однако чаще всего пользователь работает не с адресами, а с доменными именами, используя сервер DNS. В этом случае вначале необходимо воспользоваться функцией **gethostbyname**, возвращающей адрес IP, а затем записать полученный адрес в структуру **sin_addr:**

```
HOSTENT  hst;
hst = gethostbyname ("ftp.microsoft.com");
if(hst)
memcpy((char *)&(dest_sin.sin_addr ), hst ->h_addr , hst
->h_length);
```

В случае ошибки функция **gethostbyname** возвращает **NULL**. При этом причину ошибки можно выяснить, проверив код возврата функции **WSAGetLastError**.

Если же указанный узел найден в базе DNS, функция **gethostbyname** возвращает указатель на структуру **hostent**. Искомый адрес находится в этой структуре в первом элементе списка **h_addr_list[0]**, на который можно также ссылаться при помощи **h_addr**. Длина поля адреса находится в поле **h_length**:

```
structhostent
{
char  *h_name;              // имя узла
char** h_aliases;           // список альтернативных имен
short h_addr type;          // тип адреса узла
short h_length;             // длина адреса
char  ** h_addr _list;      // список адресов
#define h_addr   h_addr_list[0]     // адрес
};
```

О связанном сокете, для которого определены протокол, номер порта и IP-адрес, иногда говорят как об именованном сокете (named socket).

Пример связывания сокета:

```
sockaddr_in local_addr;
local_addr.sin_family=AF_INET;    // задание системы
                                  // адресации
local_addr.sin_port=htons(PORT); // задние порта
local_addr.sin_addr.s_addr=INADDR_ANY;  // подключения
                                     // со всех IP-адресов
bind(socket,(sockaddr *)
&local_addr,sizeof(local_addr));
```

Перевод связанного сокета в состояние прослушивания

Функция **listen** делает сервер доступным для образования соединения с клиентом (аналогичной функции для именованных каналов не существует):

```
Int listen(SOCKET s, int nQueueSize);
```

Параметр **nQueueSize** указывает число запросов на соединение, которые можно помещать в очередь сокета. В версии Winsock 2.0

значение этого параметра не имеет ограничения сверху, но в версии 1.1 оно ограничено предельным значением **SOMAXCON** (равным пяти).

Прием клиентских запросов соединения

Сервер может ожидать соединения с клиентом, используя функцию **accept**, возвращающую новый подключенный сокет, который будет использоваться в операциях ввода/вывода. Заметьте, что исходный сокет, который теперь находится в состоянии прослушивания (listening state), используется исключительно в качестве параметра функции **accept**, а не для непосредственного участия в операциях ввода/вывода.

Функция **accept** блокируется до тех пор, пока от клиента не поступит запрос соединения, после чего она возвращает новый сокет ввода/вывода (возможно создание неблокирующихся сокетов, но рассмотрение этого выходит за рамки данной книги):

```
SOCKET accept(SOCKET s, LPSOCKADDR lpAddr,
              LPINT lpAddrLen);
```

Параметр **s** – прослушивающий сокет. Чтобы перевести сокет в состояние прослушивания, необходимо предварительно вызвать функции **socket**, **bind** и **listen**.

lpAddr – указатель на структуру **sockaddr_in**, предоставляющую адрес клиентской системы.

lpAddrLen – указатель на переменную, которая будет содержать размер возвращенной структуры **sockaddr_in**. Перед вызовом функции **accept** эта переменная должна быть инициализирована значением **sizeof(struct sockaddr_in)**.

Отключение и закрытие сокетов

Для отключения сокетов применяется функция **shutdown(s, how)**. Аргумент **how** может принимать одно из двух значений: «1», указывающее на то, что соединение может быть разорвано только для посылки сообщений, и «2», соответствующее разрыву соединения как для посылки, так и для приема сообщений. Функция **shutdown** не освобождает ресурсы, связанные с сокетом, но гарантирует завершение посылки и приема всех данных до закрытия сокета. Тем не менее после вызова функции **shutdown** приложение уже не должно использовать этот сокет.

Когда работа с сокетом закончена, его следует закрыть, вызвав функцию **closesocket(SOCKET s)**. Сначала сервер закрывает сокет, созданный функцией **accept**, а не прослушивающий сокет, созданный с помощью функции **socket**. Сервер должен закрывать прослушивающий сокет только тогда, когда завершает работу или прекращает принимать клиентские запросы соединения. Даже если вы работаете с сокетом как с дескриптором типа **HANDLE** и используете функции **ReadFile** и **WriteFile**, уничтожить сокет одним только вызовом функции **CloseHandle** вам не удастся; для этого следует использовать функцию **closesocket**.

Клиентские функции сокета

Клиентская станция, которая желает установить соединение с сервером, также должна создать сокет, вызвав функцию **socket**. Следующий шаг заключается в установке соединения сервером, а кроме того, необходимо указать номер порта, адрес хоста и другую информацию. Имеется только одна дополнительная функция – **connect**:

```
Int connect(SOCKET s, LPSOCKADDR lpName, int nNameLen);
```

Ее параметр **s** – это сокет, созданный с использованием функции **socket**.

lpName – указатель на структуру **sockaddr_in**, инициализированную значениями номера порта и IP-адреса серверной системы с сокетом, связанным с указанным портом, который находится в состоянии прослушивания.

Параметр **nNameLen** необходимо инициализировать значением **sizeof (struct sockaddr_in)**.

Возвращаемое значение «0» указывает на успешное завершение функции, тогда как значение **SOCKET_ERROR** указывает на ошибку, которая, в частности, может быть обусловлена отсутствием прослушивающего сокета по указанному адресу.

Сокет **s** не обязательно должен быть связанным с портом до вызова функции **connect**, хотя это и может иметь место. При необходимости система распределяет порт и определяет протокол.

Пример подключения клиента к серверу, адрес и порт которого заданы через аргументы командной строки программы:

```
sockaddr_in server_addr;
  ZeroMemory(&server_addr, sizeof(server_addr));
  server_addr.sin_family = AF_INET;
  server_addr.sin_addr.s_addr = inet_addr(argv[1]);
  server_addr.sin_port = htons(atoi(argv[2]));
  if (connect(sockTCP, (sockaddr *)&server_addr,
  sizeof(server_addr)) == SOCKET_ERROR)
    printf("Client can NOT connect socket to Server,
    error: %d\n", WSAGetLastError());
```

Отправка и получение данных

Программы, использующие сокеты, обмениваются данными с помощью функций **send** и **recv**, прототипы которых почти совпадают (перед указателем буфера функции **send** помещается модификатор **const**). Ниже представлен только прототип функции **send**:

```
int send(SOCKET s, const char * lpBuffer,
     int nBufferLen, int nFlags);
```

Возвращаемым значением является число фактически переданных байтов. Значение **SOCKET_ERROR** указывает на ошибку.

Параметр **nFlags** может использоваться для обозначения степени срочности сообщений (например, для экстренных сообщений – **MSG_OOB** (Out Of Band)), а значение **MSG_PEEK** позволяет просматривать получаемые данные без их считывания.

Самое главное это то, что функции **send** и **recv** *не являются атомарными* (atomic), поэтому нет никакой гарантии, что затребованные данные будут действительно отправлены или получены. Передача «коротких» отправлений (short sends) встречается крайне редко, хотя и возможна, что справедливо и по отношению к приему «коротких» получений (short receive). Понятие сообщения в том смысле, который оно имело в случае именованных каналов, здесь отсутствует, поэтому необходимо проверять возвращаемое значение и повторно отправлять или принимать данные до тех пор, пока все они не будут переданы.

С сокетами могут использоваться также функции **ReadFile** и **WriteFile**, только в этом случае при вызове функции необходимо привести сокет к типу **HANDLE**.

Дейтаграммный сокет (типа **SOCK_DGRAM**) также может пользоваться функциями **send** и **recv**, если предварительно вызовет **connect**, но у него есть и свои, «персональные», функции: **int sendto (SOCKET s, const char * buf, int len, int flags, const struct sockaddr * to, int tolen)** и **int recvfrom (SOCKET s, char * buf, int len, int flags, struct sockaddr * from, int * fromlen)**.

Они очень похожи на **send** и **recv** – разница лишь в том, что **sendto** и **recvfrom** требуют явного указания адреса узла, принимающего или передающего данные. Вызов **recvfrom** не требует предварительного задания адреса передающего узла – функция принимает все пакеты, приходящие на заданный UDP-порт со всех IP-адресов и портов. Напротив, отвечать отправителю следует на тот же самый порт, откуда пришло сообщение. Поскольку, функция **recvfrom** запоминает IP-адрес и номер порта клиента после получения от него сообщения, программисту нужно передать в **sendto** тот же самый указатель на структуру **sockaddr**, который был ранее передан функции **recvfrom**, получившей сообщение от клиента.

Еще одна деталь: транспортный протокол UDP, на который опираются дейтаграммные сокеты, не гарантирует успешной доставки сообщений, и эта задача ложится на плечи самого разработчика. Решить ее можно, например, посылкой клиентом подтверждения об успешности получения данных.

Во всем остальном обе пары функций полностью идентичны и работают с флагами **MSG_PEEK** и **MSG_OOB**.

Все четыре функции при возникновении ошибки возвращают значение **SOCKET_ERROR**.

Сравнение именованных каналов и сокетов

Именованные каналы, описанные ранее, очень похожи на сокеты, но в способах их использования имеются значительные различия.

- Именованные каналы могут быть ориентированными на работу с сообщениями, что значительно упрощает программы.
- Именованные каналы требуют использования функций **ReadFile** и **WriteFile**, в то время как сокеты могут обращаться также к функциям **send** и **recv**.

• В отличие от именованных каналов сокеты настолько гибки, что предоставляют пользователям возможность выбрать протокол для использования с сокетом, например, TCP или UDP. Кроме того, пользователь имеет возможность выбирать протокол на основании характера предоставляемой услуги или иных факторов.

• Сокеты основаны на промышленном стандарте, что обеспечивает их совместимость с системами, отличными от Windows.

Имеются также различия в моделях программирования сервера и клиента. Установка соединения с несколькими клиентами при использовании сокетов требует выполнения повторных вызовов функции **accept**. Каждый из вызовов возвращает очередной подключенный сокет. По сравнению с именованными каналами имеются следующие отличия.

• В случае именованных каналов требуется, чтобы каждый экземпляр именованного канала и дескриптор типа **HANDLE** создавались с помощью функции **CreateNamedPipe**, тогда как для создания экземпляров сокетов применяется функция **accept**.

• Допустимое количество клиентских сокетов ничем не ограничено (функция **listen** ограничивает лишь количество клиентов, помещаемых в очередь), в то время как количество экземпляров именованных каналов, в зависимости от того, что было указано при первом вызове функции **CreateNamedPipe**, может быть ограниченным.

• Не существует вспомогательных функций для работы с сокетами, аналогичных функции **TransactNamedPipe**.

• Именованные каналы не имеют портов с явно заданными номерами и различаются по именам.

В случае сервера именованных каналов получение пригодного для работы дескриптора типа **HANDLE** требует вызова двух функций (**CreateNamedPipe** и **ConnectNamedPipe**), тогда как сервер сокета требует вызова четырех функций (**socket**, **bind**, **listen** и **accept**).

В случае клиента именованных каналов необходимо последовательно вызывать функции **WaitNamedPipe** и **CreateFile**. Если же используются сокеты, этот порядок вызовов инвертируется, поскольку можно считать, что функция **socket** создает сокет, а функция **connect** – блокирует. Дополнительное отличие состоит в том, что функция **connect** является функцией клиента сокета, в то время как функция **ConnectNamedPipe** используется сервером именованного канала.

Примеры приложений

Примеры применения сокетов, реализующие клиент-серверную систему с использованием протоколов TCP и UDP, доступны на сайте дисциплины в папке **Winsockets**.

Вопросы для самопроверки

1.Какие основные сетевые протоколы поддерживаются библиотекой WinSock?

2. Как производится инициализация библиотеки Winsock?

3. Как происходит создание сокета?

4. Что такое «связывание» сокета?

5. Что означает сетевой порядок байтов и порядок хоста?

6. Какие существуют функции для преобразования IP-адреса?

7. Какие существуют функции для работы с именами DNS?

8. Какие функции должно вызвать серверное приложение для ожидания подключения клиентов?

9. В чем разница отключения и закрытия сокета?

10. Какие функции должно вызвать клиентское приложение для подключения к серверу?

11. Какие функции можно использовать для получения и отправки данных через сокеты?

12. Чем отличается взаимодействие процессов через сокеты от использования именованных каналов и почтовых ящиков?

Упражнения

1. Написать алгоритм (с указанием основных функций и их параметров) работы серверного приложения по протоколу TCP.

2. Написать алгоритм (с указанием основных функций и их параметров) работы клиентского приложения по протоколу TCP.

3. Написать алгоритм (с указанием основных функций и их параметров) работы серверного приложения по протоколу UDP.

4. Написать алгоритм (с указанием основных функций и их параметров) работы клиентского приложения по протоколу UDP.

9. СИСТЕМНЫЕ СЛУЖБЫ WINDOWS

Помимо обычных процессов, в операционной системе Microsoft Windows NT создаются так называемые сервисные процессы или службы (services). Эти процессы могут стартовать автоматически при загрузке операционной системы, по запросу приложений или других сервисов, а также в ручном режиме.

Функции, выполняемые сервисами, могут быть самыми разнообразными: от обслуживания аппаратуры и программных интерфейсов до серверов приложений, таких, например, как серверы баз данных или серверы World Wide Web (WWW).

Информация о всех серверах, установленных в системе, хранится в регистрационной базе данных. Ниже приведем путь к этой информации:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Servicies
```

Для просмотра и редактирования регистрационной базы данных можно воспользоваться приложением regedit.exe, которое находится в каталоге установки Windows. Однако в программном интерфейсе WIN32 имеется набор функций, специально предназначенных для работы с записями регистрационной базы данных, имеющих отношение к сервисным процессам.

Чтобы просмотреть список установленных сервисов, можно запустить приложение «Службы» из папки «Администрирование». В нем отображается список установленных сервисов, а при выборе свойств какого-либо сервиса – его текущее состояние и режим запуска (ручной или автоматический).

Если выбран тип запуска «Automatic» («Автоматический»), то сервис будет запускаться автоматически при загрузке операционной системы. Этот режим удобен для тех сервисов, которые нужны постоянно, например, для сервера базы данных. При этом сервер базы данных

будет запускаться автоматически без участия оператора, что очень удобно.

При выборе типа запуска «Manual» («Вручную») сервис будет запускаться в ручном режиме. Пользователь может запустить сервис, нажав кнопку «Start» («Запустить») в окне свойств сервиса. Другое приложение или другой сервис также может запустить этот сервис при помощи специальной функции программного интерфейса WIN32. Эту функцию рассмотрим позже.

И наконец, если выбран тип запуска «Disabled» («Отключен»), работа сервиса будет заблокирована.

Сервис может работать с привилегиями выбранных пользователей или с привилегиями системы («LocalSystem»). Для выбора имени пользователя необходимо включить переключатель «This Account» («С учетной записью») и выбрать по нажатию кнопки «Обзор», расположеной справа от этого переключателя, имя пользователя. Дополнительно в полях «Password» («Пароль») и «Confirm Password» («Подтверждение») необходимо ввести пароль пользователя.

Если включить переключатель «System Account» («С системной учетной записью»), сервис будет работать с привилегиями системы. Если сервис будет взаимодействовать с программным интерфейсом рабочего стола, следует включить переключатель «Allow Service to Interact with Desktop» («Разрешить взаимодействие с рабочим столом»).

Сервисы могут быть двух типов: стандартные сервисы и сервисы, соответствующие протоколам драйверов устройств Microsoft Windows. Последние описаны в документации DDK и здесь не рассматриваются.

При запуске операционной системы Microsoft Windows автоматически стартует специальный процесс, который называется процессом управления сервисами (Service Control Manager, SCM). В программном интерфейсе WIN32 имеются функции, с помощью которых приложения и сервисы могут управлять работой сервисов, обращаясь к процессу управления сервисами. Некоторые из этих функций будут рассмотрены далее.

Преобразование консольного приложения в службу (сервис) Windows осуществляется в три этапа, после выполнения которых программа переходит под управление SCM.

1. Создание новой точки входа **`main()`**, которая регистрирует службу в SCM, предоставляя точки входа и имена логических служб.

2. Преобразование прежней функции точки входа **main()** в функцию **ServiceMain()**, которая регистрирует обработчик управляющих команд службы и информирует SCM о своем состоянии. Остальная часть кода, по существу, сохраняет прежний вид, хотя и может быть дополнена командами регистрации событий. Имя **ServiceMain()** является заменителем имени логической службы, причем логических служб может быть несколько.

3. Написание функции обработчика управляющих команд службы, которая должна предпринимать определенные действия в ответ на команды, поступающие от SCM.

Функция main сервисного процесса

Задачей новой функции **main()**, которая вызывается SCM, является регистрация службы в SCM и запуск диспетчера службы (service control dispatcher). Для этого необходимо вызвать функцию **StartServiceControlDispatcher()**, передав ей имя (имена) и точку (точки) входа одной или нескольких логических служб.

По сути, логические службы являются усовершенствованными версиями основной программы, преобразуемой в службу, и каждая логическая служба будет активизироваться в собственном экземпляре SCM. Например, логические службы на основе почтовых ящиков и именованных каналов могут выполняться в рамках одной и той же службы Windows, что потребует предоставления основных функций обеих служб.

Прототип функции регистрации логических служб:

```
BOOL StartServiceCtrlDispatcher
(LPSERVICE_TABLE_ENTRY lpServiceStartTable)
```

Эта функция принимает единственный аргумент **lpServiceStartTable**, являющийся адресом массива элементов **SERVICE_TABLE_ENTRY**, каждый из которых представляет имя и точку входа логической службы. Конец массива обозначается двумя последовательными значениями **NULL**.

Тип **SERVICE_TABLE_ENTRY** и соответствующий указатель определены следующим образом:

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

В поле **lpServiceName** записывается указатель на текстовую строку имени сервиса, а в поле **lpServiceProc** – указатель на точку входа сервиса.

Функция возвращает значение **TRUE**, если регистрация службы прошла успешно. Если служба уже выполняется или возникают проблемы с обновлением записей реестра (**HKEY_LOCAL_MACHINE\ SYSTEM\ CurrentControlSet\Services**), функция завершается с ошибками, обработка которых может осуществляться обычным путем, при помощи функции **GetLastError**.

Получив управление, функция **StartServiceCtrlDispatcher** не возвращает его до тех пор, пока все сервисы, запущенные в рамках данного процесса, не завершат свою работу. Заметим, однако, что фактического запуска логических служб в этот момент не происходит; запуск службы требует вызова функции **StartService**, которая описывается далее.

Рассмотрим пример:

```
#define MYServiceName "Sample of simple service"
void main(int agrc, char *argv[])
{
  SERVICE_TABLE_ENTRY DispatcherTable[] =
  {
    {
      MYServiceName,
(LPSERVICE_MAIN_FUNCTION)ServiceMain
    },
    {
      NULL, NULL
    }
  };
if(!StartServiceCtrlDispatcher(DispatcherTable))
  {
fprintf(stdout,"StartServiceCtrlDispatcher:
Error %ld\n",
```

```
GetLastError());
getch();
return;
  }
}
```

Точка входа сервиса

Точка входа сервиса – это функция, адрес которой записывается в поле **lpServiceProc** массива структур **SERVICE_TABLE_ENTRY**. Имя функции может быть любым, а прототип должен быть таким, как показанный ниже:

```
void WINAPI ServiceMain(DWORD dwArgc,
        LPSTR *lpszArgv);
```

Точка входа сервиса вызывается при запуске сервиса функцией **StartService** (эта функция рассмотрена далее). Через параметр **dwArgc** передается счетчик аргументов, а через параметр **lpszArgv** – указатель на массив строк параметров. В качестве первого параметра всегда передается имя сервиса. Остальные параметры можно задать при запуске сервиса функцией **StartService**.

Несмотря на то что функция **ServiceMain** является адаптированным вариантом функции **main** с ее параметрами, представляющими количество аргументов и содержащую их строку, между ними имеется одно незначительное отличие: функция службы должна быть объявлена с типом **void**, а не иметь возвращаемое значение типа **int**, как в случае обычной функции **main**.

Функция точки входа сервиса должна зарегистрировать функцию обработки команд и выполнить инициализацию сервиса.

Первая задача решается с помощью функции **RegisterServiceCtrlHandler**, прототип которой приведен ниже:

```
SERVICE_STATUS_HANDLE RegisterServiceCtrlHandler(
  LPCTSTR lpszServiceName,              // имя сервиса
  LPHANDLER_FUNCTION lpHandlerProc);    // адрес функции
                                        // обработки команд
```

Через первый параметр этой функции необходимо передать адрес текстовой строки имени сервиса, а через второй – адрес функции обработки команд (функция обработки команд будет рассмотрена ниже).

Вот пример использования функции **RegisterServiceCtrlHandler**:

```
SERVICE_STATUS_HANDLE ssHandle;
ssHandle =
RegisterServiceCtrlHandler(MYServiceName,
ServiceControl);
```

Функция **RegisterServiceCtrlHandler** в случае успешного завершения возвращает идентификатор состояния сервиса. При ошибке возвращается нулевое значение.

Заметим, что регистрация функции обработки команд должна быть выполнена немедленно в самом начале работы функции точки входа сервиса.

Обработчик службы должен устанавливать состояние службы при каждом вызове, даже если ее состояние не менялось.

Настройка состояния службы

В процессе инициализации функция точки входа сервиса выполняет действия, которые зависят от назначения сервиса. Необходимо, однако, помнить, что без принятия дополнительных мер инициализация должна выполняться не дольше одной секунды.

В противном случае перед началом инициализации функция точки входа сервиса должна сообщить процессу управления сервисами, что данный сервис находится в состоянии ожидания запуска. Это можно сделать с помощью функции **SetServiceStatus**, которая будет описана позже. Перед началом инициализации вы должны сообщить процессу управления сервисами, что сервис находится в состоянии **SERVICE_START_PENDING**.

Для перевода службы в состояние **SERVICE_START_PENDING** воспользуемся функцией **SetServiceStatus**. Функция **SetServiceStatus** может применяться еще в других местах для установки различных значений параметра состояния, информируя SCM о текущем состоянии службы. Описания других возможных состояний службы, характеризуемых значениями параметра состояния, отличными от

SERVICE_STATUS_PENDING, а также флагов, определяющих тип сервиса, приведены в табл. 14 и 15.

Прототип функции имеет следующий вид:

```
BOOL SetServiceStatus(SERVICE_STATUS_HANDLE
hServiceStatus, LPSERVICE_STATUS lpServiceStatus).
```

Ее параметры:

hServiceStatus – дескриптор типа **SERVICE_STATUS_HANDLE,** возвращенный функцией **RegisterCtrlHandler.** Поэтому вызову функции **SetServiceStatus** должен предшествовать вызов функции **RegisterCtrlHandler.**

lpServiceStatus – указатель на структуру **SERVICE_STATUS**, содержащую описание свойств, состояния и возможностей службы.

Ниже приведено определение структуры **SERVICE_STATUS**:

```
typedef struct _SERVICE_STATUS {
 DWORD dwServiceType;
 DWORD dwCurrentState;
 DWORD dwControlsAccepted;
 DWORD dwWin32ExitCode;
 DWORD dwServiceSpecificExitCode;
 DWORD dwCheckPoint;
DWORD dwWaitHint;
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

В поле **dwServiceType** необходимо записать один из перечисленных ниже флагов, определяющих тип сервиса (табл. 14).

Таблица 14

Флаги типов сервиса

Флаг	Описание
SERVICE_WIN32_ OWN_PROCESS	Сервис работает как отдельный процесс
SERVICE_WIN32_ SHARE_PROCESS	Сервис работает вместе с другими сервисами в рамках одного и того же процесса

Окончание табл. 14

Флаг	Описание
SERVICE_KERNEL_DRIVER	Сервис представляет собой драйвер операционной системы Microsoft Windows NT
SERVICE_FILE_SYSTEM_DRIVER	Сервис является драйвером файловой системы
SERVICE_INTERACTIVE_PROCESS	Сервисный процесс может взаимодействовать с программным интерфейсом рабочего стола Desktop

В поле **dwCurrentState** вы должны записать текущее состояние сервиса. Здесь можно использовать одну из перечисленных в табл. 15 констант.

Таблица 15

Флаги состояний сервиса

Константа	Состояние сервиса
SERVICE_STOPPED	Сервис остановлен
SERVICE_START_PENDING	Сервис находится в состоянии запуска, но еще не работает
SERVICE_STOP_PENDING	Сервис находится в состоянии остановки, но еще не остановился
SERVICE_RUNNING	Сервис работает
SERVICE_CONTINUE_PENDING	Сервис начинает запускаться после временной остановки, но еще не работает
SERVICE_PAUSE_PENDING	Сервис начинает переход в состояние временной остановки, но еще не остановился
SERVICE_PAUSED	Сервис находится в состоянии временной остановки

Задавая различные значения в поле **dwControlsAccepted**, вы можете указать, какие команды обрабатывает сервис. В табл. 16 приведен список возможных значений.

Значение в поле **dwWin32ExitCode** определяет код ошибки WIN32, который используется для сообщения о возникновении ошибочной ситуации при запуске и остановки сервиса. Если в этом поле указать значение **ERROR_SERVICE_SPECIFIC_ERROR**, то будет исполь-

зован специфический для данного сервиса код ошибки, указанной в поле **dwServiceSpecificExitCode** структуры **SERVICE_STATUS**. Если ошибки нет, в поле **dwWin32ExitCode** необходимо записать значение **NO_ERROR.**

Таблица 16

Команды обрабатываемые сервисом

Значение	Команды, которые может воспринимать сервис
SERVICE_ACCEPT_STOP	Команда остановки сервиса **SERVICE_CONTROL_STOP**
SERVICE_ACCEPT_PAUSE_CONTINUE	Команды временной остановки **SERVICE_CONTROL_PAUSE** и продолжения работы после временной остановки **SERVICE_CONTROL_CONTINUE**
SERVICE_ACCEPT_SHUTDOWN	Команда остановки при завершении работы операционной системы **SERVICE_CONTROL_SHUTDOWN**

Поле **dwServiceSpecificExitCode** используется в том случае, когда в поле **dwWin32ExitCode** указано значение **ERROR_SERVICE_SPECIFIC_ERROR**.

Теперь о поле **dwCheckPoint**. Это поле должно содержать значение, которое должно периодически увеличиваться при выполнении длительных операций запуска, остановки или продолжения работы после временной остановки. Если выполняются другие операции, в это поле необходимо записать нулевое значение.

Содержимое поля **dwWaitHint** определяет ожидаемое время выполнения (в миллисекундах) длительной операции запуска, остановки или продолжения работы после временной остановки. Если за указанное время не изменится содержимое полей **dwCheckPoint** или **dwCurrentState**, процесс управления сервисами будет считать, что произошла ошибка.

Здесь: **dwWin32ExitCode** – обычный код завершения, используемый логической службой. Служба должна установить этот код равным **NO_ERROR** в процессе выполнения и при нормальном завершении;

dwServiceSpecificExitCode – может использоваться в качестве кода завершения, когда ошибка возникает при запуске или остановке

службы, но это значение игнорируется, если значение параметра **dwWin32ExitCode** не было установлено равным **ERROR_SERVICE_SPECIFIC_ERROR**;

dwCheckPoint – служба должна периодически увеличивать значение этого параметра для индикации выполнения на всех стадиях, включая стадии инициализации и остановки. Этот параметр не действует и должен устанавливаться равным нулю, если служба не находится в состоянии запуска, остановки, паузы и не выполняет никаких длительных операций;

dwWaitHint – ожидаемая длительность интервалов времени (в миллисекундах) между последовательными вызовами функции **SetServiceStatus**, осуществляемыми с увеличенным значением параметра **dwCheckPoint** или измененным значением параметра **dwCurrentState**. Как уже отмечалось ранее, если на протяжении этого промежутка времени вызова функции **SetServiceStatus** не происходит, то SCM предполагает, что это вызвано возникновением ошибки.

После завершения инициализации функция точки входа сервиса должна указать процессу управления сервисами, что процесс запущен и находится в состоянии **SERVICE_RUNNING**.

Функция обработки команд

Как следует из названия, функция обработки команд, зарегистрированная функцией **RegisterServiceCtrlHandler**, обрабатывает команды, передаваемые сервису операционной системой, другими сервисами или приложениями. Эта функция может иметь любое имя и выглядит следующим образом:

```
void WINAPI ServiceControl(DWORD dwControlCode)
{
switch(dwControlCode)
  {
case SERVICE_CONTROL_STOP:
    {
      ss.dwCurrentState = SERVICE_STOP_PENDING;
ReportStatus(ss.dwCurrentState, NOERROR, 0);
      // Выполняем остановку сервиса, вызывая функцию,
      // которая выполняет все необходимые действия
```

```
      // ServiceStop();
ReportStatus(SERVICE_STOPPED, NOERROR, 0);
break;
    }
case SERVICE_CONTROL_INTERROGATE:
    {
ReportStatus(ss.dwCurrentState, NOERROR, 0);
break;
    }
default:
    {
ReportStatus(ss.dwCurrentState, NOERROR, 0);
break;
}
  }
}
```

В приведенном выше фрагменте кода для сообщения процессу управления сервисами текущего состояния сервиса вызывается функция **ReportStatus**, которая должна вызвать функцию **SetServiceStatus**:

```
BOOL SetServiceStatus(
  SERVICE_STATUS_HANDLE sshServiceStatus,
       // идентификатор состояния сервиса
  LPSERVICE_STATUS lpssServiceStatus);
      // адрес структуры, содержащей состояние сервиса
```

Через параметр **sshServiceStatus** функции **SetServiceStatus** необходимо передать идентификатор состояния сервиса, полученный от функции **RegisterServiceCtrlHandler**.

В параметре **lpssServiceStatus** необходимо передать адрес предварительно заполненной структуры типа **SERVICE_STATUS**:

```
typedef struct _SERVICE_STATUS
{
  DWORD dwServiceType;       // тип сервиса
  DWORD dwCurrentState;      // текущее состояние сервиса
  DWORD dwControlsAccepted; // обрабатываемые команды
```

```
  DWORD dwWin32ExitCode;     // код ошибки при запуске
                             // и остановке
DWORD dwServiceSpecificExitCode;
                          // специфический код ошибки
DWORD dwCheckPoint;
// контрольная точка при выполнении длительных операций
DWORD dwWaitHint;          // время ожидания
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

Для определения текущего состояния сервиса можно использовать функцию **QueryServiceStatus**:

```
BOOL QueryServiceStatus(
  SC_HANDLE schService,     // идентификатор сервиса
  LPSERVICE_STATUS lpssServiceStatus);
                        // адрес структуры SERVICE_STATUS
```

Установка и удаление сервиса

Установка и удаление сервиса производятся при помощи утилиты **sc**. Установка производится следующей командой:

```
sc create SampleService binpath= c:\SampleService.exe
```

Удаление сервиса:

```
sc delete SampleService
```

Для успешной установки службы необходимо иметь права администратора. Напоминаем, что по умолчанию служба запускается от имени системной учетной записи (LocalSystem).

Область видимости и права доступа к объектам, созданным службой

Если служба создает объекты ядра с атрибутами безопасности по умолчанию, то такая защита подразумевает, что создатель объекта и любой член группы администраторов получают к нему полный доступ, а все прочие к объекту не допускаются. Соответственно клиент такой службы должен быть запущен от имени пользователя, имеющего права

администратора. Либо служба должна быть запущена от имени конкретного пользователя системы. Но даже в этом случае пользовательский процесс может не получить доступ к объектам ядра, созданным службой.

Дело в том, что на машине с поддержкой удаленного доступа к рабочему столу существует множество пространств имен для объектов ядра [7]. Объекты, которые должны быть доступны всем клиентам, используют одно глобальное пространство имен (такие объекты, как правило, связаны с сервисами, предоставляемыми клиентским программам). В каждом клиентском сеансе формируется свое пространство имен, чтобы исключить конфликты между несколькими сеансами, в которых запускается одно и то же приложение. Ни из какого сеанса нельзя получить доступ к объектам другого сеанса, даже если у их объектов идентичные имена.

Именованные объекты ядра, относящиеся к какому-либо сервису, всегда находятся в глобальном пространстве имен, а аналогичный объект, связанный с приложением, служба удаленного доступа по умолчанию помещает в пространство имен клиентского сеанса. Однако и его можно перевести в глобальное пространство имен, поставив перед именем объекта префикс **"Global\"**, как в примере ниже:

```
HANDLE hEvent = CreateEvenL(NULL, FALSE,
        FALSE, "Global\\MyName");
```

При желании явно указать, что объект ядра должен находиться в пространстве имен клиентского сеанса, следует использовать префикс **"Local\"**:

```
HANDLE hEvent = CreateEvent(NULL, FALSE,
        FALSE, "Local\\MyName");
```

Характерным признаком отсутствия видимости объекта ядра является ошибка 2: «Не удается найти указанный файл», возвращаемая функцией **GetLastError** сразу после попытки открытия объекта ядра из клиентского приложения:

```
hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE,
lpEventName);
  if (hEvent == NULL)
    fprintf(stderr, "OpenEvent: Error %ld\n",
    GetLastError());
```

Microsoft рассматривает префиксы Global и Local как зарезервированные ключевые слова, которые не должны встречаться в самих именах объектов. К числу таких слов Microsoft относит и Session. Также обратите внимание на две вещи: все эти ключевые слова чувствительны к регистру букв и игнорируются, если компьютер работает без удаленного доступа к рабочему столу.

Данные утверждения относятся к операционным системам версий ***до*** Vista и Server 2008. В этих и более современных ОС области видимости объектов ядра, созданных службами и клиентскими приложениями, работающими в сеансе пользователя (не важно, консольными или терминальными) не пересекаются. Клиентские приложения могут создавать и получать доступ к объектам ядра ***только*** в области видимости сеанса (**`Session\1`**), что хорошо видно в утилите WinObj Марка Руссиновича из компании SysInternals (http://download.sysinternals.com/files/WinObj.zip) или WinObjEx (Windows Objects Explorer) версии 3.2 Андрея Ивлева (aka Four-F) (http://www.woodmann.com/collaborative/ tools/images/Bin_WinObjEx_2007-12-29_1.41_ WinObjEx.zip). Последняя утилита и новее, и более функциональна. Таким образом, служба, создающая объекты ядра в этих версиях ОС, должна использовать префикс **`Session\1`**:

```
HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE,
                "Session\\1\\MyName");
```

Напоминаем, что права доступа к объектам ядра, созданным службой, запущенной от имени локальной системы (LocalSystem), имеет только создатель и члены группы администраторов. Если клиентское приложение почему-либо нельзя запустить от имени администратора, попытка открытия объекта ядра вернет ошибку 5: «Отказано в доступе». Следовательно, служба должна создавать объекты ядра не с нулевым атрибутом безопасности (первый аргумент функций **`CreateEvent, CreateSemaphore, CreateMutex`**), а с таким, который разрешит доступ всем пользователям. Создание атрибутов безопасности (**`SECURITY_ATTRIBUTES`**) и дескрипторов безопасности (**`SECURITY_DESCRIPTOR`**), а также последующая настройка списков контроля доступа (ACL, Access-Control List) – отдельная большая тема [8], выходящая за пределы данного курса. Тема большая уже потому, что таких списков два: Discretionary Access-Control List (DACL) – список управления избирательным доступом и System Access-Control List (SACL) – системный список управления доступом. Именно DACL

формирует правила, кому разрешить доступ к объекту, а кому – запретить. SACL позволяет лишь управлять аудитом.

Альтернативным (возможно, паллиативным) решением будет вызов функции (http://rsdn.org/forum/winapi/4007024.all):

```
SetSecurityInfo(hEvent,SE_KERNEL_OBJECT,
DACL_SECURITY_INFORMATION,NULL,NULL,NULL,NULL);
```

сразу после создания объекта ядра. Данная функция, также как и указанные при ее вызове структуры, описана в include-файле **AclAPI.h**, она просто устанавливает в ноль (**NULL**) список дискреционного контроля доступа (DACL), тем самым доступ к объекту, к которому она применена, получат все.

Аналогичные рассуждения относятся и к таким объектам ядра, как именованные каналы и почтовые ящики (mailslots). Отличия лишь в том, что в утилите WinObj они находятся в папке Devices, а в WinObjEx для их просмотра можно также выбрать команду меню Extras (Дополнительно), а не искать в общей куче объектов папки Devices.

Примеры приложений

Пример системной службы, создающей события (для семафоров и мьютексов необходимо заменить соответствующие функции) и отображаемые на память файлы, для запуска в современных ОС, а также обращающегося к ней приложения, доступны на сайте дисциплины в папке **Services** с суффиксами MF (MappedFile).

Пример системной службы, создающей объекты именованных каналов, доступен на сайте дисциплины в папке **Services** с суффиксом NP (NamedPipes). В качестве обращающегося к ней приложения можно использовать обычный клиент именованных каналов из папки **Winpipes**.

Пример системной службы, создающей объекты почтовых ящиков, доступен на сайте дисциплины в папке **Services** с суффиксами MS (MailSlots). В качестве обращающегося к ней приложения можно использовать обычный клиент почтовых ящиков из папки **Mailslots**.

Вопросы для самопроверки

1. Что такое «системные службы (сервисы)» в Windows? Каково их назначение и свойства?
2. Чем служба отличается от консольного приложения?
3. Что должна содержать функция **`main`** сервисного процесса?
4. Что такое точка входа сервиса, что она содержит?
5. Каково назначение и содержание функции обработки команд?
6. Как производятся установка и удаление, запуск и останов службы?
7. Каковы особенности работы с объектами ядра, созданными службой?

Упражнение

Написать алгоритм (с указанием основных функций и их параметров) работы системной службы.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Гордеев А.В*. Системное программное обеспечение / А.В. Гордеев, А.Ю. Молчанов. – СПб.: Питер, 2002. – 736 с.

2. *Харт Дж. М.* Системное программирование в среде Windows. – М.: Вильямс, 2005. – 592 с.

3. *Гунько А.В.* Системное программное обеспечение: конспект лекций. – Новосибирск: Изд-во НГТУ, 2011. – 136 с.

4. *Гунько А.В.* Программирование в среде Windows [Электронный ресурс]. – Режим доступа: http://gun.cs.nstu.ru/winprog.

5. *Фролов А.*, *Фролов Г*. Программирование для Windows NT. Том 27, часть 2. – М.: Диалог-МИФИ, 1996. – 272 с.

6. Дэрин Кили, Winsock. [Электронный ресурс]. – Режим доступа: http://msdn.microsoft.com/ru-ru/library/dd335942.aspx.

7. *Руссинович М.* Внутреннее устройство Windows / М. Руссинович, А. Ионеску, Д. Соломон, П. Йосифович. 7-е изд. – СПб.: Питер, 2018. – 944 с

8. *Рихтер Дж*. Windows. Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. – 4-е изд. – СПб.: Питер, 2008. – 752 с.

ОГЛАВЛЕНИЕ

Гунько Андрей Васильевич

ПРОГРАММИРОВАНИЕ
(В СРЕДЕ WINDOWS)

Учебное пособие

Редактор *М.О. Мокшанова*
Выпускающий редактор *И.П. Бровановa*
Корректор *И.Е. Семенова*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *Л.А. Веселовская*

Налоговая льгота – Общероссийский классификатор продукции
Издание соответствует коду 95 3000 ОК 005-93 (ОКП)

Подписано в печать 23.05.2019. Формат 60 × 84 1/16. Бумага офсетная. Тираж 100 экз.
Уч.-изд. л. 9,06. Печ. л. 9,75. Изд. № 305/18. Заказ № 923. Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630073, г. Новосибирск, пр. К. Маркса, 20