Министерство образования и науки Российской Федерации

САНКТ–ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Приоритетный национальный проект «Образование»
Национальный исследовательский университет

М. И. ПОЛУБЕНЦЕВА

МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ

Санкт-Петербург
Издательство Политехнического университета
2010

УДК
ББК

Рецензенты:
Доктор технических наук, профессор,
Научный руководитель ИГУИ *А.М.Яшин*
Доктор технических наук, профессор,
профессор ГУАП *Г.И.Коршунов*

Полубенцева М. И. **Межпроцессное взаимодействие**: Учеб. пособие. / М. И. Полубенцева – СПб.: Изд-во Политехн. ун-та, 2010. – 258 с.

В учебном пособии излагаются способы распараллеливания и синхронизации программ, выполняющихся как на одном, так и на нескольких компьютерах в сети. Основное внимание уделяется способам реализации многозадачности с точки зрения прикладного программиста. Приводятся способы обмена данными между процессами. Рассматриваются варианты создания и использования библиотек. Приводятся рекомендации по применению системных средств для создания распределенных программ. Примеры приведены на языке C/C++ для операционной системы Windows.

Учебное пособие предназначено для студентов вузов, обучающихся по направлению подготовки магистров «Математическое обеспечение и администрирование информационных систем».

Работа выполнена в рамках реализации программы развития национального исследовательского университета «Модернизация и развитие политехнического университета как университета нового типа, интегрирующего мультидисциплинарные научные исследования и надотраслевые технологии мирового уровня с целью повышения конкурентоспособности национальной экономики»

Печатается по решению редакционно-издательского совета
Санкт-Петербургского государственного политехнического университета.

© Полубенцева М.И.. , 2010
© Санкт- Петербургский государственный
политехнический университет, 2010

ОГЛАВЛЕНИЕЕ

ВВЕДЕНИЕ 5

1. ОБЪЕКТНАЯ МОДЕЛЬ ОПЕРАЦИОННОЙ СИСТЕМЫ WINDOWS 7

1.1. СИСТЕМНАЯ ПОЛИТИКА 7

1.2. ДИСПЕТЧЕР ОБЪЕКТОВ 13

1.3. СИСТЕМНЫЕ ФУНКЦИИ ОБЩЕГО НАЗНАЧЕНИЯ 15

1.4. КАТЕГОРИИ ОБЪЕКТОВ 16

1.5. СТРУКТУРА ОБЪЕКТА ИСПОЛНЯЮЩЕЙ СИСТЕМЫ 18

1.6. УПРАВЛЕНИЕ ОБЪЕКТАМИ 26

2. МНОГОЗАДАЧНОСТЬ 34

2.1. СПОСОБЫ ОРГАНИЗАЦИИ МНОГОЗАДАЧНОСТИ 35

2.2. СРЕДСТВА ПОДДЕРЖКИ МНОГОЗАДАЧНОСТИ 39

ОПЕРАЦИОННОЙ СИСТЕМОЙ 39

2.3. ВЫБОР ТИПА МНОГОЗАДАЧНОСТИ 40

3. ПРОЦЕССЫ 43

3.1. МЕХАНИЗМ ИСПОЛЬЗОВАНИЯ ОБЪЕКТОВ ПРОЦЕССОМ 44

3.2. СИСТЕМНЫЕ СРЕДСТВА ДЛЯ СОЗДАНИЯ ПРОЦЕССА 46

3.3. СИСТЕМНАЯ ФУНКЦИЯ CreateProcess() 47

3.4. ПОЛУЧЕНИЕ ИНФОРМАЦИИ О «СВОЕМ» ПОТОКЕ ИЛИ ПРОЦЕССЕ 59

3.5. СПОСОБЫ ЗАПУСКА ДОЧЕРНИХ ПРОЦЕССОВ С ПОМОЩЬЮ ФУНКЦИИ CreateProcess() 60

3.6. СИСТЕМНАЯ ФУНКЦИЯ ShellExecute() 66

3.7. ЗАВЕРШЕНИЕ ПРОЦЕССА 68

4. ОБМЕН ДАННЫМИ МЕЖДУ ПРОЦЕССАМИ 70

4.1. ОБМЕН ДАННЫМИ ПОСРЕДСТВОМ ФАЙЛА 71

4.2. БУФЕР ОБМЕНА 71

4.3. ПОЛЬЗОВАТЕЛЬСКОЕ СООБЩЕНИЕ 72

4.4. СИСТЕМНОЕ СООБЩЕНИЕ WM_COPYDATA 75

5. МЕХАНИЗМ ОТОБРАЖЕНИЯ ФАЙЛОВ НА ПАМЯТЬ (File mapping) 80

5.1. РЕАЛИЗАЦИЯ ПРОГРАММИСТОМ МЕХАНИЗМА 81

ОТОБРАЖЕНИЯ ФАЙЛОВ НА ПАМЯТЬ 81

6. РАЗДЕЛЕНИЕ ДАННЫХ ИСПОЛНЯЕМЫМИ ФАЙЛАМИ 98

7. СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ ОБЪЕКТОВ 100

7.1. НАСЛЕДОВАНИЕ ОПИСАТЕЛЕЙ ОБЪЕКТОВ 100

7.2. ИМЕНОВАНИЕ ОБЪЕКТОВ 102

7.3. ДУБЛИРОВАНИЕ ОПИСАТЕЛЕЙ 103

7.4. ПЕРЕНАПРАВЛЕНИЕ СТАНДАРТНОГО ВВОДА/ВЫВОДА 105

8. БИБЛИОТЕКИ 109

8.1. ОБЪЕКТНАЯ БИБЛИОТЕКА 110

8.2. ДИНАМИЧЕСКИЕ БИБЛИОТЕКИ 117
9. ПОТОКИ 161
9.1. ПОНЯТИЯ, СВЯЗАННЫЕ С ПОТОКАМИ 162
9.2. СОСТОЯНИЕ ПОТОКА 164
9.3. ПЕРЕКЛЮЧЕНИЕ ПОТОКОВ 166
9.4. ПОТОКОВАЯ ФУНКЦИЯ 167
9.5. СОЗДАНИЕ ПОТОКА 170
9.6. ЗАВЕРШЕНИЕ ПОТОКА 176
9.7. ЗАДЕРЖКА И ВОЗОБНОВЛЕНИЕ ПОТОКОВ 177
10. ПРИОРИТЕТЫ ПРОЦЕССОВ И ПОТОКОВ В WIN32 181
10.1. КЛАССЫ ПРИОРИТЕТА ПРОЦЕССОВ 181
10.2. ОТНОСИТЕЛЬНЫЙ ПРИОРИТЕТ ПОТОКА 186
10.3. ДЕМОНСТРАЦИЯ ПЕРЕКЛЮЧЕНИЯ ПОТОКОВ СИСТЕМОЙ 191
11. СИНХРОНИЗАЦИЯ ПОТОКОВ 193
11.1. СИНХРОНИЗАЦИЯ ПОТОКОВ. 193
ХУДШЕЕ, ЧТО МОЖНО СДЕЛАТЬ 193
11 .2. ПРОСТЕЙШИЕ СПОСОБЫ СИНХРОНИЗАЦИИ 195
11.3. СВОБОДНОЕ СОСТОЯНИЕ ОБЪЕКТА ИСПОЛНЯЮЩЕЙ СИСТЕМЫ 199
11.4. ФУНКЦИИ ОЖИДАНИЯ 201
11.5. СИНХРОНИЗИРУЮЩИЕ ОБЪЕКТЫ 207
12. КАНАЛЫ 231
12.1. МЕХАНИЗМ УДАЛЕННОГО ОБМЕНА ДАННЫМИ 231
12.2. РЕДИРЕКТОР 232
12.3. ОСОБЕННОСТИ КАНАЛОВ 233
12.4. АНОНИМНЫЕ КАНАЛЫ 234
12.5. ИМЕНОВАННЫЕ КАНАЛЫ 238
БИБЛИОГРАФИЧЕСКИЙ СПИСОК 257

ВВЕДЕНИЕ

Современные операционные системы предоставляют прикладному программисту все больше возможностей, в частности для поддержки многозадачности. Поэтому в настоящее время умение программировать включает не только знание языка программирования как такового, но и знакомство с системным уровнем. Прикладному программисту необходимо уметь использовать средства, предоставляемые операционной системой, для того, чтобы разрабатываемое многозадачное приложение эффективно выполнялось в многозадачной операционной среде и корректно взаимодействовало с другими приложениями.

Название «межпроцессное взаимодействие» является переводом общепринятого термина «Inter-Process communication», сокращенно IPC. Межпроцессное взаимодействие – это совокупность способов обмена данными между приложениями и системных средств синхронизации задач. Задачи могут выполняться как в рамках одного приложения, так и в разных приложениях. Приложения, в свою очередь, могут быть запущены на одном или более компьютерах, связанных между собой сетью. Межпроцессное взаимодействие включает такие понятия, как обмен сообщениями, средства синхронизации, использование разделяемой памяти, каналов и других механизмов, предоставляемых операционной системой для обмена данными.

В учебном пособии излагаются базовые понятия системного уровня в объеме, достаточном для корректного и эффективного использования приложениями системных ресурсов, понимания механизмов обмена данными и синхронизации выполнения отдельных задач. Также обсуждаются возникающие при этом проблемы и способы их решения. Приводятся описания системных средств и примеры решения типовых задач. Материал структурирован таким образом, чтобы студент мог максимально быстро освоить

механизмы, предоставляемые операционной системой для межпроцессного взаимодействия. Для облегчения восприятия материала приводятся поясняющие рисунки и сравнительные таблицы. При наличии нескольких вариантов действий, приводятся рекомендации.

Так как каждая операционная система обладает своими специфическими особенностями и накладывает свои ограничения на организацию многозадачности, в пособии объясняется, как прикладной программист может организовать параллельное исполнение нескольких программ (потоков команд) под управлением ОС Windows, обеспечить совместное использование ими ресурсов вычислительной системы и обмен данными. Учебное пособие предоставляет студенту необходимый объем теоретических знаний для самостоятельного выполнения лабораторных работ по соответствующему курсу, а также может быть использовано прикладным программистом в качестве справочного материала при выборе способов реализации многозадачности.

Примеры приводятся на языках Си/C++, так как именно на этих языках написана большая часть операционной системы Windows.

Для успешного прохождения курса студент должен обладать следующими знаниями и навыками:

- программировать на языках Си/C++;
- иметь представление о структуре приложений, управляемых событиями;
- знать, как реализовано защищенное виртуальное адресное пространство;
- использование библиотеки классов Qt необязательно, но упрощает создание интерфейсной части в демонстрационных примерах.

1. ОБЪЕКТНАЯ МОДЕЛЬ ОПЕРАЦИОННОЙ СИСТЕМЫ WINDOWS

1.1. СИСТЕМНАЯ ПОЛИТИКА

Под системной политикой имеется в виду распределение ресурсов вычислительной системы и контроль над их использованием. При выполнении каждое приложение и сама операционная система (ОС) используют системные ресурсы. Ресурсы могут быть разной природы: аппаратные и программные. Основные виды системных ресурсов представлены на рис. 1.1.

Рис. 1.1. Виды системных ресурсов

Проблема использования системных ресурсов заключается в том, что:

- видов ресурсов очень много;
- разные приложения могут конкурировать за одни и те же ресурсы;
- кто-то должен выделять ресурсы приложению;
- кто-то должен контролировать использование ресурсов приложением.

В табл. 1.1 приведен перечень требований, предъявляемых к ОС для реализации системной политики, а также способы решения.

Описание ресурсов вычислительной системы посредством объектов и независимость базовых сервисов от типа объекта

позволяют сосредоточить основную часть работу с ресурсами в одном компоненте ОС, что, в свою очередь ведет к повышению безопасности функционирования системы в целом. В частности такой подход позволил сертифицировать ОС Windows 2000 на уровне C2 «Оранжевой книги».

Таблица 1.1

Реализация системной политики

Требования	Способы решения
обеспечить унифицированное описание системных ресурсов	любой системный ресурс (независимо от типа) можно описать как объект, то есть посредством унифицированной структуры данных
обеспечить унифицированное использование любого системного ресурса: • получение доступа, • общие действия, • освобождение	для каждого ресурса можно выделить общие, не зависящие от типа ресурса, действия и осуществлять такие действия одинаково посредством базовых системных сервисов и уникального описателя ресурса (дескриптора объекта)
позволить совместно использовать нескольким программам одни и те же системные ресурсы	требуется обеспечить синхронизацию доступа к ресурсам и контроль над их использованием, а также ввести общие правила для получения доступа и продолжительности жизни объектов (object retention)
защищать ресурсы от несанкционированных действий программиста	не разрешать напрямую обращаться к ресурсам, а только посредством системных сервисов

Окончание табл. 1.1

Требования	Способы решения
предоставлять разным группам пользователей разные права по использованию одних и тех же ресурсов	проверять соответствие прав и запросов по использованию ресурсов, исходя из удостоверения (прав) пользователя
ограничивать процессы по количеству используемых ими ресурсов	определять и «списывать» с процесса квоты за использование ресурсов
учитывать разницу различных операционных сред (таких как Windows, Posix…) по использованию одних и тех же ресурсов	поддерживать способность наследовать ресурсы родительского процесса дочерним процессом, реализованные в операционных системах Windows и POSIX
давать имена объектам, которые однозначно идентифицируют объект в ОС	обеспечить механизм, который позволяет по имени получить доступ к самому объекту

Замечание: можно провести аналогию с проектированием иерархии классов при объектно-ориентированном решении задачи, когда общие данные и вся не зависящая от типа функциональность выносятся в базовый класс, а специализация добавляется в производный класс.

Организация системной политики в ОС Windows

В ОС Windows (начиная с NT4) работа с ресурсами полностью монополизирована операционной системой. Базовую работу с объектами, описывающими любой ресурс, осуществляет компонент ОС, который называется диспетчером объектов (object manager), специфические действия выполняют соответствующие подсистемы. На рис. 1.2 представлены уровни абстракции, отделяющие

приложение от аппаратных ресурсов вычислительной системы.

Рассмотрим те уровни, которые связаны с реализацией системной политики (выделены на рисунке серым цветом). Начиная с Windows 2000, на самом деле существуют два вида объектов: объекты исполняющей системы (executive objects) и объекты ядра (kernel objects). Объекты ядра даже опосредованно приложениям «не видны», а используются только следующим уровнем абстракции – исполняющей системой.

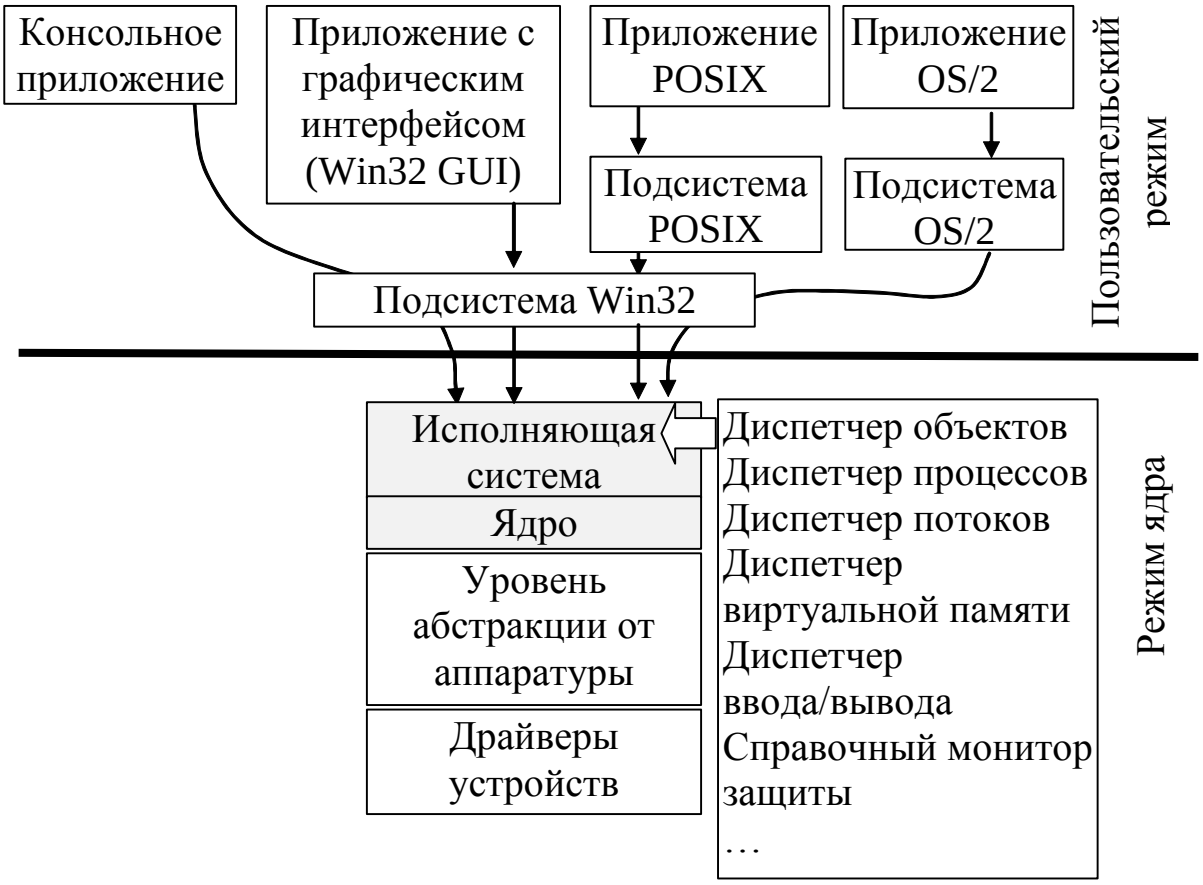

Рис. 1.2. Уровни абстракции, предоставляемые ОС

Уровень ядра

На этом уровне ОС не занимается системной политикой, кроме одного важного исключения – распределения времени между задачами, то есть планирования и диспетчирования потоков. На этом уровне создаются примитивные объекты, которые предназначены

только для внутреннего использования, поэтому приложение не получает к ним даже косвенного доступа посредством дескриптора. Объекты ядра обеспечивают фундаментальные, низкоуровневые, четко определенные и предсказуемые механизмы, которые в свою очередь используются компонентами ОС более высокого уровня.

Объекты ядра делятся на:

- управляющие – примитивные объекты процессов, прерываний, отложенного вызова процедур (DPC), асинхронного вызова процедур (APC) и т. д.;
- примитивные объекты диспетчеризации (события, таймеры, мьютексы, семафоры…) частично обрабатываются ядром, так как имеют непосредственное отношение к планированию потоков.

Замечания:

- большая часть объектов ядра все время находится в оперативной памяти (не выгружается на диск);
- большая часть кода для работы с этими объектами написана на языке Си, и только аппаратно-зависимые и критичные ко времени части на ассемблере;
- компонентам ядра (включая драйверы) все объекты ядра доступны просто по указателю.

Основное назначение уровня ядра:

- сделать вышележащий уровень ОС независимым от аппаратуры, то есть уровень ядра предоставляет более высоким уровням абстрактную модель аппаратуры (в частности, на уровне ядра происходит обработка прерываний);
- предоставить полный механизм для переключения контекста потоков (на этом уровне функционирует планировщик потоков);
- скрыть объекты ядра – позволить их использовать только для внутренних нужд исполняющей системы.

Уровень исполняющей системы

Уровень исполняющей системы отвечает за использование ресурсов (файлов, памяти, объектов синхронизации и т. д.), кроме

диспетчирования потоков. Этот уровень спроектирован как связующее звено между пользовательским режимом и фундаментальными механизмами ядра. На этом уровне создаются объекты, доступные приложению посредством дескрипторов.

Замечания:

- код исполняющей системы написан на языке Си;
- так как исполняющая система изолирована от аппаратуры уровнем ядра, она практически не зависит от архитектуры, а значит, может быть перенесена на новые платформы с относительно небольшими усилиями;
- состоит приблизительно из 10 компонентов, большинство из которых представляет просто набор процедур, выполняющихся в режиме ядра, исключениями являются диспетчер питания и диспетчер plug-and-play, которые выполняются в отдельных потоках;
- между отдельными компонентами нет четких границ (поэтому в разных источниках это деление может быть разным), более того, компоненты могут взаимодействовать, вызывая процедуры друг друга.

Связь объектов исполняющей системы и ядра

Объектам исполняющей системы в каждом приложении соответствует уникальное значение – дескриптор (handle), посредством которого приложение может косвенно получить доступ к ресурсу. Объекты ядра недоступны в пользовательском режиме, дескриптор для таких объектов не формируется. Объекты исполняющей системы просто включают один или более примитивных объектов ядра. То есть исполняющая система «строит» свои объекты на базе внутренних примитивных объектов ядра. Ядро можно рассматривать как набор строительных компонентов, который может использовать вышележащий уровень. Клиенты ядра могут комбинировать эти компоненты для построения более сложных структур, которые уже будут определять системную политику. На рис. 1.3 приведен механизм использования приложением ресурсов

посредством объектов.

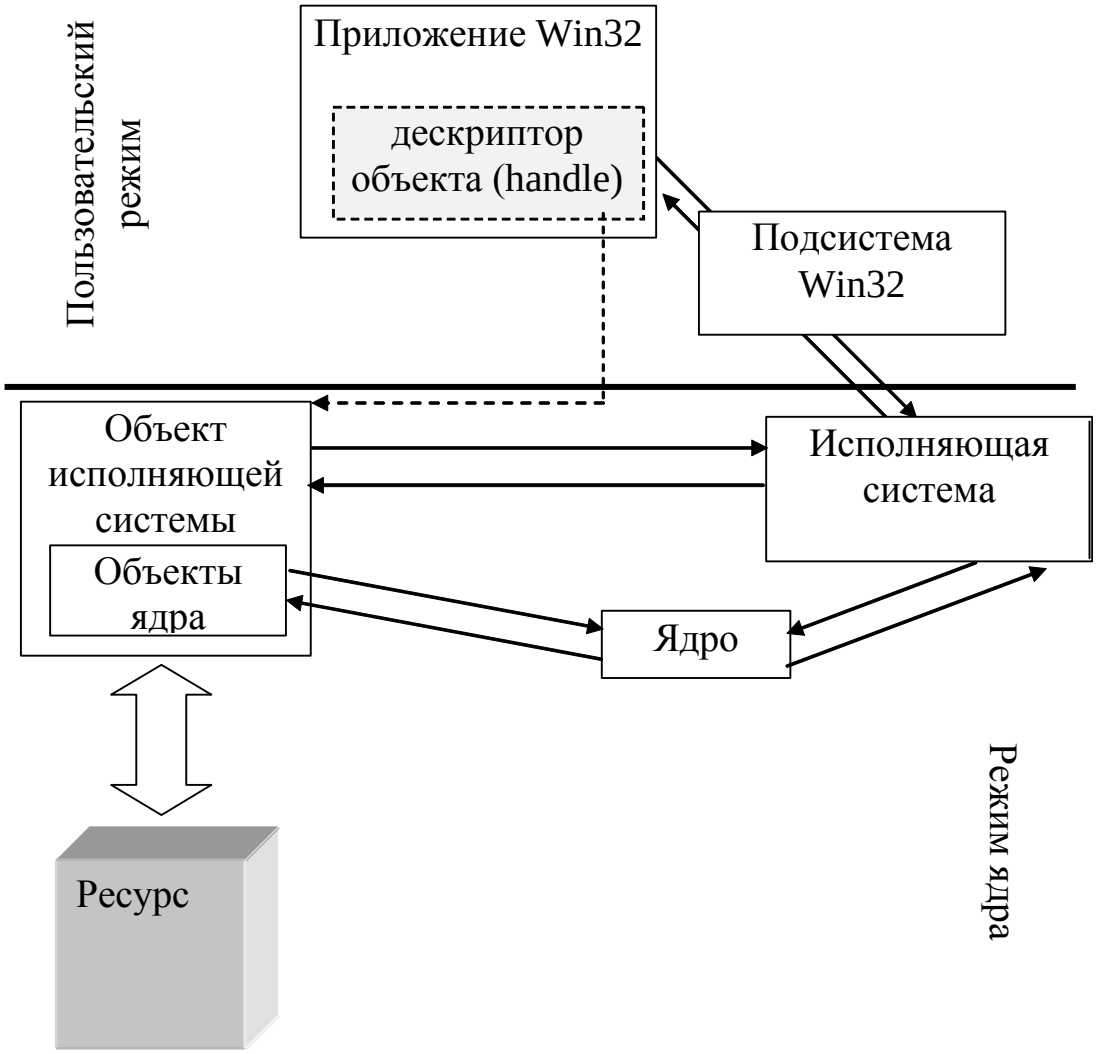

Рис. 1.3. Механизм использования приложением ресурсов посредством объектов

1.2. ДИСПЕТЧЕР ОБЪЕКТОВ

Диспетчер объектов (object manager) – это компонент исполняющей системы, который централизует контроль над ресурсами. Все общие, не зависящие от типа объекта, сервисы сосредоточены в диспетчере объектов, что повышает надежность работы системы в целом. Специфические, зависящие от типа объекта, действия выполняются другими соответствующими компонентами

ОС, такими как диспетчер процессов, диспетчер памяти и т. д., но эти компоненты при выполнении действий в свою очередь все равно обращаются к диспетчеру объектов посредством внутренних интерфейсов.

Диспетчер объектов отвечает за получение доступа к ресурсу, использование и освобождение ресурса. Для выполнения любого из перечисленных действий приложение вызывает соответствующую системную функцию.

При вызове приложением системной функции для создания объекта, подсистема Win32 прямо или косвенно обращается к диспетчеру объектов, который:

- выделяет память в виртуальном адресном пространстве ядра;
- присоединяет к объекту дескриптор защиты (security descriptor), который определяет: кому и как можно использовать данный объект;
- заносит имя объекта в каталог объектов, обеспечивая возможность обращения по имени;
- формирует дескриптор и возвращает его приложению;
- при создании нового объекта или при попытке обращения к существующему объекту диспетчер объектов «списывает» с процесса квоту за использование ресурса.

При манипулировании ресурсом диспетчер объектов:

- производит поиск объекта по указанному дескриптору или по имени;
- следит за корректным использованием, то есть при любом обращении к ресурсу проверяет, есть ли у приложения право совершать требуемые действия (например, если файл открыт только для чтения, то при попытке записи будет отказано в доступе).

При освобождении ресурса диспетчер объектов:

- проверяет, не используется ли объект другими приложениями;
- удаляет неиспользуемые объекты, то есть участок виртуального адресного пространства ядра, зарезервированный под

объект, освобождается – таким образом, память «возвращается» системе;

- возвращает приложению квоту, списанную с приложения за использование ресурса.

За деятельностью диспетчера объектов можно проследить с помощью разных утилит, например, с помощью диспетчера задач (task manager), утилиты winobj.exe, утилиты oh.exe и т. д.

1.3. СИСТЕМНЫЕ ФУНКЦИИ ОБЩЕГО НАЗНАЧЕНИЯ

Для создания объекта подсистема Win32 обращается к соответствующему компоненту ОС. Каким бы компонентом исполняющей системы не создавался объект, для выделения памяти, для контроля над использованием, освобождением, соответствующий компонент обращается к диспетчеру объектов. Но существует набор действий общего назначения, которые не зависят от типа объекта и выполняются непосредственно диспетчером объектов, минуя другие компоненты. Перед выполнением любого действия диспетчер объектов проверяет наличие прав, и, если права не соответствуют запросу, операция игнорируется. Перечень таких универсальных действий приведен в табл. 1.2.

Таблица 1.2

Универсальные системные функции

Системная функция	Назначение
CloseHandle	Закрывает дескриптор объекта. Операция вовсе необязательно приводит к уничтожению объекта
DuplicateHandle	Позволяет разделять объект разным процессам путем дублирования описателя и передачи его другому процессу

Окончание табл. 1.2

Системная функция	Назначение
GetHandleInformation	Получает информацию о свойствах дескриптора (возможности наследования данного объекта дочерним процессом, можно ли закрыть описатель объекта с помощью CloseHandle() и т. д.)
SetHandleInformation	Формирует новые свойства дескриптора
GetSecurityInfo	Получает копию дескриптора безопасности объекта
SetSecurityInfo	Изменяет параметры защиты объекта
WaitForSingleObject	Синхронизирует выполнение потока с одним объектом
WaitForMultipleObjects	Синхронизирует выполнение потока с несколькими объектами
MsgWaitForMultipleObjects	Синхронизирует выполнение потока с несколькими объектами и позволяет обрабатывать события

Замечание: перечисленные функции работают со всеми объектами, хотя для некоторых смысла не имеют и реализованы как заглушки.

1.4. КАТЕГОРИИ ОБЪЕКТОВ

Далеко не все системные структуры описывают объекты. Только данные, которые требуется разделять, защищать, именовать или делать “видимыми” (посредством системных функций) приложениям пользовательского режима, заключаются в объекты. Структуры, которые используются только одним компонентом ОС для реализации внутренних функций, не считаются объектами исполняющей системы. Это объекты подсистем User и GDI, которые иногда называют «старыми объектами». Основным отличием является то, что функции для работы с объектами User и GDI

выполняются в пользовательском режиме, в то время как работа с объектами исполняющей системы происходит в режиме ядра. На рис. 1.4 для сравнения приведены виды системных объектов.

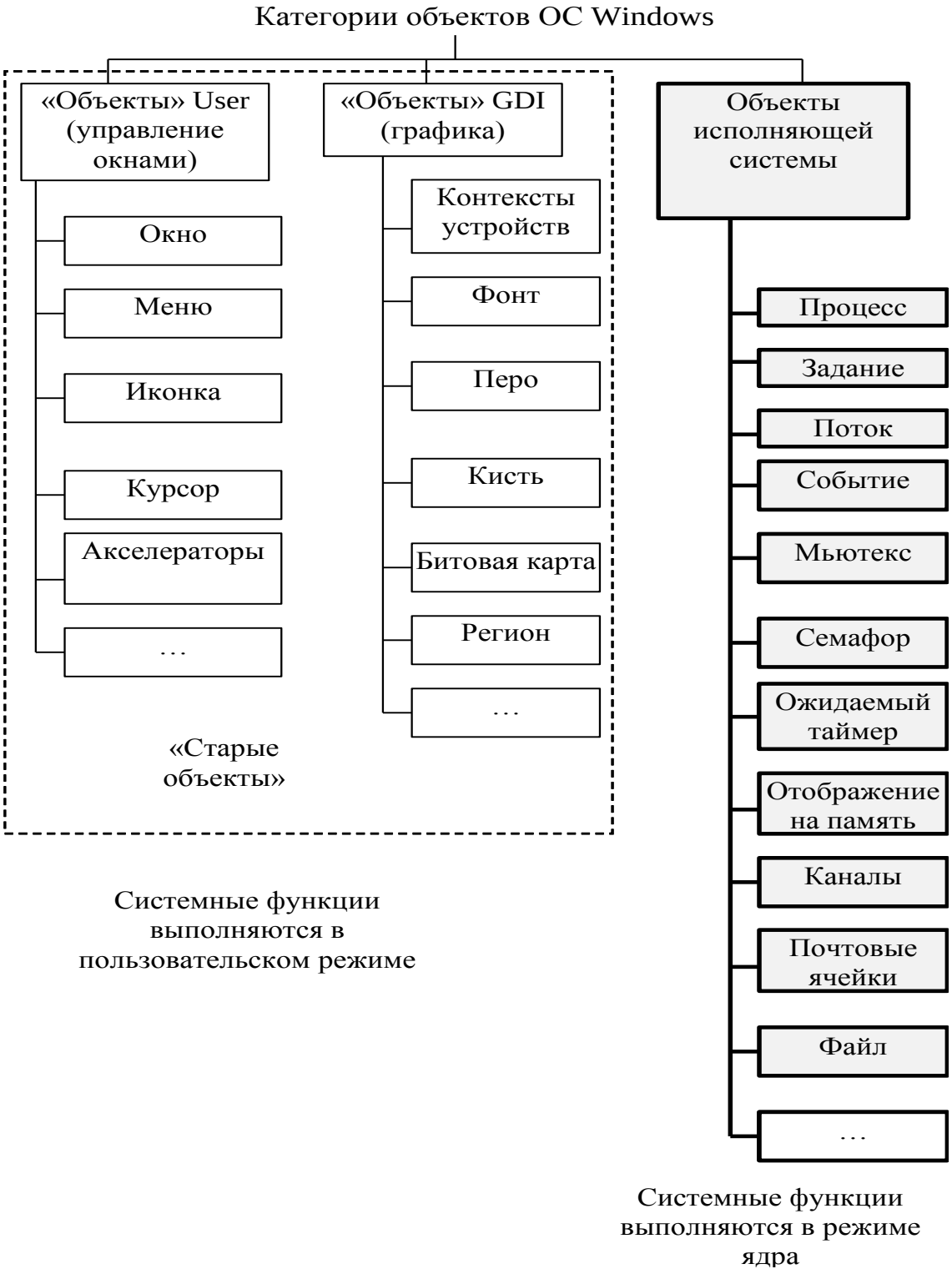

Рис. 1.4. Виды объектов ОС Windows

1.5. СТРУКТУРА ОБЪЕКТА ИСПОЛНЯЮЩЕЙ СИСТЕМЫ

Для описания каждого используемого системного ресурса создается соответствующий объект. Объект – это блок памяти, содержащий информацию о ресурсе. Специфика такого блока памяти заключается в том, что

- память выделяется в виртуальном адресном пространстве ОС, а не приложения, запросившего доступ к данному ресурсу;
- данные объекта напрямую приложению не доступны, а обращаться к ним может только исполняющая система**.**

Такое жесткое ограничение обеспечивает:

- корректное использование объекта (в частности при совместном использовании несколькими приложениями);
- дает возможность разработчикам ОС изменять содержимое системной структуры прозрачно для любого использующего этот объект приложения, при этом приложение перестраивать не нужно.

Структура данных, описывающая любой объект исполняющей системы, унифицирована и разделена на две части: заголовок и тело объекта – рис. 1.5.

Заголовок объекта

В заголовок объекта помещаются данные, смысл и формат которых одинаков для объектов любого типа, а значения могут быть различными. С заголовком объекта работает диспетчер объектов, при этом диспетчеру объектов совершенно не требуется знать:

- какой именно ресурс описывает объект;
- какие еще специфические данные присутствуют в структуре.

Рассмотрим поля заголовка объекта.

1. Имя объекта задается строкой, например, «myObject» и делает возможным использование объекта разными процессами. При задании имени регистр значения не имеет. Диспетчер объектов умеет по имени находить сам объект, возвращая приложению уникальный

идентификатор объекта – дескриптор. Имена объектов – это один из способов разделять ресурс несколькими приложениями. Имена объектов являются «глобальными» для данного компьютера, но «не видны» по сети. Однако для доступа к именованным объектам на другом компьютере диспетчер объектов предоставляет метод разбора (например, диспетчер ввода/вывода расширяет возможности диспетчера объектов для доступа к удаленным файлам).

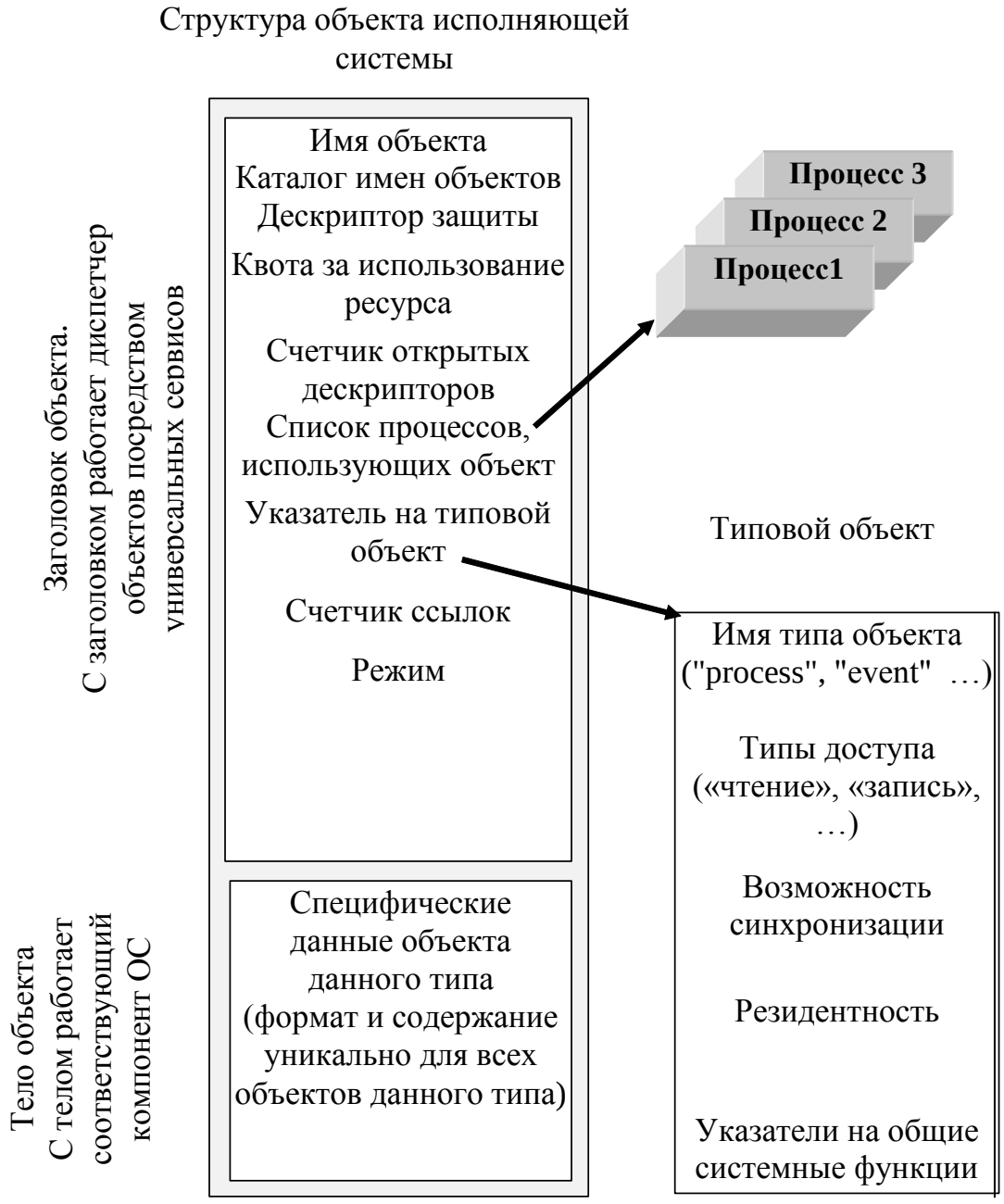

Рис. 1.5. Структура объекта исполняющей системы

Замечание: если нет необходимости использовать объект совместно несколькими процессами, то и именовать его необязательно.

2. Каталог имен объектов – это иерархическая структура для хранения имен объектов. Существует только для нужд диспетчера объектов. Реализован как именованный объект исполняющей системы. Не следует путать каталог имен объектов с файловыми системными директориями (они описываются объектами типа «файл»). Например: \Device object directory содержит именованные device-объекты, создаваемые драйверами.

Главное назначение каталога объектов – при поиске преобразует имя объекта в указатель на него.

3. Дескриптор безопасности обеспечивает контролируемое использование ресурса, то есть определяет, кто может использовать объект и что он может с ним делать. При создании объекты исполняющей системы можно защитить дескриптором безопасности. Такой дескриптор описывает права создателя объекта и каким образом создатель объекта позволяет пользоваться этим объектом остальным. Дескриптор безопасности может быть сформирован:

- явно – программистом (в соответствии с правами пользователя);
- неявно – операционной системой, исходя из прав создателя.

Практически все системные функции, создающие объекты, в качестве одного из параметров принимают указатель на системную структуру SECURITY_ATTRIBUTES. Формат структуры

```
typedef struct _SECURITY_ATTRIBUTES {
   DWORD nLength; //размер структуры в байтах
   LPVOID lpSecurityDescriptor; // указатель на дескриптор
                  безопасности
   BOOL bInheritHandle; //возможность наследования данного
                  объекта дочерним процессом
} SECURITY_ATTRIBUTES;
```

Например, использование дескриптора безопасности функцией создания объекта «файл»:

```
HANDLE CreateFile(
   …,
   PSECURITY_ATTRIBUTES psa, //указатель на структуру
SECURITY_ATTRIBUTES
  …);
```

В большинстве случаев в качестве параметра psa передается просто «0», что предписывает ОС создать дескриптор безопасности по умолчанию. В этом случае ОС использует информацию маркера доступа создающего этот объект процесса. Умолчание предполагает, что администратор и создатель объекта имеют на этот объект полные права, а всем остальным позволяется только такой доступ к объекту, который предусмотрел создатель объекта. Если же требуется «защитить» объект каким-то другим способом (то есть ввести дополнительные ограничения), то требуется создать и проинициализировать структуру SECURITY_ATTRIBUTES, а указатель на нее передать в качестве параметра в соответствующую системную функцию создания объекта.

Создание дескриптора безопасности – вещь нетривиальная. Последовательность действий программиста выглядит примерно так:

- создать переменную типа SECURITY_DESCRIPTOR (эта системная структура содержит информацию об идентификаторе безопасности владельца – SID, идентификаторе группы, список разграничительного контроля доступа – DACL, списке аудита доступа - SACL),
- напрямую инициализировать поля этой структуры не рекомендуется, для этих целей следует использовать системную функцию InitializeSecurityDescriptor().

Как используется в дальнейшем защита объекта? Когда любой процесс пытается обратиться к объекту, сначала вызывается диспетчер безопасности и проверяет соответствие прав процесса и

ограничений на использование объекта. С одной стороны, диспетчер безопасности просматривает маркер доступа процесса, с другой стороны, маркер разграничительного контроля доступа, ассоциированный с объектом в его дескрипторе безопасности. Если процесс обладает достаточными правами, процесс получает доступ к ресурсу, то есть формируется ненулевой дескриптор объекта. Если прав недостаточно или доступ невозможен, то соответствующая системная функция в большинстве случаев возвращает 0 (в некоторых случаях - 1), а с помощью системной функции GetLastError() можно узнать причину отказа. Если в доступе отказано из-за недостатка прав, GetLastError() возвращает ERROR_ACCESS_DENIED. Например, если файл был открыт таким образом, что создателю объекта были разрешены и запись, и чтение, а всем остальным разрешено только читать, то при попытке другим процессом получить доступ к файлу для записи, в доступе будет отказано.

Замечание: один и тот же объект для разных пользователей может быть доступен с разными правам, правда только в том случае, когда запросы не противоречат друг другу.

4. Квота пользователя – это «стоимость» использования ресурса и соответственно возможность контроля использования процессом ресурсов. Когда процесс создает новый объект или получает доступ к существующему объекту, эта квота списывается диспетчером объектов с процесса. Чтобы ограничить процесс в использовании ресурсов, каждому процессу выделяется определенный объем квот. Если процесс пытается получить доступ к очередному ресурсу, но объем квот уже исчерпан, в доступе будет отказано, даже в случае соответствия прав.

Замечание: квоты по использованию ресурсов разного типа у процесса могут быть разными. По каждому виду квот накладываются свои ограничения. Например, диспетчер объектов вычитает из отведенной процессу квоты системной памяти объем памяти, соответствующий данному объекту. Кроме того, различные

компоненты ОС налагают свои специализированные ограничения: диспетчер ввода/вывода на количество открытых файлов, диспетчер процессов на использование времени, а также количество запускаемых данным пользователем процессов и максимальное количество потоков в процессе.

Посредством учета используемых процессом ресурсов у ОС появляется возможность ограничить «жадный» процесс, чтобы он не заблокировал работу всей системы в целом.

5. Счетчик открытых описателей содержит количество пользователей ресурса. Специфика работы с объектами исполняющей системы заключается в том, что:

- все манипуляции ресурсами осуществляются только посредством ОС (диспетчер объектов ведет строгий учет и контроль объектов);
- все объекты принадлежат ОС, а не процессу, использующему данный ресурс;
- для того чтобы можно было использовать объект, нужно открыть его описатель (дескриптор). Описатели формирует только диспетчер объектов, а, следовательно, он всегда знает, какие процессы используют данный объект.

Посредством счетчика открытых описателей диспетчер объектов защищает объект от уничтожения, если ресурс используется другими процессами. Механизм, регулирующий существование объекта, называется «удержанием объекта» (object retention). Сущность механизма:

- в момент создания объекта (при вызове системных функций CreateFile(), CreateMutex() и т. д.) счетчику открытых описателей присваивается значение 1;
- если другой процесс (или даже сам создатель объекта) пытается получить доступ к уже существующему ресурсу (при вызове тех же системных функций CreateFile(), CreateMutex() или OpenFile(), OpenMutex() и т. д.), то новый объект не создается, а просто

формируется еще один (новый) дескриптор для того же самого объекта, а значение счетчика открытых описателей увеличивается на единицу;

- после того, как процесс попользовался ресурсом, полагается ресурс освободить системной функцией CloseHandle(), так как содержание объектов обходится ОС достаточно дорого (память выделяется в виртуальном адресном пространстве самой ОС). При вызове этой системной функции диспетчер объектов уменьшает значение счетчика открытых описателей на единицу и только в том случае, когда счетчик принимает нулевое значение (это означает, что пользователей данного ресурса больше нет), сам объект может быть уничтожен, а занимаемая им память может быть освобождена;
- если процесс не вызывает функцию CloseHandle() (программист неаккуратен, или процесс «не подозревает» о принадлежащих ему ресурсах), то это делает код завершения стандартной библиотеки для обеспечения надежности функционирования системы в целом.

Этот простой механизм гарантирует, что ни один процесс не может уничтожить объект, пока кто-то другой пользуется ресурсом.

6. Счетчик ссылок используется диспетчером объектов для внутренних нужд ОС, так как ОС может тоже пользоваться объектом. В этом случае системным функциям выдается не дескриптор, а указатель на объект. При каждом таком вызове счетчик ссылок инкрементируется, а когда системная функция завершает работу с объектом, она обращается к диспетчеру объектов для уменьшения счетчика ссылок. И только когда значения обоих счетчиков обнуляются, диспетчер объектов действительно удаляет объект и освобождает память.

7. Список процессов позволяет диспетчеру объектов при необходимости перебрать все процессы, использующие данный ресурс.

8. Режим использования – это признак того, как в данный

момент используется ресурс: в пользовательском режиме или режиме ядра.

9. Если в заголовке объекта хранятся данные, формат которых одинаков для объекта любого типа, а содержимое разное, например, уникальное имя объекта, уникальный дескриптор защиты и т. д., то указатель на типовой объект содержит адрес структуры данных, содержащей информацию, одинаковую для всех объектов данного типа:

- имя типа (“event”, “process”, “mutex” и т. д.);
- тип доступа (для файлов – читать, писать, удалять, и т. д., для потоков – завершить, приостановить, возобновить и т.д.);
- признак – можно ли выгружать на диск или объект должен все время находиться в оперативной памяти;
- возможность синхронизации (есть ли свободное/занятое состояние, и могут ли другие потоки синхронизировать свое выполнение с данным объектом, то есть ожидать перехода данного объекта в свободное состояние);
- и т. д.

Так как эта информация одинакова для всех объектов одного и того же типа, то совершенно незачем для каждого объекта создавать свою копию этих общих данных. Поэтому ОС в целях экономии памяти формирует эти данные один раз, а для хранения таких данных использует специальный типовой объект (структура данных).

Также для каждого типа объекта характерны общие специфические действия, которые вызывает диспетчер объектов в некоторых точках жизни объекта:

- при получении доступа к ресурсу – “открыть дескриптор”;
- при отказе от ресурса – «закрыть дескриптор»;
- при удалении объекта – «удалить объект»;
- «запросить имя»;
- при поиске диспетчером по имени «разбор»;
- при считывании или изменении параметров защиты –

«защита»;

- и т. д.

Указатели на соответствующие этим действиям системные функции тоже хранятся в типовом объекте.

Кроме того, типовой объект связывает друг с другом объекты одного и того же типа, чтобы при необходимости диспетчер объектов мог бы перебрать их все.

Тело объекта

Так как все ресурсы разные, кроме общих, не зависящих от типа объекта данных, для каждого объекта формируются дополнительные данные и существуют дополнительные функции, которые учитывают специфику конкретного ресурса. Например, у объекта «процесс» есть такие понятия как идентификатор, приоритет, код завершения, а у объекта «файл» смещение относительно начала, режимы (как он был открыт и может ли им пользоваться другое приложение). Для хранения специфических данных и указателей на специфические функции используется та часть объекта, которая называется телом объекта. Если с заголовком объекта работает диспетчер объектов, то с телом объекта работают соответствующие (другие специализированные для этих целей) компоненты исполняющей системы.

1.6. УПРАВЛЕНИЕ ОБЪЕКТАМИ

Под управлением понимается создание, уничтожение и использование объектов исполняющей системы. В большинстве случаев для манипулирования объектом процесс использует имя объекта или дескриптор объекта.

Дескриптор

Хотя имена объектов важны, особенно при использовании одного объекта разными процессами, но они:

- не обязательны;
- даже располагая именем объекта, прежде чем обратиться к

объекту (вызвать системную функцию), по имени следует получить дескриптор объекта.

Память под объекты исполняющей системы выделяется в виртуальном адресном пространстве самой ОС. Пользовательским процессам напрямую доступ к объектам запрещен. Объекты создаются и уничтожаются всегда только диспетчером объектов и доступны только посредством системных функций. Механизм доступа к объектам реализуется посредством дескрипторов (описателей).

При запуске процесса ОС формирует в виртуальном адресном пространстве процесса специальную таблицу, которая содержит информацию обо всех используемых процессом ресурсах и одновременно обеспечивает опосредованный доступ к ним. Таблица располагается в той области адресного пространства процесса, адреса которой самому процессу не доступны, зато к этим адресам имеют доступ системные функции. В терминах языка программирования таблица представляет собой массив структур, поля которых содержат информацию, необходимую и достаточную для того, чтобы можно было определить, где находится сам объект, а кроме того свойства данного дескриптора (информацию о том, какие действия можно осуществлять над объектом посредством данного дескриптора).

Формат таблицы объектов не документирован. Каждая структура имеет примерно следующие (далеко не все перечисленные ниже) поля. ОС заносит в таблицу описателей процесса следующие данные:

- адрес объекта в виртуальном адресном пространстве ОС;
- права доступа (как данный процесс может пользоваться ресурсом);
- ограничения, например (может ли дочерний процесс наследовать данный объект);
- и т. д.

Дескриптор, который «выдается» системой процессу для

доступа к ресурсу, имеет смысл индекса в описанной выше таблице. Всегда важно помнить о том, что в целях безопасности значение дескриптора имеет смысл только в данном процессе, а в другом процессе тому же самому индексу может соответствовать совершенно другой объект.

На рис. 1.6 представлена реализация дескриптора. Так как процесс-пользователь и сам объект находятся в разных виртуальных адресных пространствах, то доступ к объекту напрямую невозможен.

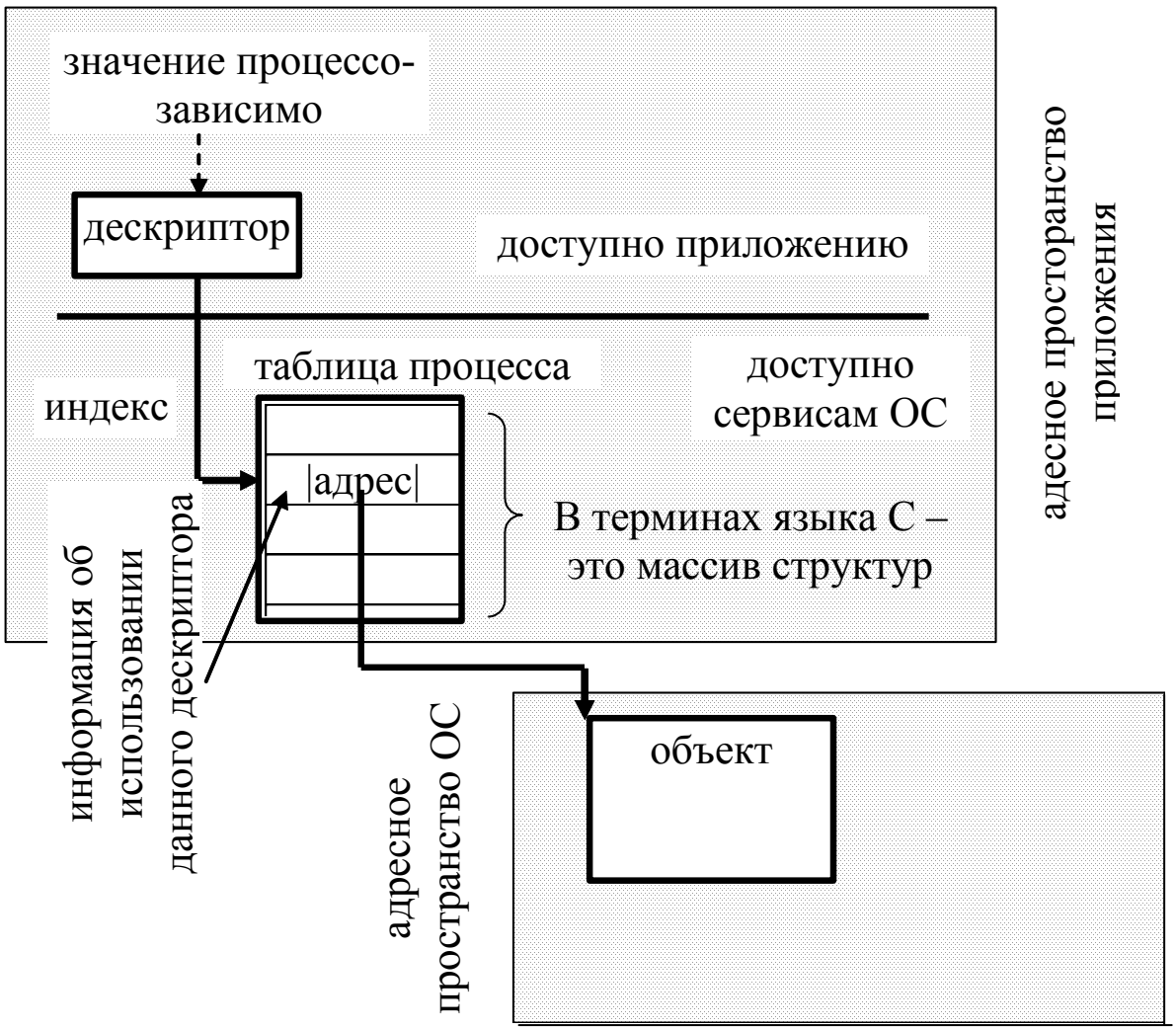

Рис. 1.6. Реализация дескриптора

Такая косвенность при обращении к объекту, с одной стороны, и ограничение использования значения дескриптора только одним

процессом, с другой стороны, обеспечивают:

- защиту данных от несанкционированных действий программиста (использование разрешено только посредством системных функций), что в свою очередь обеспечивает устойчивость к сбоям системы в целом, так как гарантирует, что с объектом будут произведены только корректные действия;
- защиту данных от несанкционированного доступа другого процесса. Если бы описатели были общесистемными, то один процесс мог бы запросто получить описатель объекта, используемого другим процессом, и устроить в этом другом процессе настоящий хаос. Процесс-создатель объекта может предотвратить несанкционированный доступ к нему другого процесса;
- возможность получения разных прав доступа к одному и тому же объекту как в разных процессах, так и в одном и том же процессе посредством формирования разных записей в таблице дескрипторов процесса и соответственно получения разных дескрипторов для одного и того же объекта. Таким образом, посредством разных дескрипторов одним и тем же объектом можно пользоваться по-разному;
- возможность по мере необходимости выгружать объект из памяти на диск и вновь загружать по совершенно другому физическому адресу, то есть адрес может изменяться, а дескриптор остается одним и тем же.

Замечание: понятие дескриптора специфично для Win32, например, в ОС UNIX такого понятия нет. Именно использование дескриптора дает возможность диспетчеру объектов одинаковым образом осуществлять базовые действия с любым объектом независимо от его типа.

Замечание: значение дескриптора имеет смысл только для данного приложения. Это означает: если два приложения пользуются одним и тем же объектом, значения дескрипторов, ассоциированных с этим объектом, в разных приложениях могут быть разными! Поэтому

передавать значение дескриптора другому приложению можно, только если Вы уверены, что одно и то же значение относится к одному и тому же объекту, а это бывает только в исключительных случаях.

Дескриптор объекта выдается приложению в результате:

- вызова одной из системных функций Create – CreateMutex(), CreateThread() и т. д.;
- вызова одной из системных функций Open – OpenMutex(), OpenThread() и т. д.;
- наследования объекта от родительского процесса;
- создания дубликата дескриптора для другого процесса.

Действия программиста для получения доступа к ресурсу

Последовательность действий программиста для получения доступа к ресурсу выглядит примерно так:

1. Создание объекта.

При запуске приложения ОС сама автоматически создает минимум объектов исполняющей системы, например, такие как «процесс», первичный «поток», «файл» и еще несколько объектов, которые будут обсуждаться позже. Далее при необходимости приложение может обращаться к ОС для запроса ресурса посредством соответствующей Win32-функций типа Create*:

```
HANDLE CreateThread(…);
HANDLE CreateFile(…);
HANDLE CreateFileMapping(…);
```

При этом системная функция обращается к соответствующему специализированному компоненту ОС, а тот в свою очередь для создания объекта к диспетчеру объектов. Диспетчер объектов:

- если объект, соответствующий запрашиваемому ресурсу, еще не существует, выделяет для этого объекта блок памяти в адресном пространстве ОС и инициализирует поля этой структуры данных, в частности формируются правила использования и устанавливается в единицу счетчик пользователей объекта;

- если объект уже существует, просто увеличивает счетчик пользователей;
- в любом случае создает в таблице дескрипторов процесса соответствующую новую запись;
- формирует значение процессо-зависимого дескриптора (как возвращаемое функцией значение или по полученному в качестве параметра адресу).

Замечание: если создать объект по каким-то причинам не удалось (это может произойти при несоответствии прав или крайне редко при наличии каких-то проблем с защитой или при нехватке памяти), то некоторые функции возвращает 0, а другие значение INVALID_HANDLE_VALUE (-1). Так как вариантов неудачи несколько, то следует читать инструкции по использованию конкретной функции в справочной системе. Причину отказа можно узнать, вызвав системную функцию GetLastError(), а расшифровку отказа можно получить с помощью утилиты Error Lookup или в окне отладчика Watch интегрированной среды Visual Studio, набрав @err,hr.

2. Использование объектов.

Располагая дескриптором объекта, приложение может пользоваться ресурсом, передавая значение дескриптора в соответствующую системную функцию. Например:

```
BOOL ReadFile( HANDLE hFile, <другие параметры>);
```

При этом следует помнить, что при создании объекта формируются правила использования ресурса, и при любом обращении к ресурсу диспетчер объектов проверяет, есть ли у процесса права на выполнение требуемой операции.

Всякий раз, когда приложение вызываете функцию ОС, диспетчер объектов, используя дескриптор в качестве индекса, «заглядывает» в таблицу дескрипторов процесса и:

- сверяет права процесса по доступу к объекту (в таблице дескрипторов) с правилами использования (дескриптором защиты в

заголовке объекта);

- если права позволяют из таблицы извлекается адрес объекта;
- и управление передается соответствующей специализированной компоненте ОС для выполнения запроса;

Замечание: если системной функции передан неверный дескриптор, скорее всего, сработает защита, и функция просто ничего не будет делать. Статус завершения функции можно узнать, вызвав GetLastError(). В случае некорректного значения дескриптора функция GetLastError() вернет ERROR_INVALID_HANDLE.

В многозадачной ОС сплошь и рядом возникает ситуация, когда несколько процессов (или в общем случае несколько потоков) используют один и тот же ресурс, например:

- объекты «file mapping» позволяют двум процессам, исполняемым на одной машине, использовать одну и ту же физическую память;
- почтовые ящики («mailslots») и именованные каналы («named pipes») дают возможность обмениваться данными с процессами, исполняемыми на других машинах в сети;
- объекты синхронизации («mutex», «event», «semaphore») позволяют синхронизировать потоки, исполняемые в разных процессах;
- и т. д.

Но так как дескрипторы объектов процессо-зависимы (и таким образом защищены), совместное использование объектов – задача нетривиальная. Умение корректно совместно использовать разделяемые ресурсы является одним из предметов данного курса. Способов совместного использования объектов несколько:

- именованные объекты;
- унаследование объектов дочерним процессом;
- дублирование описателя родителем для дочернего процесса;
- «перенаправление» стандартного ввода/вывода.

3. Освобождение ресурса.

Независимо от того, какой специализированной функцией процесс получал доступ к ресурсу, по окончании работы с ресурсом дескриптор объекта следует закрыть системной функцией CloseHandle(). Вызов CloseHandle() важен, так как содержание объектов вещь дорогая. Прототип системной функции:

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

Но закрыть дескриптор – не значит уничтожить сам объект! Если дескриптор корректен, CloseHandle() по указанному адресу находит объект и уменьшает счетчик пользователей на единицу. И только в том случае, когда счетчик пользователей обнуляется, диспетчер объектов уничтожает сам объект, то есть освобождает занятую им область памяти.

Замечания:

- если переданный функции дескриптор неверен, функция CloseHandle() возвращает FALSE, а функция GetLastError() - ERROR_INVALID_HANDLE;
- перед самым возвратом функция CloseHandle() помечает соответствующую запись в таблице описателей процесса как недействительную, поэтому после вызова функции CloseHandle() процесс не имеет права пользоваться данным дескриптором и не получит доступа к объекту, хотя сам объект может еще продолжать существовать (это означает, что ресурс продолжает использоваться еще кем-то другим);
- если процесс «забыл» вызвать функцию CloseHandle(), при завершении процесса ОС гарантированно освобождает все ресурсы, принадлежащие данному процессу.

2. МНОГОЗАДАЧНОСТЬ

Можно сформулировать несколько точек зрения на термин многозадачность рис. 2.1:

• для пользователя – это, с одной стороны, очевидная возможность работы «одновременно» с несколькими приложениями, с другой стороны, иногда менее очевидная - выполнять несколько задач в одном приложении «одновременно» (например, таким образом в редакторе Word происходит фоновое сохранение данных);

Рис. 2.1. Точки зрения на многозадачность

• для разработчика (прикладного программиста) – это возможность: управлять пакетом приложений как единым целым, запускать другие приложения (процессы), а также обмениваться между разными процессами данными, дробить приложение на более мелкие независимые задачи (threads), которые могут выполняться «одновременно планировать переключение подзадач в рамках одной задачи самостоятельно (fibres);

• для ОС – это решение таких проблем, как разделение ресурсов между приложениями и обеспечение безопасности каждого приложения от других и самой ОС от приложений.

2.1. СПОСОБЫ ОРГАНИЗАЦИИ МНОГОЗАДАЧНОСТИ

В соответствии с перечисленными задачами ОС предоставляет программисту следующие способы организации многозадачности – рис. 2.2:

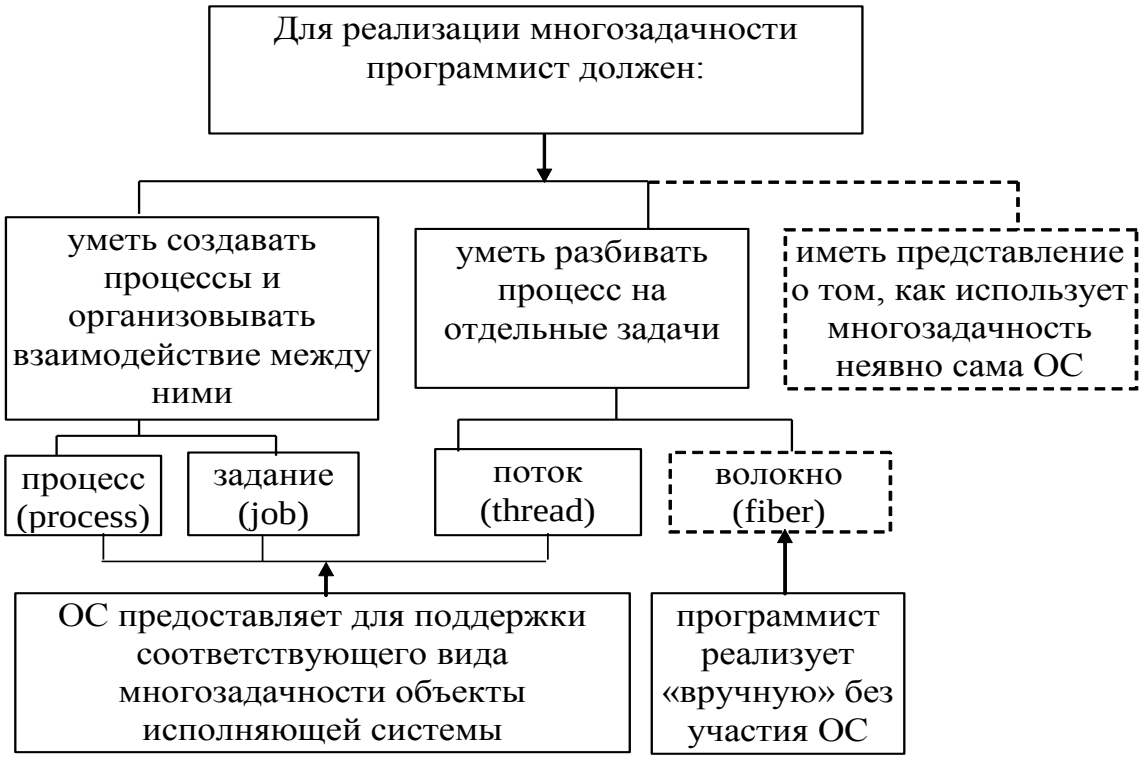

Рис. 2.2. Средства, предоставляемые ОС для организации многозадачности

Многозадачность посредством «заданий»

Задание (job) – это набор процессов с общими квотами и лимитами, например, максимальное количество процессов, суммарное время центрального процессора для каждого процесса и для задания в целом, максимальное количество используемой памяти для каждого процесса и для задания в целом и т. д. Процессы можно собрать в объект «задание» таким образом, чтобы ими можно было управлять как группой. Объект job позволяет управлять группой процессов, то есть операции, применяемые к заданию, на самом деле выполняются для всех процессов, ассоциированных с данным заданием Объекты «job» поддерживаются только начиная с версии ОС Windows 2000.

Последовательность действий программиста для использования заданий:

- создание нового объекта job или получение доступа к существующему объекту с помощью функции CreateJobObject() или использование существующего – функция OpenJobObject();
- добавление нового процесса в задание – функция AssignProcessToJob();
- завершение всех процессов задания – функция TerminateJobObject();
- закрыть дескриптор – функция CloseHandle(). Если при уничтожении объекта – «задания» еще существуют ассоциированные с ним процессы, они продолжают выполняться.

Реализация многозадачности посредством процессов

Такой вид многозадачности используется, если задача реализуется с помощью нескольких самостоятельных приложений. При этом нужно помнить, что каждое приложение выполняется в своем обособленном защищенном адресном пространстве – рис. 2.3. Так как одно приложение не имеет доступа к адресному пространству другого приложения, то основной проблемой становится обмен данными. Соответственно, прикладной программист должен знать

системные возможности для обмена данными и грамотно их использовать.

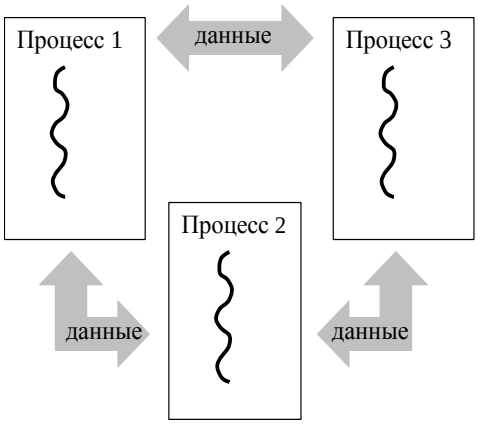

Рис. 2.3. Реализация многозадачности посредством процессов

Реализация многозадачности посредством потоков

Задачу можно в пределах одного процесса разбить на отдельные самостоятельные задачи посредством потоков (threads). При этом все потоки выполняются в одном адресном пространстве (рис. 2.4) и имеют доступ ко всем ресурсам процесса. Это облегчает обмен данными между задачами, но оборачивается проблемами синхронизации доступа к данным, поэтому прикладной программист должен знать системные возможности синхронизации выполнения задач.

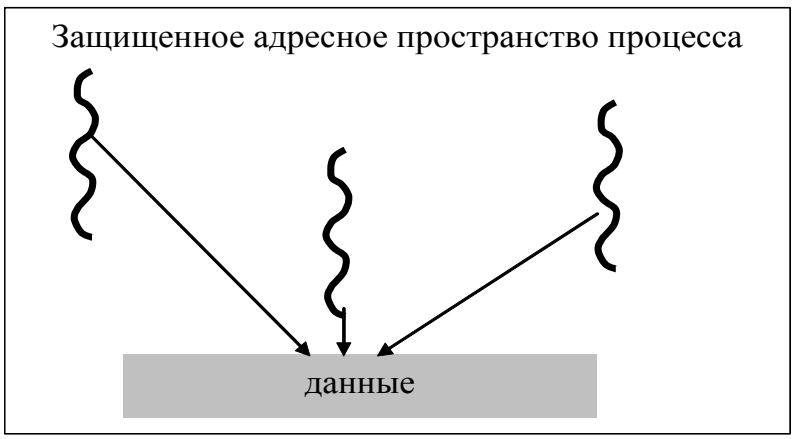

Рис. 2.4. Реализация многозадачности посредством потоков

Многозадачность посредством «волокон»

Переключение потоков – вещь дорогая, так как необходимо сохранить контекст, переключиться в режим ядра, загрузить новый контекст, вернуться в режим пользователя. Начиная с ОС Windows 2000, программисту предоставляется облегченный вариант псевдо параллелизма – волокна (fibers). Реализованные посредством волокон задачи выполняются в рамках одного потока. Последовательность выполнения (переключение) волокон программист планирует сам.

Замечания:

- волокно (fiber) не является объектом исполняющей системы;
- ОС ничего не знает о переключении волокон, так как они выполняются в контексте одного и того же потока;
- квант времени для выполнения предоставляется не волокну, а потоку, которому волокна принадлежат.

Неявно реализованная многозадачность

Даже если приложение состоит из единственного потока:

- выполняя некоторые запросы приложения, которые для программиста выглядят как вызовы обычных функций, ОС посредством механизмов RPC (Remote Procedure Call – удаленный вызов процедур) и LPC (Local Procedure Call – локальный вызов процедур) выполняет эти запросы в отдельном серверном потоке;
- поток может блокировать свое выполнение до наступления какого-нибудь события – таким образом, он неявно взаимодействует с каким-то другим процессом;
- также неявно реализуют многозадачность: асинхронный ввод/вывод, порты завершения (I/O completion ports), асинхронные вызовы процедур (APC) и т. д.

2.2. СРЕДСТВА ПОДДЕРЖКИ МНОГОЗАДАЧНОСТИ ОПЕРАЦИОННОЙ СИСТЕМОЙ

Для поддержки таких понятий как «процесс», «поток», «задание» ОС предоставляет соответствующие объекты исполнительной системы и системные функции для работы с этими объектами. Для организации взаимодействия потоков ОС предоставляет синхронизирующие объекты.

Почему для описания приложения существуют два отдельных понятия (процесс и поток) и соответственно разные объекты исполняющей системы? Основными критическими ресурсами при выполнении приложения являются время и память. Процесс описывает статику приложения и является контейнером для всех ресурсов, используемых приложением (выделяемую для приложения память, открытые файлы и другие ресурсы приложения). Для описания динамики выполнения приложения используется объект «поток» рис. 2.5.

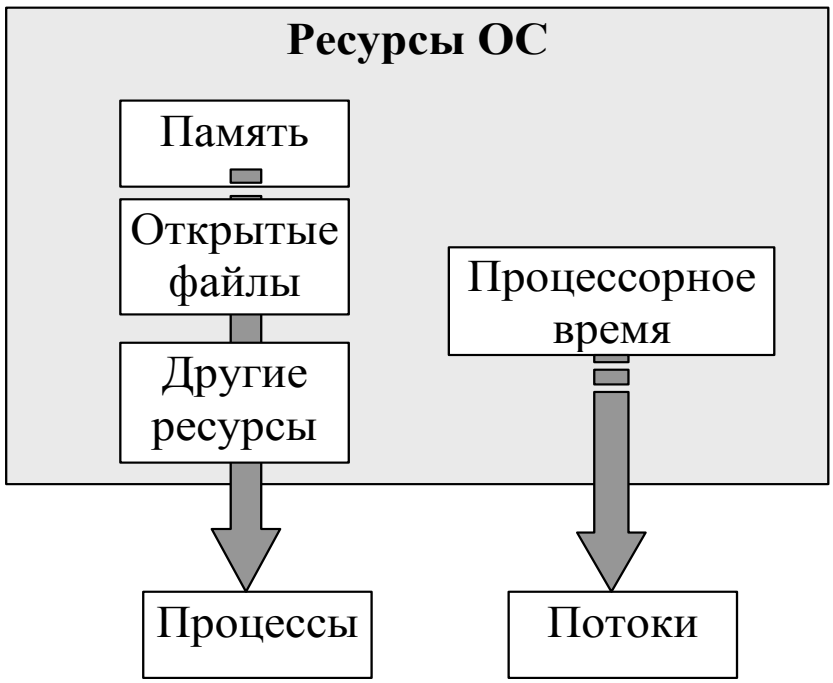

Рис. 2.5. Разделение ресурсов между процессами и потоками

2.3. ВЫБОР ТИПА МНОГОЗАДАЧНОСТИ

Выбор типа многозадачности в основном определяется конкретной задачей, но иногда возможно и эффективно только какое-нибудь одно решение. Рассмотрим применение двух основных типов многозадачности, основанных на процессах и потоках.

Реализация многозадачности посредством процессов

Совершенно очевидно, что если Ваше приложение живет обособленно от других, то никаких потребностей общаться с другими приложениями у него не возникает. Но как только возникает необходимость в распределенном решении задачи, то без реализации многозадачности посредством процессов не обойтись. Самый очевидный пример – это Internet-приложение, когда браузер работает локально в среде Windows и отображает Web-страницу, сервер Web – удаленно (на другом компьютере), возможно в другой операционной среде.

На самом деле и локально всегда может возникнуть необходимость общения Вашей программы с другим приложением. Например, Ваше приложение формирует данные, которые требуется отобразить в виде привычных для пользователя таблиц или графиков. Для такого представления данных прекрасно подходят электронные таблицы, например Excel. Если программист умеет оформить свои данные требуемым образом и передать их Excel для отображения, то отпадает необходимость в разработке большого объема кода.

Замечание: при разбиении задачи на процессы программист должен знать и эффективно применять системные механизмы для обмена данными между процессами.

Реализация ногозадачности посредством потоков

Многопоточный процесс дает возможность выполнения фоновых задач «одновременно» с основной или нескольких идентичных задач «параллельно». Для каких задач может понадобиться многопоточность:

- разделение задачи на несколько подзадач, последовательность выполнения которых несущественна, например, фоновое сохранение данных в редакторе Word, подсчет и разбиение на страницы. При грамотной организации фоновых потоков экономятся такие ресурсы как время, так как процессор работает гораздо быстрее, чем пользователь нажимает клавиши. В этом случае от программиста требуется корректно представить задачу в виде подзадач и, возможно, назначить отдельным потокам разные приоритеты;
- разделение задачи на несколько подзадач, когда «выход» одной подзадачи является «входом» для другой, например, для того, чтобы данные можно было обрабатывать, их сначала нужно сформировать (все или хотя бы часть). Для таких задач на первый план выходит синхронизация потоков;
- разделение задачи на несколько подзадач, когда выполнение одной подзадачи, например ввод данных с внешнего устройства, является приоритетной по отношению к другим действиям, например обработке данных. Внешнее устройство «ждать не будет», поэтому данные следует как можно быстрее считать, а обрабатывать, когда будет время.

Замечание: обычно реализация многопоточного процесса является более эффективным решением по сравнению с реализацией посредством процессов, так как:

- на создание потоков система тратит времени и ресурсов гораздо меньше, чем на создание процессов;
- переключение потоков требует относительно небольших затрат (сохранение и восстановление контекста потока);
- все потоки одного процесса разделяют одно адресное

пространство и имеют доступ к глобальным данным, поэтому не нужно задействовать системные механизмы для обмена данными;

- все потоки одного процесса имеют доступ к ресурсам, принадлежащим этому процессу, таким как открытые файлы, каналы и т. д.;
- потоки, принадлежащие разным процессам, не имеют между собой ничего общего, однако они могут получить доступ к одним и тем же ресурсам и общей памяти, используя системные механизм взаимодействия между процессами.

3. ПРОЦЕССЫ

Понятие процесса разное в разных ОС. В общем случае любая ОС содержит блок кода, управляющий созданием и удалением процессов, а также отношениями между ними. Этот код называется структурой процессов (process structure) и в ОС Windows NT реализован с помощью диспетчера процессов рис. 3.1. В разных ОС процессы реализованы по-разному. Процессы различаются своим представлением (структурами данных), способами именования и защиты и отношениями между собой.

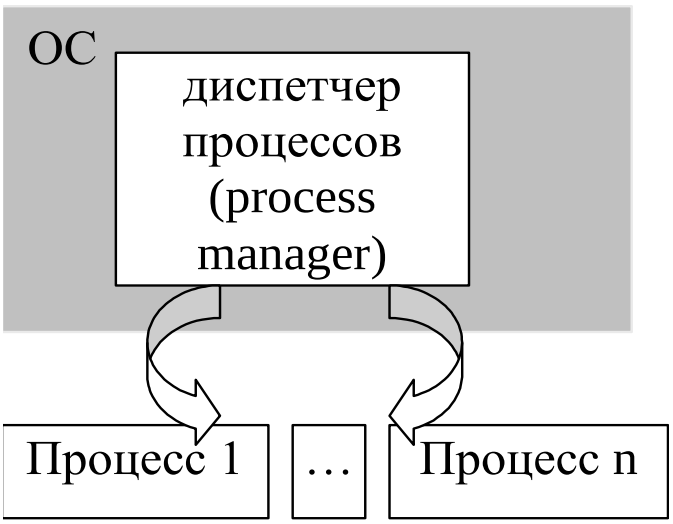

Рис. 3.1. Диспетчер процессов

В ОС Windows NT процесс – это единица работы и контейнер для всех ресурсов, которыми владеет приложение. Когда пользователь запускает приложение, ОС создает объект «процесс» (process). Также ОС формирует защищенное адресное пространство и располагает в этом пространстве код и данные приложения, а также таблицу дескрипторов данного процесса и инициализирует ее.

На самом высоком уровне абстракции процесс состоит из рис. 3.2:

- исполняемой программы (кода и данных);
- закрытого адресного пространства, то есть набора адресов виртуальной памяти, который процесс может использовать;
- системных ресурсов, выделяемых ОС процессу на время выполнения программы (коммуникационные порты, семафоры, файлы…), которые доступны всем потокам процесса;
- по крайней мере, одного первичного потока;
- маркера доступа (контекст безопасности), который содержит информацию о правах зарегистрированного в системе пользователя (группы безопасности и привилегии, ассоциированные с процессом);
- уникального идентификатора, который однозначно описывает данный процесс в системе.

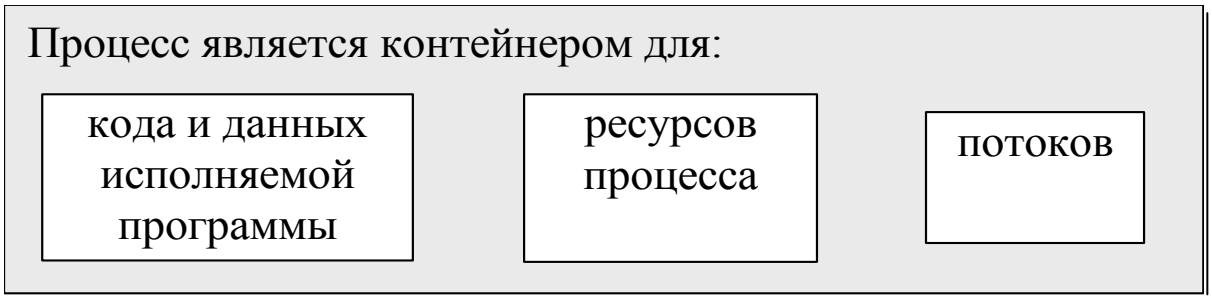

Рис. 3.2. Процесс можно представить как контейнер

3.1. МЕХАНИЗМ ИСПОЛЬЗОВАНИЯ ОБЪЕКТОВ ПРОЦЕССОМ

При инициализации процесса ОС создает в нем таблицу описателей, используемую только для объектов исполняющей системы. Эта таблица может быть реализована по-разному в разных ОС, но смысл – одинаков: приложение может создавать новые или пользоваться существующими объектами исполняющей системы только «косвенно» - посредством процессо-зависимых индексов (дескрипторов) в таблице описателей рис 3.3.

Рис. 3.3. Механизм владения объектами процесса

При запуске приложения таблица описателей почти пуста. Но стоит приложению вызвать системную функцию для получения доступа к ресурсу, например CreateThread(), CreateFile(), CreateFileMapping() и т.д., так в таблицу добавляется новая запись, а приложению возвращается индекс этой записи – дескриптор.

3.2. СИСТЕМНЫЕ СРЕДСТВА ДЛЯ СОЗДАНИЯ ПРОЦЕССА

Если нужно из одного приложения запустить другое приложение, необходимо создать новый процесс. При создании процессов употребляются термины: «родительский/дочерний», «master/slave», но такие отношения в действительности не поддерживаются Win32 – это только удобная и не очень корректная терминология. На самом деле корректнее было бы употреблять «порождающий / порожденный», но уж очень длинно. При этом между процессами может не существовать никаких взаимоотношений, кроме запуска, не говоря уже о «Я тебя породил – я тебя и убью!»

Для запуска другого приложения программисту предоставляются следующие системные функции:

- WinExec() считается устаревшей и не поддерживает строковые параметры, заданные в кодировке Unicode;
- LoadModule() – аналогично:
- CreateProcess() и его разновидности: CreateProcessAsUser() и CreateProcessWithLogonW();
- ShellExecute().

Замечание: посмотреть, сколько процессов запущено и их атрибуты можно с помощью:

- диспетчера задач (TaskManager);
- утилиты ProcessViewer, которая поставляется в составе Resource Kit for Windows NT и в составе SDK. Пользуясь этой утилитой, можно просмотреть и в некоторых случаях изменить приоритеты процессов, а также получить другую информацию о запущенных приложениях, например, использованное процессорное время с момента запуска, процент работы системного и пользовательского кода, использование виртуальной памяти и т. д.

3.3. СИСТЕМНАЯ ФУНКЦИЯ CreateProcess()

Основным средством запуска приложений является системная функция CreateProcess(), так как она обладает широким спектром возможностей.

Замечание: эквивалентами этой системной функции являются функции стандартной библиотеки C spawn() и exec(). Если приложение выполняется в среде Windows, то на самом деле эти функции в свою очередь обращаются к функции CreateProcess().

Когда приложение вызываете функцию CreateProcess() ОС:

- находит нужный исполняемый файл;
- создает объект исполняющей системы «процесс» с начальным значением счетчика числа его пользователей, равным 1;
- формирует для нового процесса виртуальное адресное пространство и загружает в него код и данные исполняемого файла, а также требуемые динамические библиотеки (DLL);
- инициализирует таблицу дескрипторов в создаваемом процессе;
- создает объект «поток», который соответствует первичному потоку запускаемого процесса, и выполняется совершенно независимо от породившего процесса (первичный поток начинает выполнение стартового кода из стандартной библиотеки С);
- перед самым возвратом из функции формирует в таблице дескрипторов родительского процесса записи, относящиеся к объектам «процесс» и «поток» дочернего процесса, и дескрипторы этих объектов возвращает родительскому процессу (это в дальнейшем обеспечивает возможность взаимодействия родительского и дочернего процессов);
- увеличивает счетчики каждого объекта еще на 1, так как в лице родительского процесса у созданных для дочернего процесса объектов появляется еще один пользователь.

Прототип функции:

```
BOOL CreateProcess(//Если системе удается создать новый
                   процесс и его первичный поток, она
                   возвращает TRUE
    LPCTSTR lpApplicationName,  //имя исполняемого файла или 0,
                   если имя указано в командной строке
    LPTSTR lpCommandLine,   //командная строка
    LPSECURITY_ATTRIBUTES lpProcessAttributes, //атрибуты
                   безопасности создаваемого процесса
    LPSECURITY_ATTRIBUTES lpThreadAttributes, //атрибуты
                   безопасности первичного потока
    BOOL bInheritHandles,   //наследует ли дочерний процесс
                   дескрипторы родителя
    DWORD dwCreationFlags, //параметры создания потока
    LPVOID lpEnvironment, //значения переменных окружения или
                   0, если наследуется окружение порождающего
                   процесса
    LPCTSTR lpCurrentDirectory //текущий каталог или 0, если
                   используется текущий каталог родителя
    LPSTARTUPINFO lpStartupInfo, //задаваемая информация о
                   запускаемом процессе
    LPPROCESS_INFORMATION lpProcessInformation// сюда
                   возвращается функцией информация о дочернем
                   процессе
);
```

Рассмотрим параметры этой функции.

lpApplicationName, lpCommandLine

Два первых параметра определяют имя исполняемого файла и командную строку, передаваемую создаваемому процессу. Следует помнить, что тип параметров может быть разным, в зависимости от кодировки (ANSI и UNICODE).

Существуют несколько комбинаций значений этих параметров. Так как это основная функция запуска процессов, рассмотрим возможные варианты.

1. Если *lpApplicationName*!=0 (параметр введен для поддержки подсистемы POSIX операционной системой NT), а *lpCommandLine*==0, то нужно учитывать следующую специфику:

• обязательно должен быть указан полный путь к исполняемому файлу, а если указано только имя, то файл должен находиться в

текущем каталоге;

- обязательно должно быть указано расширение файла (.exe);
- нельзя передать дополнительные параметры командной строки.

Например:

```
CreateProcess( "c:\\MyDir\\MyApp.exe", 0, ... ) // Замечание:  так как в строковых литералах символ «\» является служебным, то его следует удвоить
CreateProcess( "MyApp.exe Param1 Param2", 0, ... ) //завершится ошибкой "Файл не найден"
```

2. Обычно в Win32-приложении используется комбинация *lpApplicationName*==0, а *lpCommandLine!*=0. В этом случае *lpCommandLine* содержит командную строку, которую функция CreateProcess() «разбирает» в следующем порядке:

- извлекает первый (до пробела) компонент и интерпретирует его как имя файла. Если в имени файла не указано расширение, то считает, что по умолчанию нужно искать файл с расширением “.exe”;
- приступает к поиску файла в следующем порядке:
  - если в имени файла указан полный путь, ищет там, где указано;
  - в каталоге, содержащем исполняемый файл родителя;
  - в текущем каталоге родительского процесса;
  - в системном каталоге Windows (System32);
  - в основном каталоге Windows;
  - в каталогах, перечисленных в переменной окружения PATH;
- разбирает все остальные компоненты строки как параметры командной строки.

Пример:

```
CreateProcess( 0, "MyApp Param1 Param2", ...) //запустит MyApp.exe с параметрами командной строки
```

Полезные системные функции для получения путей к файлу: GetCurrentDirectory(), GetFullPathName(), GetSystemDirectory().

3. Если одновременно используются оба ненулевых параметра

(эта возможность введена для поддержки подсистемы POSIX операционной системой WindowsNT), могут возникать неоднозначные ситуации. Так как согласно стандарту имя файла является первым параметром командной строки, то это имя придется указывать в параметрах функции дважды:

CreateProcess(“**MyApp.exe**”, "**MyApp** Param1 Param2", ...);

А такой вызов будет означать, что ОС запустит Notepad, но поместит в его командную строку WORDPAD README.TXT:

CreateProcess("C:\\WINNT\\SYSTEM32\\NOTEPAD.exe", "WORDPAD README.TXT", ...)

lpsaProcess, lpsaThread

При создании объектов «процесс» и «поток» им можно сопоставить атрибуты защиты. Из порождающего процесса задание этих атрибутов можно осуществить с помощью параметров lpsaProcess, lpsaThread:

- если они равны 0, система закрепит за данными объектами дескрипторы защиты по умолчанию;
- если родительский процесс хочет создать новые объекты с их собственными атрибутами защиты, необходимо предварительно создать и проинициализировать две структуры SECURITY_ATTRIBUTES.

bfInheritHandles

Это признак, с помощью которого можно указать ОС передать дочернему процессу «по наследству» все наследуемые объекты родителя. Если этот параметр имеет значение TRUE, то ОС, формируя таблицу описателей дочернего процесса, копирует из таблицы родительского все наследуемые описатели в те же позиции рис. 3.4. Последний факт чрезвычайно важен, так как это означает, что описатели будут идентичны (их значения будут одинаковыми) в обоих процессах!

Рис. 3.4. Механизм наследования объектов

Помимо копирования таблицы описателей система увеличивает значения счетчиков пользователей всех объектов, так как объекты теперь могут быть использоваться дочерним процессом также.

Замечания:

- только в случае наследования одному и тому же объекту в разных процессах будет соответствовать один и тот же дескриптор;
- наследуются только описатели наследуемых объектов, существующие в родительском процессе на момент создания дочернего процесса. Если родитель после вызова создаст еще какие-

нибудь объекты, то они уже будут недоступны дочернему процессу;

- при наследовании описателей объектов дочерний процесс не имеет ни малейшего представления о тех описателях, которые он наследует. В зависимости от конкретной задачи иногда требуется именно такое «неведение» (например, перенаправление стандартного ввода/вывода). Во всех остальных случаях нужно известить дочерний процесс о таких объектах, передав ему значение дескриптора.

dwCreationFlags

Этот параметр объединяет две группы флагов, которые можно комбинировать поразрядным «ИЛИ»:

- способ создания нового процесса (здесь можно указать несколько флагов);
- приоритет дочернего процесса (можно указать только одно значение).

Наиболее интересными флагами создания дочернего процесса являются:

- CREATE_SUSPENDED позволяет создать процесс (то есть сформировать соответствующие структуры данных в памяти), и в то же время приостановить его первичный поток;
- CREATE_NEW_CONSOLE предписывает ОС создать для дочернего консольного процесса собственное консольное окно и собственные потоки ввода/вывода. В противном случае (если флаг отсутствует), дочерний процесс будет пользоваться окном и потоками ввода/вывода родителя (при этом нет возможности определить, какому из двух процессов предназначен ввод);
- CREATE_NO_WINDOW имеет смысл для процессов, не взаимодействующих с пользователем, а осуществляющих какие-нибудь вспомогательные действия (вычисления);
- DEBUG_PROCESS позволяет родительскому процессу производить отладку дочернего. При этом родителю поступают все события отлаживаемого процесса.

Вторая группа флагов задает приоритет запускаемого процесса.

Можно указать одно из значений: REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS и, возможно, некоторые другие значения.

Замечания:

- по умолчанию (если не указано ни одно из значений) в большинстве случаев у дочернего процесса будет нормальный приоритет;
- запускать процессы с приоритетом выше NORMAL_PRIORITY_CLASS пользователь имеет право, только если ему это позволено, то есть у него есть права администратора.

lpEnvironment

Содержит адрес блока памяти, в котором сформированы строки переменных окружения для дочернего процесса. Обычно в качестве этого параметра передается значение 0, что означает, что дочерний процесс наследует строки окружения от родительского процесса. Родительский процесс может также подготовить новый блок среды, состоящий из расположенных друг за другом символьных строк, завершающихся нулем. Последняя из этих строк должна заканчиваться двумя нулями.

Замечание: при необходимости процесс может получить адрес блока среды при помощи функции GetEnvironmentStrings().

lpCurrentDirectory

Позволяет родительскому процессу установить текущий диск и каталог для дочернего процесса. Если в качестве этого параметра передается 0, рабочий каталог порожденного процесса будет тот же, что и у родителя.

Замечание: если родитель формирует для дочернего процесса специфический каталог, необходимо указать полный путь, включая букву диска.

lpStartupInfo

Для запуска процесса нужно создать структурную переменную типа STARTUPINFO и заполнить ее поля. Это та информация, которую будет использовать ОС для формирования внешнего вида и

свойств запускаемого приложения. В параметре lpStartupInfo передается адрес заполненной структуры. Большинство приложений порождают новые процессы с атрибутами по умолчанию, в этом случае минимальные действия программиста выглядят так:

```
STARTUPINFO si; //отвести память
ZeroMemory( &si, sizeof(si) ); //заполнение полей значением
                по умолчанию 0
si.cb = sizeof(si); //размер структуры в байтах
```

Формат структуры:

```
typedef struct _STARTUPINFO
{
  DWORD  cb;              // размер структуры в байтах
  LPTSTR lpReserved;      // зарезервировано
  LPTSTR lpDesktop;    // рабочий стол и станция для процесса
  LPTSTR lpTitle;      // заголовок окна консольного процесса
  DWORD  dwX;             // координата угла окна в пикселах
  DWORD  dwY;             // координата угла окна в пикселах
  DWORD  dwXSize;         // ширина окна в пикселах
  DWORD  dwYSize;         // высота окна в пикселах
  DWORD  dwXCountChars;   // размер буфера консольного окна по
                  x (если размер меньше, появляется скроллинг
                  и resize-ить окно можно только до этих
                  размеров)
  DWORD  dwYCountChars;   // размер буфера консольного окна по
                  y
  DWORD  dwFillAttribute; // атрибуты текста консольного окна
  DWORD  dwFlags;         // какие поля структуры следует учитывать
  WORD   wShowWindow;     // как отображать главное окно
  WORD   cbReserved2;     // зарезервировано
  LPBYTE lpReserved2;     // зарезервировано
  HANDLE hStdInput;       // дескриптор потока ввода
  HANDLE hStdOutput;      // вывода
  HANDLE hStdError; // сообщений об ошибках
} STARTUPINFO, *LPSTARTUPINFO;
```

Существенно:

• разные поля структуры имеют смысл для приложений разных типов (консольных и оконных), причем для консольных приложений флаги будут учтены, только если при вызове функции CreateProcess()

установлен флаг CREATE_NEW_CONSOLE;

- нулевое значение поля подразумевает умолчание;
- поле cb содержит размер структуры STARTUPINFO в байтах (если поле нулевое, то используется размер структуры по умолчанию, но для лучшей переносимости программы лучше задавать это значение явно);
- поле dwFlags является битовой маской, посредством которой родительский процесс указывает ОС, какие остальные поля структуры следует учитывать.

Поля структуры можно разделить на следующие группы:

- общие (имеют смысл для приложения любого типа) – cb, dwFlags, dwX, dwY, dwXSize, dwYSize, lpDesktop, hStdInput, hStdOutput, hStdError;
- поля, имеющие смысл только для консольного приложения – dwXCountChars, dwYCountChars, dwFillAttribute, lpTitle;
- поля, имеющие смысл только для оконного (GUI) приложения - wShowWindow.

Рассмотрим некоторые поля более подробно.

1. Поле dwFlags содержит флаги, определяющие, содержимое каких полей структуры STARTUPINFO необходимо учитывать при запуске нового процесса - табл. 3.1, а также три дополнительных флага:

Таблица 3.1

Соответствие флагов поля dwFlags и задаваемых полей

Значение	Используемые поля
STARTF_USESHOWWINDOW	wShowWindow
STARTF_USEPOSITION	dwX, dwY
STARTF_USESIZE	dwXSize, dwYSize
STARTF_USECOUNTCHARS	dwXCountChars, dwYCountChars
STARTF_USEFILLATTRIBUTE	dwFillAttribute
STARTF_USESTDHANDLES	hStdInput, hStdOutput и hStdError

Дополнительными являются флаги:
STARTF_FORCEONFEEDBACK, STARTF_FORCEOFFFEEDBACK и STARTF_SCREENSAVER.

Использование первого из этих флагов приводит к тому, что при запуске приложения курсор мыши принимает форму песочных часов со стрелкой. При этом пользователь может работать с другими приложениями, зная, что процесс запуска данного приложения еще не завершен. Как только приложение будет запущено и станет активным (после первого вызова функции GetMessage()), курсор примет свою обычную форму.

Если же указан флаг STARTF_FORCEOFFFEEDBACK, в процессе запуска приложения курсор имеет обычную форму.

Флаг STARTF_SCREENSAVER вызывает снижение класса приоритета после запуска до IDLE_PRIORITY_CLASS, несмотря на то, что при инициализации класс приоритета был NORMAL_PRIORITY_CLASS. Если пользователь активизирует окно такого приложения, его приоритет автоматически повышается.

2. Поле lpDesktop используется для выбора рабочего стола и рабочей станции, где будет запущен процесс. Если вы собираетесь использовать текущий рабочий стол и текущую рабочую станцию, укажите в этом поле значение 0.

3. Для консольного процесса (и только для него) вы можете указать заголовок окна в поле lpTitle. Если же в этом поле задать нулевое значение, для заголовка будет использовано имя файла.

Например:

```
ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
si.lpTitle = «MyWindow»;
```

4. Поля dwX, dwY dwXSize определяют, соответственно, координаты X и Y левого верхнего угла окна приложения и его размеры в пикселях. Эти значения используются для позиционирования главного окна приложения, являются рекомендательными и будут использованы только в том случае, если

при создании окна дочерним процессом функцией CreateWindow() расположение и размеры окна указаны посредством дефолтовых констант CW_USEDEFAULT. Например:

```
ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
si.dwFlags = STARTF_USEPOSITION;
si. dwX =100;
si. dwY =100;
```

5. Поля dwXCountChars и dwYCountChars задают, соответственно, ширину и высоту окна консольного приложения в символах. Если одновременно задан размер окна в пикселях, и он меньше, чем в символах, то автоматически появляются полосы прокрутки.

6. Содержимое поля dwFillAttribute задает цвет текста и фона для окна консольного приложения. Например:

```
ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
si.dwFlags = STARTF_USEFILLATTRIBUTE;
si. dwFillAttribute = FOREGROUND_RED| BACKGROUND_GREEN;
                //задает красный текст на зеленом фоне
```

6. Поле wShowWindow определяет состояние главного окна GUI дочернего процесса (в каком виде должно отображаться окно - в нормальном, минимизированном, максимизированном и т. д.). Это то значение, которое будет использовано при первом вызове функции ShowWindow() для главного окна дочернего приложения. Например:

```
si.dwFlags = STARTF_USESHOWWINDOW;
si.wShowWindow = SW_SHOWMAXIMIZED;
```

7. Поля hStdInput, hStdOutput и hStdError определяют дескрипторы буферов, которые используются приложениями, соответственно, для ввода, вывода пользовательской информации и вывода сообщений об ошибках. Если при запуске дочернего процесса эти поля не заданы, то соответствующие объекты создаются в стартовом коде стандартной библиотеки С. При этом для консольного приложения ввод осуществляется с клавиатуры, а вывод в

стандартное текстовое окно. Если при запуске дочернего процесса родительский процесс формирует специфические дескрипторы, это называется перенаправлением стандартного ввода/вывода. Смысл такого приема заключается в следующем: если приложение запускается автономно, то ввод/вывод осуществляется посредством тех объектов, которые создаются стартовым кодом, а если то же самое приложение запускается как дочерний процесс, то тот же самый код может читать/писать в файл, канал и т. д.

При перенаправлении стандартного ввода/вывода важно помнить:

- передаваемые дескрипторы должны быть наследуемыми;
- при создании дочернего процесса параметр bInheritHandles функции CreateProcess() должен быть установлен в TRUE.

Замечание: для получения копии своей STARTUPINFO-структуры:

```
STARTUPINFO si;
si.cb = sizeof(si);
GetStartupInfo(&si);
```

8. Поле lpProcessInformation должно содержать адрес структурной переменной типа PROCESS_INFORMATION. Если поля вышеописанной структуры STARTUPINFO системная функция CreateProcess() использует для инициализации дочернего процесса, то поля структуры PROCESS_INFORMATION ОС заполняет для родительского процесса данными о дочернем процессе. Поэтому достаточно зарезервировать память и передать адрес этого блока в качестве поля структуры. Вид структуры:

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

Важно: создание нового процесса начинается с создания двух объектов исполняющей системы «процесс» и «поток». Для

родительского процесса ОС формирует записи об этих объектах в таблице дескрипторов и в полях структуры hProcess и hThread возвращает дескрипторы этих объектов для того, чтобы родительский процесс мог посредством этих дескрипторов взаимодействовать с дочерним. Эти значения имеют смысл только в родительском процессе и скорее всего не совпадают с дескрипторами тех же объектов дочернего процесса.

Кроме того, созданным процессу и потоку присваиваются уникальные идентификаторы, которые функция CreateProcess() заносит в поля dwProcessId и dwThreadId**.** Эти значения однозначно описывают оба объекта в системе. Используя эти идентификаторы, как родительский процесс, так и любой другой могут обращаться к созданному процессу и его первичному потоку. Например, системная функция OpenProcess(), используя идентификатор, позволяет получить дескриптор процесса.

3.4. ПОЛУЧЕНИЕ ИНФОРМАЦИИ О «СВОЕМ» ПОТОКЕ ИЛИ ПРОЦЕССЕ

Некоторые системные функции требуют в качестве параметра дескриптор процесса или потока. Для получения дескрипторов своего процесса используется системная функция:

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

Возвращаемое этой функцией значение является псевдо дескриптором и имеет значение -1. Несмотря на это, не рекомендуется использовать константу -1 явно. Возвращаемое значение можно использовать при вызове любой системной функции, требующей в качестве параметра дескриптор процесса. Закрывать этот дескриптор (с помощью функции CloseHandle()) не требуется. Настоящий дескриптор своего процесса можно получить посредством функции DuplicateHandle().

Для получения псевдо дескриптора своего потока используется системная функция:

```
HANDLE GetCurrentThread(VOID); // имеет смысл только при
                               использовании его в контексте текущего
                               потока
```

В набор системных функций включены также несколько функций, требующих идентификатор процесса или потока. Запросить идентификатор можно посредством системных функций:

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

3.5. СПОСОБЫ ЗАПУСКА ДОЧЕРНИХ ПРОЦЕССОВ С ПОМОЩЬЮ ФУНКЦИИ CreateProcess()

Взаимодействие родительского и дочернего процессов может осуществляться несколькими способами.

Обособленный (detached) дочерний процесс

Если после создания дочернего процесса родителю не нужно с дочерним процессом общаться и выполнение родителя никак не синхронизировано с выполнением дочернего процесса, то от родительского процесса требуется только запуск. Дальше оба приложения функционируют совершенно независимо. Например, именно так работают утилиты “проводник” (Explorer), Windows Commander и т.д.

При таком подходе порождающему процессу следует сразу же после создания дочернего процесса закрыть свои описатели дочернего «процесса» и его первичного «потока». Пример:

```
PROCESS_INFORMATION pi;
BOOL bSuccess = CreateProcess(<параметры>, &pi);
if(bSuccess)
{
     CloseHandle( pi.hThread );
     CloseHandle( pi.hProcess );
}
```

Важно: рекомендуется закрывать все неиспользуемые дескрипторы. Дескрипторы, как дочернего процесса, так и его первичного потока, полученные во владение родителем, должны быть

родителем и закрыты, поэтому если родителю они не нужны, лучше их закрыть сразу же после создания дочернего процесса.

Что происходит, если родительское приложение этого не сделает? Неэффективное использование виртуального адресного пространства ОС. Независимо от того, завершился дочерний процесс или нет, объекты «процесс» и «поток» будут существовать до тех пор, пока счетчики пользователей этих объектов не обнулятся. После запуска дочернего процесса счетчики объектов имеют значение 2. Если первым завершается дочерний процесс, то счетчики уменьшаются на единицу, но сами объекты продолжают существовать, бесцельно тратя память ОС. И только при завершении процесса-родителя ОС сама вызовет для всех таких объектов функцию CloseHandle(), счетчики примут нулевое значение, и память, наконец, будет освобождена.

Синхронизация родительского и дочернего процессов

В общем случае родительский и дочерний процесс не связывают отношения подчиненности. Допустим, выполнение родительского и дочернего процессов должно быть синхронизировано. Например, родитель может продолжить выполнение только после завершения дочернего процесса. Это можно сделать с помощью системной функции ожидания WaitForSingleObject().

```
{
    STARTUPINFO si; //элементы этой структуры используют Win32-
                    функции для создания нового процесса.
                    Большинство приложений порождают процессы с
                    атрибутами по умолчанию. Это означает, что
                    достаточно проинициализировать все
                    неиспользуемые поля нулем
    PROCESS_INFORMATION pi; //элементы этой структуры
                    заполняются самой функцией CreateProcess()
                    и будут содержать в частности дескрипторы
                    дочернего процесса и его первичного потока
    //Заполнение полей значением по умолчанию
    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si); //формирование минимума данных
```

```
    //Создание дочернего процесса
    if( !CreateProcess(0,
        "имя_исполняемого_файла",  //командная строка
        0,      // защита по умолчанию
        0,      // "-"
        FALSE,      // объекты родителя не наследуются дочерним
                    процессом
        0,          // приоритет NORMAL.
        0,      // переменные окружения родителя
        0,      // текущая директория родителя
        &si,
        &pi)
    )
    {
        MessageBox(0, "CreateProcess failed.", "Info", MB_OK );
        return;//дальше продолжать выполнение функции не имеет
                    смысла
    }
    //Закрывайте описатели, как только необходимость в них
                    отпадает!!!
    CloseHandle( pi.hThread );  //уменьшает счетчик объекта
                    «Поток». Это не приводит к завершению
                    потока, а уменьшает значение счетчика
                    пользователей данного объекта
    // Ждем, пока дочерний поток завершится
    if(WaitForSingleObject( pi.hProcess, INFINITE
                    )!=WAIT_FAILED)     //подробное
                    рассмотрение отложим до темы «Потоки», а
                    сейчас ограничимся определением: вызов
                    данной функции задерживает выполнение кода,
                    пока объект, определяемый первым параметром
                    (HANDLE hObject) не перейдет в «свободное»
                    состояние. То есть в нашем случае вызов
                    функции приостанавливает выполнение потока
                    родительского процесса до завершения
                    порожденного им дочернего процесса
    {
//Дочерний процесс завершился
    DWORD dwExitCode;
    GetExitCodeProcess(pi.hProcess, &dwExitCode);//получаем
                    код завершения
    }
```

```
    // Закрывайте описатели, как только необходимость в них
                    отпадает
    CloseHandle( pi.hProcess );
}
```

Обладая дескриптором дочернего процесса, родитель (если он обладает достаточными правами) может в некоторой степени управлять дочерним процессом при помощи Win32-функций. Соответствующие системные функции приведены в табл. 3.2.

Таблица 3.2

Функции управления и получения информации

Когда	Win32 API	Что делает	Требуемый уровень доступа
После завершения и во время выполнения	GetExitCodeProcess	получает код завершения дочернего процесса. Возвращает STILL_ACTIVE или код завершения	PROCESS_QUERY_INFORMATION
Во время выполнения:	CreateRemoteThread	создает поток в другом процессе	PROCESS_CREATE_THREAD
	GetProcessTimes	степень использования CPU процессом (создание, завершение, работа в пользовательском режиме и в режиме ядра)	PROCESS_QUERY_INFORMATION

Продолжение табл. 3.2

Когда	Win32 API	Что делает	Требуемый уровень доступа
	GetThreadTimes	аналогична	PROCESS_QUERY_INFORMATION
	GetPriorityClass SetPriorityClass GetThreadPriority SetThreadPriority	Получать или изменять класс приоритета процесса или относительный приоритет его потока	PROCESS_SET_INFORMATION
	GetGuiResources	Определить – сколько объектов User и GDI использует процесс	PROCESS_QUERY_INFORMATION
	SetProcessWorkingSetSize GetProcessWorkingSetSize	Получить или установить минимальный и максимальный допустимый объем физической ОП, используемый процессом	PROCESS_SET_QUOTA
Завершить	TerminateProcess	Аварийное завершение	PROCESS_TERMINATE

Окончание табл. 3.2

Когда	Win32 API	Что делает	Требуемый уровень доступа
Ожидать завершения инициализации дочернего процесса	WaitForInputIdle	Ждет завершения инициализации дочернего процесса	
Ожидать завершения дочернего процесса	WaitForSingleObject WaitForMultipleObjects	Ожидать завершения одного или нескольких процессов	SYNCHRONIZE
	MsgWaitForMultipleObjects	Позволять родителю ожидать завершения дочернего процесса, но при этом пробуждаться и обрабатывать все или некоторые свои события	SYNCHRONIZE

Приведем пример синхронизации процессов посредством функции MsgWaitForMultipleObjects().

Замечание: для оконных приложений эта функция предпочтительнее других функций ожидания, так как позволяет обрабатывать все, или только указанные события, в частности дает возможность перерисовки окна, иначе возникает не очень приятный для пользователя эффект.

```
while(true)
{
    DWORD res = MsgWaitForMultipleObjects(1,//количество
                ожидаемых объектов
                &pi.hProcess,//в общем случае указатель на
                массив «ожидаемых» дескрипторов
                FALSE, //TRUE – ждать освобождения всех
                объектов + событие
                FALSE - когда любой из ожидаемых объектов
                освободился или событие
                INFINITE,//интервал ожидания в мс
                QS_ALLEVENTS);//тип события - QS_KEY |
                QS_MOUSE

                возвращает индекс освободившегося объекта
                от WAIT_OBJECT_0 до
                WAIT_OBJECT_0+количество ожидаемых
                объектов.
    if(res==WAIT_OBJECT_0) break;// дождались, так как у
                нас всего один объект
//Сюда попадем, если процессу поступило событие и его нужно
                обработать
    MSG m;
//Эмулируем цикл обработки сообщений
    while(PeekMessage(&m,0,0,0,PM_REMOVE))//если в очереди
                есть сообщение – удаляем из очереди и
                обрабатываем, иначе функция PeekMessage()
                неизрасходованный квант «предаст» другому
                потоку
    {
        DispatchMessage(&m);//”обрабатываем» сообщение
    }

}
```

3.6. СИСТЕМНАЯ ФУНКЦИЯ ShellExecute()

Для запуска процесса можно использовать вторую рекомендуемую системную функцию ShellExecute(). Эта функция:

- обладает меньшими возможностями по сравнению с функцией CreateProcess();
- специализирована для операций над файлами и каталогами;

• проще в использовании (по сравнению с функцией CreateProcess()) и имеет приятные особенности, например, поддерживает вызов приложений по расширению файла, зарегистрированного в ОС. То есть если в качестве параметра указан файл с расширением .txt, то будет запущен редактор Notepad или другая программа, которая ассоциирована в Вашей системе с расширением .txt для просмотра текстовых файлов.

Можно использовать функцию ShellExecute() следующими способами:

• запуск соответствующей утилиты Shell (например, Explorer);

• запуск любого приложения;

• запуск приложения по ассоциированному с ним расширению файла.

Прототип функции:

```
HINSTANCE ShellExecute(//возвращаемое значение – если >=32 –
                       OK, иначе причина неудачи
    HWND hwnd, //дескриптор окна (этому окну будет поступать
                       окна сообщений MessageBox, в которых
                       дочерний процесс может сообщать об ошибках)
                       или 0
    LPCTSTR lpOperation, //какое действие нужно совершить:
                       "open" –запустить исполняемый файл или
                       открыть документ или папку. Имя того, что
                       нужно открыть указано в lpFile
                       "explore" – изучать
                       «edit» - загружает редактор и открывает
                       указанный в lpFile документ для
                       редактирования. Если lpFile не имя
                       документа – fail
                       "find" – поиск, начиная с директории,
                       указанной в lpFile
                       "print" печатает документ, указанный в
                       lpFile (не все документы поддерживают эту
                       возможность)
    LPCTSTR lpFile, //имя файла или директории
    LPCTSTR lpParameters, //если в lpFile указано имя
                       исполняемого файла, - параметры командной
                       строк для передачи, если lpFile – имя
                       документа, то должен быть равен 0
    LPCTSTR lpDirectory, //директория по умолчанию или 0
```

```
    INT nShowCmd //как отобразить главное окно (если lpFile –
                   имя исполняемого файла) – это значение
                   константы для ShowWindow() – обычно
                   SW_SHOWNORMAL
                   если lpFile – имя документа, то значение
                   передается ассоциированному с этим
                   расширением приложению (а  приложение может
                   использовать его или нет по своему
                   усмотрению)
);
```

Замечание: второй расширенный вариант функции ShellExecuteEx() формирует дескриптор запущенного процесса.

Примеры использования функции ShellExecute():

1. чтобы открыть корневой каталог диска C:

```
ShellExecute(handle, "open", «c:\\», 0, 0, SW_SHOWNORMAL);
ShellExecute(handle, "explore", «c:\\», 0, 0, SW_SHOWNORMAL);
```

2. для запуска утилиты поиска:

```
ShellExecute(handle, "find", <путь к директории>, 0, 0, 0);
```

3. пусть требуется вызвать из своего приложения приложение Excel и передать ему данные посредством файла с именем «Macros.xls», который находится в Вашей текущей директории. Для того чтобы использовать функцию CreateProcess(), нужно каким-то образом найти директорию (например, посредством функции FindExecutable()), в которой находится файл Excel.exe или “прописать” путь к нему в переменной окружения PATH. Для такой задачи гораздо удобнее использовать функцию ShellExecute():

```
ShellExecute(0,“open”, "Excel", "Macros.xls", 0,
                   SW_SHOWNORMAL);
или ShellExecute(0,“open”, "Macros.xls", 0, 0, SW_SHOWNORMAL);
                   //запуск Excel по ассоциации с расширением
                   .xls
```

3.7. ЗАВЕРШЕНИЕ ПРОЦЕССА

Способы завершения процесса:

- самый естественный, если все потоки процесса завершаются;
- если естественным образом завершается первичный поток процесса, то управление возвращается стартовому коду стандартной

библиотеки, в котором вызывается функция ExitProcess(), завершающая в свою очередь все остальные еще выполняющиеся потоки;

- один из потоков процесса вызывает функцию ExitProcess();
- при вызове функции TerminateProcess(), которую может вызвать кто угодно, обладая дескриптором процесса и соответствующими правами. Это аварийное завершение процесса, поэтому использовать этот способ рекомендуется только в крайнем случае. Именно так поступает диспетчер задач (Task Manager), когда пользователь хочет завершить «зависшее» приложение.

Прототипы системных функций:

```
VOID WINAPI ExitProcess(
  UINT uExitCode  // код завершения процесса
);
BOOL TerminateProcess (
  HANDLE hProcess,   // дескриптор завершаемого процесса
  UINT   uExitCode); // код завершения процесса
```

Независимо от способа завершения, при завершении процесса:

- его код и выделенные ему ресурсы удаляются из памяти;
- закрываются дескрипторы всех объектов, созданных потоками процесса (которые явно не закрыл программист);
- все потоки процесса завершают свое выполнение.

4. ОБМЕН ДАННЫМИ МЕЖДУ ПРОЦЕССАМИ

Совокупность средств и механизмов ОС для обеспечения обмена данными между процессами называется межпроцессным взаимодействием, соответствующий английский термин - IPC (Interprocess Communications).

Так как адресное пространство каждого процесса обособлено (недоступно для других процессов), возникает проблема обмена информацией и синхронизации. В целом можно сказать, что приложения, использующие IPC, поддерживают клиент-серверную технологию. Клиент – это приложение, которое запрашивает сервис от другого приложения. Сервер – это приложение, обслуживающее клиентский запрос. Многие приложения являются одновременно и клиентом, и сервером.

ОС семейства WindowsNT предоставляет следующие механизмы для обеспечения коммуникаций и совместного использования данных разными приложениями - рис. 4.1.

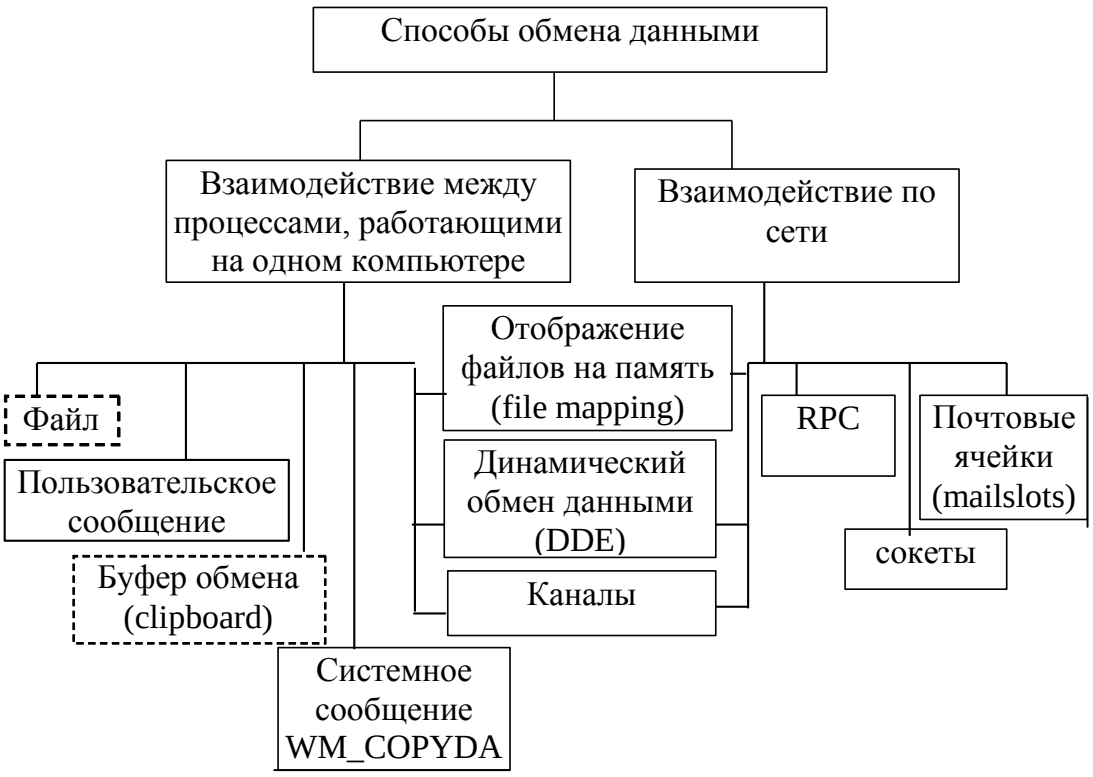

Рис. 4.1. Механизмы обмена данными между процессами

Одни формы IPC поддерживают разделение труда между процессами на одном компьютере, другие ориентированы на поддержку обмена данными между процессами, работающими на разных компьютерах (что оказывается гораздо сложнее, если аппаратные платформы разные и среды разные).

4.1. ОБМЕН ДАННЫМИ ПОСРЕДСТВОМ ФАЙЛА

Самой древней и примитивной формой обмена данными является обмен через файл. Это простое, но чрезвычайно медленное средство обмена. Кроме того, всегда существует возможность “пересечения” пишущего и читающего процессов, поэтому актуальна синхронизация.

4.2. БУФЕР ОБМЕНА

Буфер обмена (clipboard) предназначен в основном для обслуживания пользователя посредством “copy/paste”. Это довольно дорогое, неэффективное и устаревшее средство, но привычное и любимое пользователем. Поэтому продолжает существовать, несмотря на более эффективные способы обмена данными. Приложение должно поддерживать привычные для пользователя акселераторы Ctrl/C и Ctrl/V, для чего в цикле обработки сообщений должна присутствовать системная функция TranslateAccelerator(), а обработчик сообщения WM_COMMAND должен содержать обработку этих нажатий. Формат передаваемых таким образом данных выбирается из предопределенных или свои пользовательские форматы необходимо предварительно зарегистрировать.

Последовательность действий программиста для использования буфера обмена:

- функция OpenClipboard() – получить доступ к буферу обмена;
- функция EmptyClipboard() – очистить буфер обмена;

• функции SetClipboardData()/SetClipboardData() – загрузить в буфер или получить из буфера данные;

• функция CloseClipboard() – закрыть буфер.

4.3. ПОЛЬЗОВАТЕЛЬСКОЕ СООБЩЕНИЕ

Простейшим и очень эффективным способом обмена данными является посылка пользовательского сообщения. Любое сообщение можно отправить двумя способами:

• синхронно посредством функции SendMessage**(**HWND h,UINT msg, WPARAM wParam, LPARAM lParam), то есть управление к процессу-отправителю не вернется, пока сообщение не будет обработано процессом-получателем;

• асинхронно посредством функции PostMessage**(**HWND h,UINT msg, WPARAM wParam, LPARAM lParam), то есть функция поставит сообщение в очередь сообщений процесса-получателя и сразу же вернет управление отправителю, а обработано сообщение будет только тогда, когда оно будет извлечено из очереди сообщений получателем.

Это средство ограничивает объем передаваемых данных только двумя 32-разрядными значениями, которые можно передать посредством двух параметров функции – wParam, lParam. Но иногда этого бывает вполне достаточно.

Способы формирования получателя сообщения:

• сообщение может быть отправлено конкретному окну, указав его дескриптор в качестве первого параметра (дескриптор можно получить, например, с помощью функции FindWindow());

• можно отправить сообщение всем окнам, в том числе неактивным и невидимым, – широковещательно, тогда в качестве первого параметра указывается константа HWND_BROADCAST. При этом если оконная процедура обрабатывает данное сообщение, то выполнится соответствующий обработчик, если нет, то сообщение

будет проигнорировано.

Для посылки пользовательского сообщения нужно сформировать его идентификатор (номер), который указывается в качестве второго параметра. Номер нельзя назначать произвольно, так как есть диапазоны значений, зарезервированные ОС для определенных целей – табл. 4.1.

Таблица 4.1

Диапазоны идентификаторов сообщений

Диапазон	Предназначен для:
0 – (WM_USER–1)	Системные сообщения
WM_USER(0x4000) - WM_APP (0x7FFF)	Некоторые глобальные системные классы окон Windows, такие как BUTTON, EDIT, LISTBOX могут использовать для значений своих сообщений этот диапазон
WM_APP(0x8000) - 0xBFFF	В этом диапазоне пользовательские сообщения никаким образом не конфликтуют с системными, но рекомендуются для обмена сообщениями между окнами одного и того же процесса
0xC000 - 0xFFFF	Из этого диапазона ОС «выдает» уникальный идентификатор сообщения по запросу приложения во время выполнения
больше чем 0xFFFF	Зарезервирован для светлого будущего

Проблемы, возникающие при обмене сообщениями между разными приложениями:

- в разных приложениях сообщение, передаваемое от одного процесса другому, должно иметь одно и то же значение, поэтому разработчики должны каким-то образом договариваться о значении;
- нет гарантии, что какое-то третье приложение использует для своего пользовательского сообщения то же значение;
- если данные, которые нужно передать, не помещаются в

wParam, lParam, то такой способ вообще не годится;

• передавать посредством этих wParam, lParam адреса от одного процесса другому запрещается, так как адреса отправителя не будут иметь никакого смысла в адресном пространстве получателя.

Системная функция RegisterWindowMessage() позволяет определить пользовательское сообщение, номер которого будет гарантированно уникален в системе (на весь сеанс работы ОС). Возвращаемое сервисом значение можно использовать в качестве второго параметра (msg) в функциях SendMessage() или PostMessage(). Прототип функции:

```
UINT WINAPI RegisterWindowMessage( //возвращается уникальный
                    идентификаттор
  LPCTSTR lpString //строка, предназначенная для регистрации
                    сообщения
);
```

Специфика:

• функция RegisterWindowMessage() обычно используется для получения одинакового уникального значения для сообщения при взаимодействии двух или более приложений;

• если два приложения используют для регистрации одну и ту же строку, функция RegisterWindowMessage() возвращает одно и то же значение;

• сообщение остается зарегистрированным на все время текущей сессии ОС.

Пример передачи сообщения другому приложению:

```
{
    HWND h = FindWindow( 0, "<заголовок главного окна>" );
    if(h != 0)
    {
        UINT MyMessage=::RegisterWindowMessage("MY_MESSAGE");
        if(MyMessage==0)
        {
            MessageBox(0,"Не могу зарегистрировать Ваше
                    сообщение", “Информация”, MB_OK);
            return;
```

```
        }
        SendMessage(h,MyMessage,<32-разрядное значение>,<32-
                разрядное значение> ); //или HWND_BROADCAST
    } else MessageBox(0,"Не могу найти Ваше окно!»,
                ”Информация”,MB_OK);
}
```

Пример получения сообщения:

```
//Оконная процедура окна получателя сообщения
BOOL CALLBACK Proc(HWND hwndDlg, UINT message, WPARAM wParam, LPARAM
                lParam)
{
    static UINT  wm_My = RegisterWindowMessage("MY_MESSAGE"); //регистрируем
                сообщение, указывая в качестве параметра такую же строку,
                как и в отправителе
    if(message== wm_My) //если полученное сообщение совпадает с
                зарегистрированным
    {
        //обработка сообщения
        MessageBox(hwndDlg,"Got", "My Message Test", MB_OK);
    }else
        switch(message)
        {
        //обработка всех остальных сообщений
        }
}
```

4.4. СИСТЕМНОЕ СООБЩЕНИЕ WM_COPYDATA

В качестве передаваемых данных от одного приложения другому не могут использоваться адреса, так как каждое приложение выполняется в собственном адресном пространстве и адреса одного приложения не доступны и не имеют смысла для другого приложения. Поэтому в функциях SendMessage() и PostMessage() в общем случае адреса указывать нельзя. Исключения составляет сообщение WM_SETTEXT, например:

```
SendMessage(FindWindow(0,”MyApp”), WM_SETTEXT,0,(LPARAM)”New
                Caption”);
```

Обратите внимание: в LPARAM передается адрес строки с

новым заголовком, расположенной в адресном пространстве процесса-отправителя. Система отслеживает сообщение WM_SETTEXT, делая для него исключение: функция SendMessage() проверяет, не пытается ли отправитель послать WM_SETTEXT, и если да, то копирует строку из адресного пространства отправителя в блок памяти, доступный получателю. А процесс-приемник в свою очередь копирует буфер в свое адресное пространство.

Наряду с сообщением WM_SETTEXT существует второе исключение - сообщение WM_COPYDATA. Механизм WM_COPYDATA используется для передачи между процессами небольших порций простых данных. Передача данных посредством сообщения WM_COPYDATA является достаточно эффективным решением, это очень простое средство IPC, и иногда оказывается, что его достаточно, и совершенно незачем привлекать более изощренные механизмы.

Рассмотрим последовательность действий программиста для реализации этого механизма.

1. Сначала нужно создать переменную типа COPYDATASTRUCT и проинициализировать ее поля. Поля структуры:

```
struct COPYDATASTRUCT{
    ULONG_PTR dwData; //32-битовое число для передачи данных
                      принимающему процессу (любое значение).
                      Например, если Вы используете WM_COPYDATA в
                      нескольких целях, здесь может быть
                      значение, которое является признаком – что
                      именно Вы передаете. Полезно задавать это
                      значение, так как WM_COPYDATA это системное
                      сообщение, поэтому многие приложения могут
                      его использовать для разных целей
    DWORD cbData; //количество передаваемых байтов
    PVOID lpData; //указатель на данные. Может быть NULL.
                      Замечание: это опять адрес в адресном
                      пространстве процесса-отправителя!- каким
                      бы образом Вы этот блок не создавали – в
                      стеке, глобально, динамически!
};
```

2. Вызвать системную функцию SendMessage() со следующими параметрами:

```
SendMessage(
  <дескриптор целевого окна-получателя>,
  WM_COPYDATA,
  (WPARAM) <дескриптор окна-отправителя>,
  (LPARAM) <указатель на структуру COPYDATASTRUCT>
);
```

Возвращаемое функцией SendMessage() значение имеет следующий смысл: если приложение-получатель обработало данное сообщение, функция SendMessage() возвращает TRUE, в противном случае – FALSE.

Когда отправитель посылает сообщение WM_COPYDATA, функция SendMessage() резервирует блок в страничной памяти, размером не менее cbData байтов и копирует данные из адресного пространства процесса-отправителя в выделенный блок и отображает этот блок памяти на адресное пространство процесса-получателя, а только потом посылает сообщение окну-получателю. Когда оконная процедура окна-получателя обрабатывает это сообщение, lParam указывает на структуру COPYDATASTRUCT, расположенную в адресном пространстве процесса-получателя. А член этой структуры – lpData является указателем уже на свою копию данных, которая тоже находится в адресном пространстве процесса-получателя – рис. 4.2.

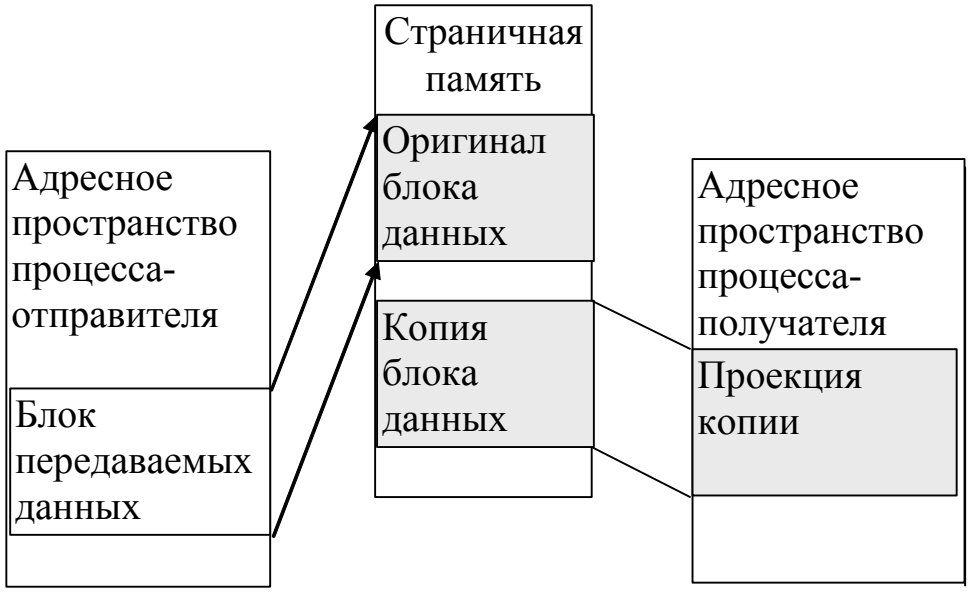

Рис. 4.2. Механизм передачи данных посредством WM_COPYDATA

Специфика:

- для пересылки WM_COPYDATA следует использовать функцию SendMessage(), а не PostMessage(). Последнее просто невозможно, так как, как только принимающая оконная процедура обработает сообщение, система должна освободить временный блок для копии данных в страничной памяти;
- хотя в качестве параметра передается адрес сформированной структуры, а в ней в свою очередь фигурирует указатель на блок данных, это не значит, что два процесса имеют доступ к одному и тому же физическому участку (блоку) памяти. «Пересылается» копия данных, поэтому:
  - следствие 1: что бы процесс-получатель ни делал со своей копией, процесс-отправитель об этом никогда не узнает, поэтому эти данные являются информацией только для чтения;
  - следствие 2: после отправки сообщения, что бы процесс-отправитель ни делал с оригиналом данных (возможно, с помощью своего другого потока), процесс-получатель тоже об этом не узнает, а будет работать со своей копией;
  - следствие 3: принимаемый процессом-получателем блок данных является достоверным (valid) только на время обработки сообщения WM_COPYDATA, а если процесс-получатель собирается использовать эти данные в дальнейшем, необходимо их скопировать в локальный буфер;
  - следствие 4: в передаваемых данных не должны фигурировать указатели, так как адреса данных в виртуальных адресных пространствах двух процессов вряд ли будут совпадать;
- процесс-получатель не должен освобождать память, на которую указывает lParam (не он ее занимал, не его дело освобождать).

Пример: один процесс посылает другому текстовую информацию.

Процесс-отправитель:

```
{
    //Посылаем дочернему процессу текстовую информацию (строку)
    HWND hChild = FindWindow( 0, <строка с заголовком окна-
                  получателя> );
    if(h !=0) //окно найдено
    {
        char str[]="Это проба WM_COPYDATA";
        COPYDATASTRUCT data;
        data.cbData = sizeof(str);
        data.dwData = 33; //признак – какого рода данные
                  передаются
        data.lpData = str;
        SendMessage(hChild, WM_COPYDATA,
                  (WPARAM)hParent,(LPARAM) &data);
    }else
    MessageBox(0,"","Не могу найти окно Slave!",MB_OK);
}
```

Оконная процедура процесса-получателя:

```
BOOL CALLBACK WinProc(HWND hwnd, UINT msg, WPARAM wParam,
                  LPARAM lParam)
{
   switch (msg)
    {
    case WM_COPYDATA:
        {
            COPYDATASTRUCT* pCopyData =
                  (PCOPYDATASTRUCT)lParam;
            if(pCopyDataStruct-> dwData==33) //сообщение
                  предназначается данному процессу
            char* pStr = (char*)pCopyData->lpData;//полученный
                  текст
            MessageBox(hwnd, pStr, "Got!", MB_OK);
        }
        …
    }
}
```

5. МЕХАНИЗМ ОТОБРАЖЕНИЯ ФАЙЛОВ НА ПАМЯТЬ (File mapping)

Если бы этого механизма не было, получение данных из файла выглядело бы примерно так:

- считать содержимое файла в виртуальное адресное пространство процесса;
- работа с содержимым;
- возможно, запись модифицированных данных обратно в файл.

Такой подход может быть использован, но имеет следующие недостатки:

- если файл большой (больше 2Гб), ОС не позволит передать приложению такой объем памяти, поэтому программист должен обеспечить считывание по частям;
- модифицированные данные нужно явно записывать обратно в файл;
- если к данному файлу одновременно обращаются несколько приложений, не гарантируется синхронизация доступа.

Отображение файлов на память (file mapping) – это ассоциация содержимого файла с порцией виртуального адресного пространства процесса. ОС создает объект исполняющей системы «file mapping» для поддержки этой ассоциации.

Отображение файлов на память используется в следующих случаях:

- ОС сама использует этот механизм для загрузки исполняемых (.exe и .dll) файлов. Это экономит как память, так и время, необходимое для того, чтобы приложение начало выполняться;
- это удобное и эффективное средство одновременной работы одного или нескольких приложений с одним файлом, или с частью файла, или с разными частями одного и того же файла;
- при использовании этого механизма упрощаются операции ввода/вывода, нет необходимости считывать содержимое в буфер

приложения, если файл слишком велик – его можно обрабатывать по частям;

○ практически единственное средство непоследовательной работы с файлами, так как приложение может адресоваться к данным точно также как к элементам массива;

○ всю работу по кэшированию файла выполняет сама ОС, поэтому не нужно явно выделять память, загружать данные из файла в память, переписывать измененные данные обратно на диск, освобождать выделенную память.

5.1. РЕАЛИЗАЦИЯ ПРОГРАММИСТОМ МЕХАНИЗМА ОТОБРАЖЕНИЯ ФАЙЛОВ НА ПАМЯТЬ

Последовательность действий программиста для использования механизма «file mapping» представлена на рис. 5.1.

Рассмотрим этапы подробно.

Вызов функции CreateFile()

Чтобы получить доступ к содержимому файла, требуется создать или открыть существующий объект исполняющей системы «файл». Функция CreateFile():

○ определяет текущий размер файла;

○ расположение файла (на жестком диске, на CD, на удаленном компьютере в сети;

○ способ доступа к файлу (создать, открыть существующий, получить информацию);

○ правила использования содержимого файла, как для процесса - создателя объекта, так и для других процессов.

Причем функция CreateFile(), хоть и трудно предположить об этом по названию, предназначена как для создания нового объекта, так и для доступа к существующему. Хотя для достижения второй цели можно

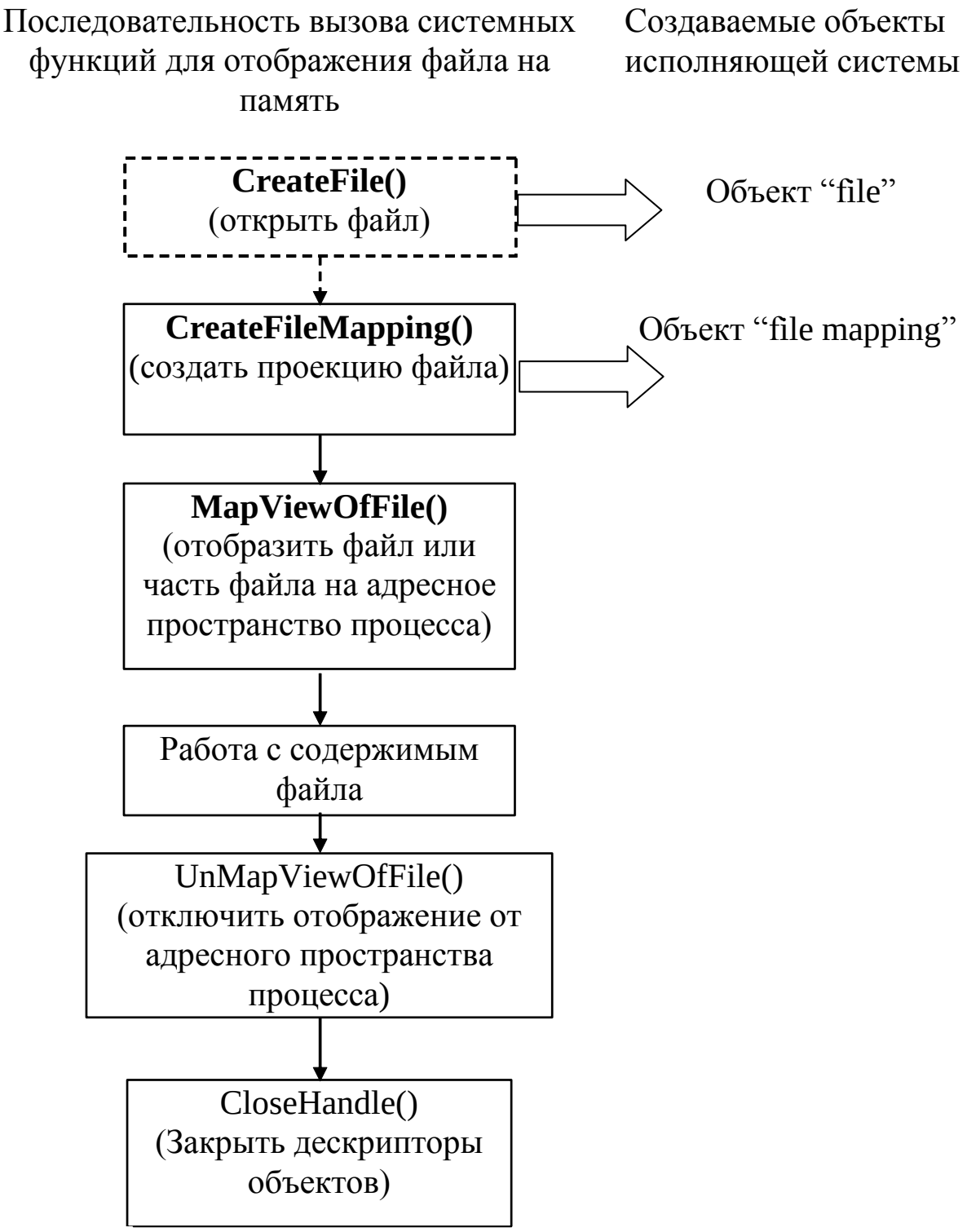

Рис. 5.1. Последовательность действий для реализации отображения файла на память

использовать функцию OpenFile(), рекомендуется использовать для

обеих целей функцию CreateFile(). Кроме того, этот же сервис используется для создания объектов «Mailslot», «Pipe», для доступа к директориям и многого другого.

Рассмотрим прототип функции и возможные значения параметров.

```
HANDLE CreateFile(
    LPCTSTR lpFileName, //указатель на строку с именем файла
                    (при необходимости с путем)
    DWORD   dwAccess,   //нужный Вам режим доступа:
                    файл будет открыт только для чтения –
                    GENERIC_READ,
                    только запись - GENERIC_WRITE,
                    а если и запись, и чтение - GENERIC_READ |
                    GENERIC_WRITE.
                    0-содержимое файла нельзя считывать или
                    записывать. Применяется, если Вы хотите
                    всего лишь получить атрибуты файла –
    DWORD   dwShareMode,    //режим совместного использования
                    файла (как могут другие этот файл
                    использовать):
                    0 – совместный доступ запрещен, то есть
                    файл не подлежит открытию «со стороны»
                    (любой другой CreateFile() для этого файла
                    завершится INVALID_HANDLE_VALUE), более
                    того – даже этот же процесс не сможет
                    «открыть» данный файл еще раз
                    FILE_SHARE_READ – попытка другим процессом
                    открыть файл с GENERIC_WRITE не удастся
                    (защита на время чтения), то есть другие
                    процессы могут открывать данный файл только
                    для чтения
                    FILE_SHARE_WRITE- попытка другим процессом
                    открыть файл с GENERIC_READ не удастся
                    (защита на время записи), но разрешается
                    параллельная запись в файл
                    FILE_SHARE_READ | FILE_SHARE_WRITE – другой
                    процесс может открывать файл без
                    ограничений.

    LPSECURITY_ATTRIBUTES lpSecurityAttributes,  //защита и
                    признак – может ли объект наследоваться
                    дочерним процессом
    DWORD   dwCreation, //параметры создания – определяет: надо
                    ли создавать новый файл, заменять
                    существующий… Следующие флаги можно
```

```
                объединять поразрядным или «|»
                CREATE_NEW – завершится неудачей, если
                указанный файл уже существует, иначе
                создается новый файл
                CREATE_ALWAYS – существующий файл
                заменяется новым
                OPEN_EXISTING – неудача, если файл не
                существует
                OPEN_ALWAYS – файл открывается и создается,
                если не существует
                TRUNCATE_EXISTING – устанавливается нулевая
                длина файла, при этом в dwAccess должен
                быть, по крайней мере, GENERIC_WRITE
DWORD   dwFlagsAndAttributes,  //два вида: атрибуты и флаги
                Обычно FILE_ATTRIBUTE_NORMAL
                Замечание: атрибуты - это характеристики
                самого файла, а не способа его открытия, =>
                имеют смысл только при создании нового
                файла => они игнорируются, если открывается
                существующий файл:
                FILE_ATTRIBUTE_NORMAL – используется только
                если не установлены никакие другие атрибуты
                (однако флаги могут быть установлены)
                FILE_ATTRIBUTE_HIDDEN
                FILE_ATTRIBUTE_READONLY – нельзя ни писать
                в файл, ни удалять его

                флаги:
                FILE_FLAG_DELETE_ON_CLOSE – полезен для
                временных файлов. Файл удаляется, когда
                закрывается его последний открытый
                дескриптор
                FILE_FLAG_OVERLAPPED – важен для
                асинхронного ввода/вывода
                …

HANDLE  hTemplateFile); //обычно 0, если нет - дескриптор
                файла, содержащего атрибуты для вновь
                создаваемого файла. При этом значения
                dwFlagsAndAttributes игнорируются
```

При успешном завершении функция возвращает дескриптор открытого файла, в противном случае – значение INVALID_HANDLE_VALUE (которое соответствует -1).

Замечания:

○ если атрибуты совместного доступа и безопасности разрешают, один и тот же объект файл может иметь несколько

открытых дескрипторов, которые могут относиться к одному и тому же процессу или к разным;

○ чтобы обезопасить себя от ситуации, когда другие процессы могут записывать данные в ту часть файла, которую использует процесс-создатель объекта, следует разрешить остальным только чтение (параметр dwShareMode должен иметь значение FILE_SHARE_READ) или вообще ничего не позволять (0).

При вызове функции CreateFile() никакого отображения не выполняется! Но, получив дескриптор объекта «файл» можно использовать его значение для чтения из файла посредством функции ReadFile() или для записи в него с помощью функции WriteFile(). Но это механизм исключительно последовательного доступа к содержимому файла.

Вызов функции CreateFileMapping()

Функция CreateFileMapping() определяет объем страничной памяти, необходимый для дальнейшего отображения, и атрибуты защиты страниц. Если вызов прошел успешно, то создается объект исполняющей системы «проекция файла» («file mapping»). Этот этап является подготовкой к проецированию.

Объект «file mapping» содержит следующую информацию:

○ максимальный объем требуемой страничной памяти;

○ общие атрибуты защиты страниц;

○ при проецировании для организации свопинга использовать файл подкачки или сам файл на диске;

○ отображать данные во всех процессах по одному и тому же виртуальному адресу (базированное отображение);

○ и т. д.

Прототип функции:

```
HANDLE CreateFileMapping( //в случае успеха возвращается
                          дескриптор объекта, в противном случае 0
    HANDLE  hFile,        //дескриптор, возвращенный CreateFile() или
                          константа INVALID_HANDLE_VALUE (-1). В
                          первом случае сервис в дальнейшем отобразит
                          заданный файл в память, во втором создаст
```

блок памяти для совместного использования разными процессами. Замечание: так как CreateFile()в случае неудачи возвращает INVALID_HANDLE_VALUE, программист должен обязательно проверять возвращаемое значение, иначе результат может быть неожиданным

LPSECURITY_ATTRIBUTES lpAttributes, //указатель на структуру SECURITY_ATTRIBUTES, в большинстве случаев используется NULL, что означает создание дескриптора защиты по умолчанию, и объект не будет наследоваться

DWORD flProtect, //два вида, которые можно комбинировать поразрядным «OR»:
1.как собираемся использовать - атрибуты защиты страниц. Должны быть согласованы с правами доступа в CreateFile(). Это будет использовано в дальнейшем при “связывании” содержимого файла с физической памятью
PAGE_READONLY – после отображения можно считывать данные из файла (но нельзя записывать или исполнять). При этом в CreateFile() – GENERIC_READ
PAGE_READWRITE – полный доступ, то есть после отображения можно и читать и писать. При этом в CreateFile() – GENERIC_READ | GENERIC_WRITE
PAGE_WRITECOPY - после отображения можно и читать, и писать. При этом при изменении содержимого отображенной памяти собственная (для данного процесса) копия записывается в файл подкачки, а не непосредственно в исходный файл (при записи будет создана закрытая копия страницы), а в CreateFile() – GENERIC_READ | GENERIC_WRITE

2. как использовать файл подкачки и формировать атрибуты защиты страниц:
SEC_IMAGE – предписывает ОС при отображении исполняемого файла задавать разные атрибуты защиты страниц для кода и данных,
SEC_COMMIT используется только при hFile, равном INVALID_HANDLE_VALUE, и предписывает ОС выделить требуемый объем в страничной памяти,

DWORD dwMaximumSizeHigh, // Сколько собираемся использовать памяти (если размер файла меньше, он «вырастает»). Основное

```
                    назначение сервиса – убедиться, что для
                    объекта «проекция файла» доступен нужный
                    объем страничной памяти. Это старшая часть
                    64-разрядного размера файла (для файлов <
                    4Гб значение равно 0)
    DWORD dwMaximumSizeLow, // младшая… Замечание: если оба
                    параметра равны 0, используется текущий
                    размер открытого файла, но размер файла
                    изменяться не будет! Если Вы не собираетесь
                    писать в файл или изменять его размер,
                    используйте два нуля. Если ожидается, что
                    размер файла может увеличиться, используйте
                    ожидаемый размер (тогда при необходимости
                    будет использован нужный размер файла).
    LPCTSTR lpName);    // имя объекта, посредством которого
                    отображение может быть доступно всем
                    работающим процессам, например
                    “MyFileMapping”. Это строка,
                    заканчивающаяся 0 и не содержащая символа
                    «\», регистр не влияет. Если отображение
                    будет использоваться только одним
                    процессом, то
```

Если данный процесс является первым, кто вызвал эту функцию, создается новый объект «file mapping» и его счетчик пользователей устанавливается в 1. Если процесс вызывает функцию CreateFileMapping() для существующего объекта, то возвращается процессо-зависимый дескриптор существующего объекта, и счетчик пользователей инкрементируется. Как получен доступ, можно узнать посредством функции GetLastError(): если создан новый объект, возвращаемое значение NO_ERROR, а если объект уже существовал - ERROR_ALREADY_EXISTS.

Замечания:

- если для отображения используется реальный файл, то параметры доступа в обеих функциях должны быть синхронизированы (например, если в функции CreateFileMapping() указан флаг PAGE_READONLY, то в функции CreateFile() необходимо задать GENERIC_READ);
- при задании размера отображения ОС проверяет наличие запрашиваемого объема в страничной памяти, а не в виртуальном

адресном пространстве процесса;

- если запрашиваемый размер отображения больше размера файла, то файл увеличивается (разница заполняется нулями), если размер отображения меньше размера файла, то можно использовать только часть файла;
- нельзя в область файла за указанной границей, так как объект отображения не может «расти»;
- так как невозможно создать отображение только для записи (если позволено записывать, то прочитать можно всегда), то соответствующего флага PAGE_WRITE не существует;
- если механизм используется для резервирования памяти, доступной нескольким процессам (при этом никакой файл не используется), то выделенные в страничной памяти страницы будут обнулены ОС;
- два процесса могут пользоваться одним и тем же отображением, используя одно и то же имя в качестве параметра функции CreateFileMapping() или OpenFileMapping(), передавать дескриптор объекта по наследству дочернему процессу, а также с помощью дублирования дескриптора.

Вызов функции MapViewOfFile()

Вызов функции CreateFileMapping() - это еще только подготовка к проецированию, то есть проверка и обеспечение возможности проецирования. Проецирование выполняется функцией MapViewOfFile(). Эта операция делает указанную порцию файла “видимой” в адресном пространстве процесса. Это означает, что ОС формирует в таблице страниц процесса нужное количество записей с требуемыми флагами. Каждая запись является средством, которое позволяет транслировать виртуальный адрес процесса в физический. За счёт этого механизма приложение получает так называемое представление файла («ViewOfFile»), можно считать, что это окно, через которое приложение «смотрит» на файл или часть файла.

Прототип функции:

```
LPVOID MapViewOfFile( //возвращаемое значение: в случае успеха
                      указатель, которым можно пользоваться как
                      для чтения данных, так и для записи (если
                      разрешено). Если по каким-то причинам
                      спроецировать данные не удалось, то
                      возвращается 0
HANDLE hFileMappingObject,  //дескриптор объекта «проекция
                      файла», полученный предыдущим
                      CreateFileMapping() или OpenFileMapping()
DWORD dwDesiredAccess,  //вид доступа к данным – как процесс
                      хочет обращаться к данным именно
                      посредством создаваемого отображения.
                      Должен соответствовать правам доступа
                      объекта FileMapping.
                      FILE_MAP_WRITE – можно и считывать, и
                      записывать (должны были вызвать
                      CreateFileMapping с PAGE_READWRITE)
                      FILE_MAP_READ – можно только считывать,
                      (можно было вызвать CreateFileMapping с
                      любым :PAGE_READWRITE, PAGE_READONLY,
                      PAGE_WRITECOPY)
                      FILE_MAP_ALL_ACCESS== то же, что
                      FILE_MAP_WRITE (и чтение, и запись)
                      FILE_MAP_COPY – если отображение создано с
                      PAGE_WRITECOPY, Вы получаете доступ к файлу
                      следующим образом: если Вы туда пишите, то
                      изменения не затрагивают оригинальный файл,
                      а касаются только Вашей копии содержимого
                      файла.
DWORD dwFileOffsetHigh, //с какого байта проецировать:
                      необязательно проецировать на адресное
                      пространство весь файл сразу – можно только
                      требуемую часть, которая в таком случае
                      называется представлением – view.=>
                      смещение первого байта view относительно
                      начала. Замечание: смещение в файле должно
                      быть четным числом и кратным гранулярности
                      выделения ресурсов в данной системе – в
                      Win32 – 64Кб. Для отображения от начала
                      файла - 0
DWORD dwFileOffsetLow,      // если оба параметра=0, то от
                      начала файл
SIZE_T dwNumberOfBytesToMap ); //количество байт (размер
                      представления не может превышать 4Гб). Если
                      параметр=0, система попытается
                      спроецировать представление с указанного
                      смещения и до конца файла.
```

Этот вызов увеличивает счетчики пользователей объектов «файл» и «проекция файла» на 1.

Замечания:

- одни и те же страницы могут отображаться на разные виртуальные адреса в разных процессах;
- для отображения можно использовать также функцию MapViewOfFileEx(), которая пытается «привязать» проецируемые данные к указанному программистом виртуальному адресу. Функция MapViewOfFileEx() в основном используется самой ОС для того, чтобы загрузить исполняемый файл по указанному адресу. Можно в двух разных процессах отобразить одни и те же данные (например, массив) по одному и тому же адресу, но при этом адрес должен соответствовать гранулярности выделения памяти;
- дескриптор файла, для которого создается отображение, должен оставаться открытым, пока не вызвана функция MapViewOfFile();
- указатель, возвращаемый функцией MapViewOfFile(), понадобится при освобождении памяти, поэтому модифицировать значение не следует;
- так как при выполнении этой функции ОС увеличивает счетчики объектов «файл» и «проекция файла» на 1, то после вызова можно закрыть дескрипторы этих объектов с помощью функции CloseHandle().

Вызов функции UnmapViewOfFile()

Когда необходимость в данных файла или совместного использования памяти отпадает, следует отключить зарезервированные страницы от адресного пространства процесса, то есть освободить «представление». Для этих целей используется функция UnmapViewOfFile().

Прототип функции:

```
BOOL UnmapViewOfFile(
  LPCVOID lpBaseAddress ); //тот базовый адрес, который был получен
                           в результате MapViewOfFile().
);
```

Замечания:

- если процесс не вызовет эту функцию, зарезервированный регион не будет освобожден до завершения процесса;
- этот вызов нужен также для синхронизации содержимого файла и зарезервированных для отображения страниц. ОС буферизует изменения в отображении и не обновляет немедленно дисковый образ, а просто помечает страницы как измененные. При вызове функции UnmapViewOfFile() система перекачивает все изменения на диск. В принципе для этих же целей можете использовать функцию FlushViewOfFile(), которая с точностью до гранулярности позволяет обновить любую часть представления («ViewOfFile»);
- специфика вызова функции UnmapViewOfFile() для файлов, отображенных с флагом FILE_MAP_COPY, заключается в том, что все изменения касаются только копии данных, поэтому в этом случае никакие изменения записаны в файл не будут, а функция UnmapViewOfFile() просто освободит регион памяти (все Ваши изменения пропадут). Для сохранения изменений можно создать второй объект «проекция файла» с PAGE_READWRITE и спроецировать с помощью FILE_MAP_WRITE, а потом перенести данные из одного представления («ViewOfFile») в другое с помощью функции MoveMemory();
- функция UnmapViewOfFile() декрементирует счетчики пользователей объектов «файл» и «проекция файла».

Закрытие дескрипторов объектов «проекция файла» и «файл»

Любой открытый дескриптор объекта исполняющей системы должен быть закрыт с помощью функции CloseHandle(), иначе в процессе может произойти утечка ресурсов. Конечно, при завершении процесса система автоматически закроет все дескрипторы объектов, принадлежащих процессу. Но, если процесс работает достаточно долго, и за собой «не чистит», накопится слишком много незакрытых дескрипторов и соответственно не освобожденных ресурсов. Поэтому хорошим тоном считается: «открыл – попользовался-закрыл».

Пример

Рассмотрим пример:

```
{
    //допустим, файла “My.dat” в текущем каталоге еще не
                    существует
    HANDLE hfile = CreateFile(“My.dat”,
                    GENERIC_READ | GENERIC_WRITE, //полный
                    доступ для своего процесса
                    FILE_SHARE_READ | FILE_SHARE_WRITE,
                    //полный доступ для других процессов
                    0,
                    CREATE_ALWAYS,
                    FILE_ATTRIBUTE_NORMAL,
                    0);
    //теперь “My.dat” существует, но имеет 0-ую длину
    if(hfile!= INVALID_HANDLE_VALUE)
    {
    HANDLE hfilemap = CreateFileMapping(hfile, //дескриптор
                    файла
                    0, //дескриптор защиты формируется по
                    умолчанию, объект не может быть унаследован
                    дочерним процессом
                    PAGE_READWRITE, //страницы предназначены
                    как для чтения, так и для записи
                    0,100, //размер отображения. Так как размер
                    отображения больше размера файла, ОС
                    увеличивает размер файла до указанного (то
                    есть до 100 байтов)
                    0); //объект неименованный
//Проецирование и использование
    char* p = static_cast<char*>(MapViewOfFile(hfilemap,
                    FILE_MAP_WRITE, 0,0,0));//проецируем весь
                    файл с начала
    if(p) //если удалось отобразить файл на адресное
                    пространство процесса
    {
    p[0] = ‘Q’; //
    strcpy(p,”QWERTY”);//теперь после завершения приложения в
                    файле будет строка «QWERTY»
    UnmapViewOfFile(p);
    }
    //Освобождение ресурсов
    CloseHandle(hfilemap);
}
    CloseHandle(hfile);
```

```
    //по завершении процесса файл будет существовать, и иметь
                    размер 100 байтов
}
```

Если требуется создавать объект, только если он еще не существует:

```
    HANDLE hfilemap = CreateFileMapping(hfile,0,
                    PAGE_READWRITE,0,100,"MyFileMapping");
    if (hfilemap != NULL && GetLastError() ==
                    ERROR_ALREADY_EXISTS)
    {
        CloseHandle(hfilemap);
        return;
    }
```

Просмотр больших файлов по частям

```
{
DWORD part = <значение>;//должно быть кратно гранулярности
                    страницы. Для получения гранулярности можно
                    использовать функцию GetSystemInfo ().
                    Обычно гранулярность страницы 64К
HANDLE hFile = CreateFile("Big.dat", GENERIC_READ,
                    FILE_SHARE_READ, 0, OPEN_EXISTING,
                    FILE_FLAG_SEQUENTAL_SCAN,0);//собираемся
                    просматривать файл последовательно от
                    начала до конца. ОС использует флаг
                    FILE_FLAG_SEQUENTAL_SCAN как возможность
                    оптимизации кэширования (что невозможно,
                    если мы устанавливаем file pointer
                    произвольно). За счет такого просмотра
                    можно существенно повысить
                    производительность
if(hFile!= INVALID_HANDLE_VALUE)
{
HANDLE hFileMap = CreateFileMapping (hFile, 0,
                    PAGE_READONLY,0,0,0);//читать весь файл с
                    самого начала
if(hFileMap)
{
//Вычисляем 64-разрядный размер файла
DWORD high;//для старшей части
DWORD low = GetFileSize(hFile,&high);
```

```
long long total = low + (static_cast<long long>(high))<<32;
long long offset = 0;
while(total>0)
{
DWORD n=(total<part)?total:part; //размер порции отображения
char* p = static_cast<char*>(MapViewOfFile(hFileMap,
                    FILE_MAP_READ,
                     offset>>32,//старшая часть смещения
                    offset&0xffffffff,//младшая часть смещения
                    n);//сколько байтов требуется отобразить
if(p)
{
    …//работаем с частью отображения
}
UnmapViewOfFile(p);
offset += part;
total -= part;
}
CloseHandle(hFileMap); //
}
CloseHandle(hFile);//
}
```

Отображение по указанному адресу

```
{
    int ar[BIG]; //это в моем адресном пространстве

    __try{
        HANDLE hFile = CreateFile(_T("FileToMap.txt"),
                    GENERIC_READ|GENERIC_WRITE, FILE_SHARE_READ
                    | FILE_SHARE_WRITE, NULL, CREATE_ALWAYS, 0,
                    0);
        if(hFile == INVALID_HANDLE_VALUE) __leave;
        HANDLE hFileMapping = CreateFileMapping(hFile, NULL,
                    PAGE_READWRITE, 0, 100, NULL);
        if(hFileMapping == NULL) __leave;
        // Приведем к 64к границе...
SYSTEM_INFO si;
GetSystemInfo(&si);
DWORD d= si.dwAllocationGranularity;
```

```
      int* pMap = static_cast<int*>
                  (((reinterpret_cast<DWORD>(ar)+d)/d) * d);
      pMap = MapViewOfFileEx(hFileMapping, FILE_MAP_READ |
                  FILE_MAP_WRITE, 0, 0, 0, pMap);
           if(pMap == NULL) __leave;
        //работаем с pMap
    }
    __finally{
        if(hFile != INVALID_HANDLE_VALUE) CloseHandle(hFile);
        if(hFileMapping)
                  CloseHandle(hFileMapping);
        if(pMap)                    UnmapViewOfFile(pMap);
    }
}
```

Когерентность проекций

Когерентность всех проекций, соответствующих одному объекту «file mapping» поддерживается на уровне ОС. Когерентность гарантирована за счет того, что система использует одни и те же страницы в страничном файле, которые отображает на разные виртуальные адресные пространства. Поэтому все процессы, совместно использующие «file-mapping», имеют согласованные образы данных. То есть, если один процесс внесет изменения в область своей отображенной памяти, другой эти изменения «увидит» в своей отображенной памяти (потому что разные виртуальные адреса обоих процессов ссылаются на одни и те же физические адреса). Например, можно создать несколько отображений (несколько «окон») в разных процессах или даже в одном и том же процессе, посредством которых можно работать с разными частями одного файла или с одной и той же частью рис. 5.2.

Замечания:

- отображение файла на память это также способ (практически единственный легальный) для организации памяти общего доступа для разных процессов. Если требуется просто обеспечить нескольким процессам доступ к одному общему участку памяти, можно создать объект «file-mapping», не ставя ему в соответствие никакого реально существующего файла на диске;
- если отображения используют разные объекты «file-mapping»,

когерентность не гарантируется, так как при этом для каждого объекта выделяются свои страницы в страничной памяти;

• даже если несколько процессов используют для отображения один объект «file mapping», изменения в страничной памяти мгновенно в файл на диске не переписываются. Изменения записываются при вызове функций UnmapViewOfFile() и FlushViewOfFile();

• если указан файл, расположенный на другом компьютере в сети, то только процессы на одном компьютере будут когерентны, то есть будут использовать одну и ту же область страничного файла;

• не лучшей идеей является доступ к отображенному файлу с помощью функций ReadFile() и WriteFile(), так как в этом случае не гарантируется согласованность.

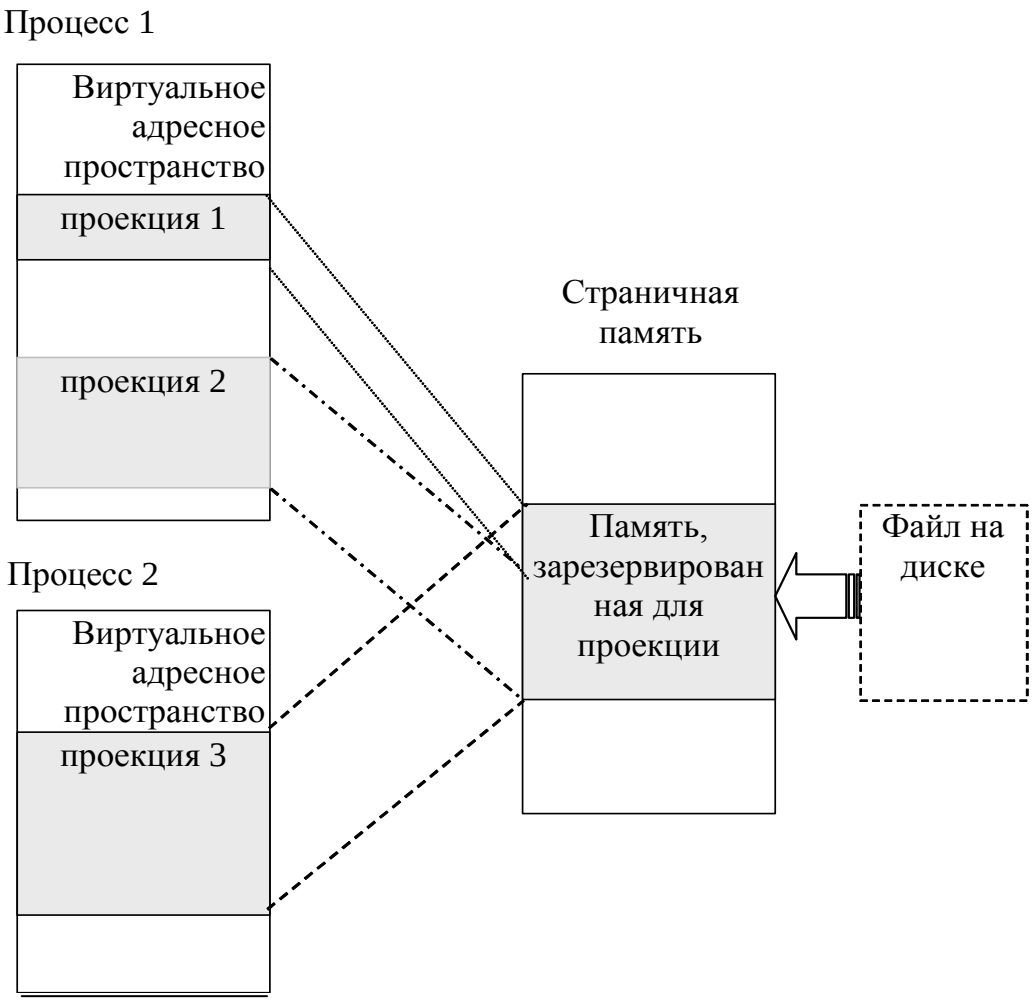

Рис. 5.2. Когерентность отображений

Итог

Преимущества отображения файлов на память:

- это практически единственный легальный механизм, позволяющий совместно использовать память нескольким процессам;
- быстрый и простой способ манипулирования содержимым файла:
  - скорость доступа увеличивается за счет того, что отображение данных находятся в страничной памяти, а сам файл остается на диске;
  - появляется возможность не только последовательно считывать/записывать данные, но и иметь произвольный доступ к ним;
  - при необходимости работы процесса с большим объемом данных (например, файлами баз данных) можно отображать на память не целиком файл, а последовательно отображать содержимое на память «порциями»;
  - обращение к данным становится проще за счет привычного в языке С указателя.

Недостаток этого механизма один: механизм отображения файлов очень удобно использовать для чтения, однако если вы хотите дописать данные в файл (увеличить), то при отображении файла на память Вы должны указать результирующий размер. Отображение файла не может быть увеличено, поэтому при создании отображения файла требуется знать максимальный размер, который определить бывает трудно или невозможно.

6. РАЗДЕЛЕНИЕ ДАННЫХ ИСПОЛНЯЕМЫМИ ФАЙЛАМИ

В среде Win32 каждый процесс выполняется в обособленном защищенном адресном пространстве. Если запущено несколько экземпляров приложения, то каждому процессу ОС предоставляет отдельную копию глобальных переменных. Это справедливо и для глобальных данных динамических библиотек. Это означает, что в Win32 глобальные переменные не могут использоваться процессами совместно, так как это было в Win16.

И все же, выполнив ряд замысловатых манипуляций над сегментом данных приложения или динамической библиотеки, можно создать общую область памяти (одни и те же страницы в страничной памяти) для всех процессов.

Чтобы обеспечить совместное использование памяти несколькими экземплярами приложения или библиотеки, нужно предоставить необходимую информацию компилятору и компоновщику.

Компилятору нужно предписать создать в объектном модуле раздел с указанным программистом именем и поместить в этот раздел блок переменных.

```
#pragma data_seg (“MyShared”) //задается имя раздела в
                    исполняемом файле
volatile int count=0; //инициализация обязательна, иначе
                    переменная будет помещена не в раздел с
                    именем MyShared, а в специальный раздел
                    неинициализированных переменных
//другие переменные
#pragma data_seg () //конец блока переменных, которые должны
                    быть помещены в указанный раздел
```

Замечание: второй способ специфичен для Microsoft:

```
    #pragma data_seg (“MyShared”)//все равно требуется, так как
                    компилятор должен создать соответствующий
                    раздел
    __declspec(allocate(“MyShared”))volatile DWORD dll_count;
                    //инициализация не обязательна
```

Этого недостаточно. Компоновщику тоже нужно сообщить, что переменные в этом разделе должны быть общими и как с ними можно обращаться:

- ключ компоновщику в командной строке (MyShared – имя раздела, RWS – атрибуты «читать», «писать», «разделять»): /SECTION:MyShared,RWS;
- в исходном тексте программы посредством директивы #pragma comment(linker, “/SECTION:MyShared,RWS”);
- в файле определений (с расширением .def), указав секцию «SECTIONS», а в ней .MyShared READ WRITE SHARE.

7. СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ ОБЪЕКТОВ

Сами объекты, описывающие ресурсы, существуют в единственном экземпляре и не копируются из процесса в процесс. Чтобы несколько процессов могли совместно использовать любой ресурс, каждый процесс должен получить собственный процессо-зависимый дескриптор одного и того же объекта. Существует несколько способов передачи дескрипторов другому процессу.

7.1. НАСЛЕДОВАНИЕ ОПИСАТЕЛЕЙ ОБЪЕКТОВ

Наследование дескрипторов применимо только тогда, когда процессы связаны «родственными» отношениями. Например, родительскому процессу доступны один или несколько описателей объектов исполняющей системы, и он решает, породив дочерний процесс, передать ему по наследству доступ к своим объектам. Чтобы обеспечить такую передачу дескрипторов дочернему процессу, требуется выполнить следующие действия.

Создать наследуемый дескриптор

При создании объекта указать системе, что порождающему процессу нужен наследуемый описатель. Чтобы создать наследуемый описатель, родительский процесс создает и инициализирует экземпляр структуры SECURITY_ATTRIBUTES, а затем передает его адрес соответствующей Create-функции.

Например:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = 0;    //защита по умолчанию
sa.bInheritHandle = TRUE;   //описатель должен быть наследуемым

HANDLE hMutex = CreateMutex(&sa, FALSE, 0);
```

Порождение дочернего процесса

Передать дескриптор по наследству можно только дочернему

процессу. Для передачи дескрипторов в функции CreateProcess() следует в параметре bInheritHandles передать значение TRUE.

```
CreateProcess(0,<параметры командной строки>,0,0,
                    TRUE, //указание ОС передать дочернему
                    процессу все наследуемые дескрипторы
                    0,0,0,<указатель на STARTUPINFO>,<указатель
                    на PROCESS_INFORMATION>);
```

Тогда ОС, формируя таблицу описателей дочернего процесса, копирует из таблицы родительского все наследуемые описатели, причем в те же позиции. Последний факт чрезвычайно важен, так как это означает, что описатели будут идентичны в обоих процессах. Помимо копирования наследуемых описателей, система увеличивает значения счетчиков всех объектов, так как объекты теперь будут использоваться дочерним процессом также.

Замечания:

- наследуются только описатели наследуемых объектов, существующие в родительском процессе на момент создания дочернего процесса. Если родитель после вызова создаст еще какие-нибудь объекты, то они уже будут недоступны дочернему процессу.
- при наследовании описателей объектов дочерний процесс не имеет ни малейшего представления о тех описателях, которые он унаследовал.

Как дочерний процесс может узнать об унаследованных дескрипторах

Родительский процесс может передать значения своих наследуемых дескрипторов (так как только в этом случае они будут одинаковыми в обоих процессах) дочернему процессу:

- в качестве аргумента командной строки;
- с помощью простейших средств межпроцессного взаимодействия (например, отправить дочернему процессу сообщение);

Замечание: в некоторых случаях дочернему процессу как раз необязательно знать об унаследованных объектах, например в случае перенаправления стандартного ввода/вывода.

7.2. ИМЕНОВАНИЕ ОБЪЕКТОВ

Именование объектов - это способ, позволяющий нескольким процессам (совсем необязательно связанным родством) совместно использовать объекты исполняющей системы. Именование допускают многие объекты исполняющей системы. Во всех таких функциях Create* в качестве параметра фигурирует параметр, который содержит указатель на строчку с именем объекта – LPCTSTR lpszName. Если этот параметр равен нулю, создается безымянный объект исполняющей системы. В этом случае можно разделять его между процессами либо с помощью наследования, либо с помощью дублирования описателя.

Если Вы создаете именованный объект, параметр lpszName должен содержать адрес строки, заканчивающейся 0, не содержащей «\» и длиной не более константы MAX_PATH.

Замечание: Вы никогда не застрахованы от ситуации, когда в системе уже существует объект с таким именем. Еще хуже, что такие объекты как мьютексы, семафоры, события, проекции файлов и ждущие таймеры находятся в едином пространстве имен, поэтому в ситуации:

```
HANDLE hMutex = CreateMutex(0,FALSE,”MyObject”);
HANDLE hSema = CreateSemaphore(0,1,1,”MyObject”);//всегда
                    будет возвращать неудачу, то есть 0
```

Пример использования именования.

1. Процесс А создает ненаследуемый именованный объект «mutex»:

```
HANDLE hMutex = CreateMutex(0,FALSE,”MyMutex”);   //система
                    создает новый объект со счетчиком, равным
                    1, а в таблице описателей процесса А
                    соответствующую запись
```

2. Процесс Б тоже создает объект «mutex» с тем же именем:

```
HANDLE hMutex = CreateMutex(0,FALSE,”MyMutex”);
```

Система сначала проверяет, не существует ли уже объект исполняющей системы с таким именем. Если существует, проверяет

его тип. Так как мы пытаемся создать объект того же типа и с тем же именем, система считает вызов правомерным и создает в таблице описателей процесса Б соответствующую запись, инициализируя ее так, чтобы она указывала на уже существующий объект “mutex”. Хотя процесс Б вызвал функцию CreateMutex(), новый объект не создается. Вместо этого формируется процессо-зависимый дескриптор, который идентифицирует уже существующий объект, а счетчик объекта увеличивается на 1.

Замечание: скорее всего значения дескрипторов одного и того же объекта в разных процессах будут разными. Каждый процесс будет использовать свой описатель для работы с одним и тем же объектом ядра.

Замечание: всегда можно определить: создан новый объект или открыт существующий с помощью:

```
HANDLE hMutex = CreateMutex(0,FALSE,”MyMutex”);
if(GetLastError() == ERROR_ALREADY_EXISTS) – открыт уже
                    существующий объект
```

Есть другой способ разделения объектов по именам: вместо функции CreateMutex()- вызовите функцию OpenMutex(). В функциях Open* нельзя передать вместо указателя на строку с именем - 0.

Главное отличие функций Create* и Open*: если объект еще не существует, функция Create* его создает, а функция Open* просто сообщает, что такого объекта не существует.

7.3. ДУБЛИРОВАНИЕ ОПИСАТЕЛЕЙ

Этот механизм реализуется с помощью функции:

```
BOOL DuplicateHandle(
  HANDLE hSourceProcessHandle,  // дескриптор процесса
                    источника
  HANDLE hSourceHandle, // дескриптор объекта ядра любого типа
                    процесса источника
  HANDLE hTargetProcessHandle,  // дескриптор процесса
                    приемника
```

```
  LPHANDLE lpTargetHandle,  // указатель на переменную типа
                    HANDLE, куда будет помещен процессо-
                    зависимый дескриптор для процесса приемника
  DWORD dwDesiredAccess,        // доступ к новому описателю.
                    Если dwOptions== DUPLICATE_SAME_ACCESS,
                    этот параметр игнорируется
  BOOL bInheritHandle,          // говорит о том, будет ли
                    дескриптор наследуемым - TRUE (если
                    процессу приемнику вздумается порождать
                    дочерний процесс)
  DWORD dwOptions // 0 или комбинация следующих значений:
                    DUPLICATE_CLOSE_SOURCE (приводит к закрытию
                    описателя источника) и
                    DUPLICATE_SAME_ACCESS (у описателя
                    приемника тот же доступ, что и у описателя
                    источника)
);
```

Замечание: как и при наследовании дескрипторов дочерним процессом, так и при дублировании процесс-приемник никогда не узнает о том, что он получил доступ к новому объекту, если ему об этом каким-либо образом не сообщить.

Пример: пусть процесс А хочет предоставить доступ к своему мьютексу процессу Б:

Процесс А:

```
//создаем объект мьютекс, доступный А
HANDLE hMutexA = CreateMutex(0,FALSE,0);     //неименованный
//пусть каким-либо способом мы получили дескриптор процесса Б
                    – hProcessB
HANDLE hMutexB;//переменная (неинициализированная), куда будет
                    помещено значение дескриптора мьютекса для
                    процесса Б
//
DuplicateHandle(GetCurrentProcess(), hMutexA, hProcessB,
                    &hMutexB,
                    0,
                    FALSE,//ненаследуемый
                    DUPLICATE_SAME_ACCESS);// тот же доступ,
                    что и у hMutexA
//Используем какую-либо форму межпроцессной связи, чтобы
                    передать значение hMutexB процессу Б
```

```
//Если процессу А больше не нужен его мьютекс, - закрыть
    CloseHandle(hMutexA);
```

Замечание: никогда не пытайтесь закрыть продублированный описатель, так как он предназначен для процесса-приемника и не имеет смысла в процессе-отправителе.

```
CloseHandle(hMutexB);   //если у процесса А случайно
                        окажется описатель с таким значением, то
                        закроется объект процесса А с таким
                        описателем – последствия могут быть
                        катастрофичными!
```

7.4. ПЕРЕНАПРАВЛЕНИЕ СТАНДАРТНОГО ВВОДА/ВЫВОДА

Механизм работает только для процессов, связанных родством. Это прием, который используется с такими объектами, как файлы, каналы. Суть этого приема заключается в том, что если приложение запускается автономно и выводит данные в стандартный поток вывода, то используются потоки (структуры данных), которые создает стартовый код стандартной библиотеки. Если то же самое приложение запускается как дочерний процесс, родитель может перенаправить дочернему процессу вывод данных в файл, канал и т. д. Так же обстоят дела с потоками ввода и ошибок.

Рассмотрим пример. В родительском процессе требуется создать объект «file» и передать дочернему процессу дескриптор этого объекта для записи в качестве стандартного потока вывода. Последовательность действий при перенаправлении стандартного вывода может быть следующей. В родительском процессе:

- обеспечить наследование объекта «file», для чего создать структурную переменную типа SECURITY_ATTRIBUTES и проинициализировать поле этой переменной bInheritHandle значением TRUE;
- открыть (создать) файл с помощью функции CreateFile(), в

качестве параметра lpSecurityAttributes передав функции адрес проинициализированной структуры SECURITY_ATTRIBUTES;

• собственно для перенаправления стандартного ввода/вывода необходимо создать и проинициализировать структурную переменную типа STARTUPINFO, задав поля .dwFlags = STARTF_USESTDHANDLES, а в поле hStdOutput занести дескриптор наследуемого объекта «file»;

• при создании дочернего процесса в функции CreateProcess() в качестве параметра bInheritHandles передать TRUE, что будет указанием ОС о том, что дочерний процесс должен «унаследовать» все наследуемые дескрипторы родителя;

• после использования объекта “file” родительским процессом дескриптор объекта нужно не забыть закрыть:

• если родительскому процессу после запуска не требуется взаимодействовать с дочерним процессом, то дескрипторы дочернего процесса и его первичного потока (сформированные ОС для родителя) следует тоже закрыть.

Пример:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = 0;    //защита по умолчанию
sa.bInheritHandle = TRUE;   //описатель должен быть наследуемым
HANDLE hfile = CreateFile(<спецификация файла>,
                     GENERIC_READ | GENERIC_WRITE,
                    FILE_SHARE_READ | FILE_SHARE_WRITE,
                    &sa,
                    CREATE_ALWAYS,
                    FILE_ATTRIBUTE_NORMAL,
                    0);
if(hfile != INVALID_HANDLE_VALUE)
{
    STARTUPINFO si;
    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    si.dwFlags = STARTF_USESTDHANDLES; //флаг, который
                    заставит ОС учесть поля hStdOutput,
```

```
                hStdInput, hStdError при создании дочернего
                процесса
    si. hStdOutput = hfile; //перенаправляем дочернему процессу
                стандартный вывод в файл
    PROCESS_INFORMATION pi;
    //Создание дочернего процесса
    if( !CreateProcess(0,
        <спецификация исполняемого файла>,   //командная строка
        0,      // защита по умолчанию
        0,      // "-"
        TRUE,       //наследуемые объекты родителя наследуются
                дочерним процессом
        0,          // приоритет NORMAL.
        0,      // переменные окружения родителя
        0,      // текущая директория родителя
        &si,
        &pi)
    )
    {
        MessageBox(0, "CreateProcess failed.", "Info", MB_OK );
        CloseHandle( hFile ); //закрываем открытый дескриптор
                файла, если родителю этот файл больше не
                нужен
        return;//дальше продолжать выполнение функции не имеет
                смысла
    }
    //Закрывайте описатели, как только необходимость в них
                отпадает!!!
    CloseHandle( pi.hThread );
    CloseHandle( pi.hProcess );
    CloseHandle( hFile );
}
```

Дочерний процесс может не догадываться о том, что его стандартный вывод перенаправлен в файл. Он может любым способом использовать стандартный поток вывода:

```
printf("Write to stdout");
std::cout<<("Write to stdout");
HANDLE hout = GetStdHandle(STD_OUTPUT_HANDLE);
DWORD wereWritten;
```

```
WriteFile(hout, "Write to stdout", sizeof("Write to
                stdout"), &wereWritten,0);
```

Замечания:

- все три приведенных примера можно использовать только при перенаправлении вывода в файл, с каналами будет работать только системная функция WriteFile();
- не следует закрывать дескриптор стандартного вывода функцией CloseHandle(), так как это будет сделано стартовым кодом стандартной библиотеки.

8. БИБЛИОТЕКИ

Любую задачу всегда можно представить как набор логически обособленных понятий (данных, классов, функций, ресурсов). Такое разделение имеет ряд преимуществ:

- над задачей может работать команда программистов, каждый из которых разрабатывает свою обособленную часть проекта достаточно независимо;
- возможность использования “чужого” кода библиотек (run-time библиотеки C++, Win32API, MFC и т. д.), этим кодом могут быть часто используемые функции, данные, а также классы;
- взаимозаменяемые части приложения удобно реализовывать посредством библиотек;
- если есть надежда, что Ваш код еще кем-нибудь будет использоваться, можно оформлять свой код таким образом, чтобы им могли пользоваться другие.

Такие компоненты программного обеспечения обычно поставляются в виде библиотек. Например, стандартная библиотека C реализована в нескольких вариантах: статическая - LIBCPMTD.LIB, динамическая для отладочной версии проекта - MSVCP71D.dll, динамическая для «release» версии проекта - MSVCP71.dll.

Рассмотрим понятия, связанные с библиотеками – рис. 8.1.

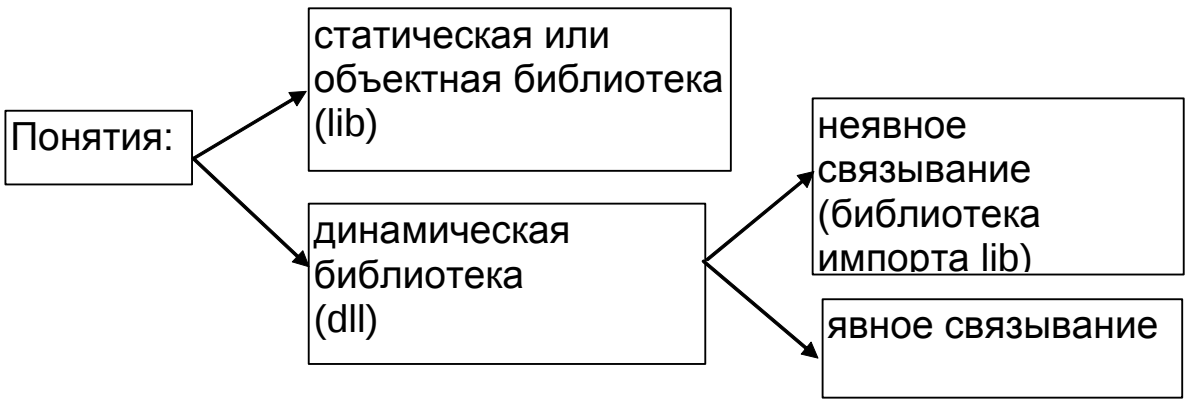

Рис. 8.1. Понятия, связанные с библиотеками

Термины, связанные с библиотеками:

- исполняемый файл – это файл с расширением .EXE, .DLL и, возможно, с другим расширением, содержащий исполняемый код, проинициализированные глобальные и статические данные и/или ресурсы;
- приложение - реализовано как исполняемый (.exe) файл, который можно вызвать непосредственно;
- динамическая библиотека – обычно расположена в .DLL файле. Динамические библиотеки не являются исполняемыми непосредственно, а служат как хранилище функций, данных, ресурсов, которые могут использовать .exe-файлы или другие dll.

Замечание: в принципе, динамическая библиотека может иметь любое расширение, даже .EXE (долгое время системные динамические библиотеки имели именно такое расширение - USER.EXE, GDI.EXE), или .DRV для драйверов устройств. Информация о типе файла определяется загрузчиком ОС не по расширению, а извлекается из заголовка файла. Но автоматически система работает только с расширением .DLL, а во всех остальных случаях требуется явная загрузка библиотеки с помощью системной функции LoadLibrary().

8.1. ОБЪЕКТНАЯ БИБЛИОТЕКА

Специфика:

- объектная библиотека – это аналог объектного модуля, поэтому подключение объектной библиотеки – это механизм этапа компоновки (то есть все внешние зависимости разрешаются компоновщиком) – рис. 8.2;
- компоновщику в командной строке нужно перечислить имена всех используемых объектных библиотек;
- после компоновки весь код такой библиотеки присутствует в исполняемом файле.

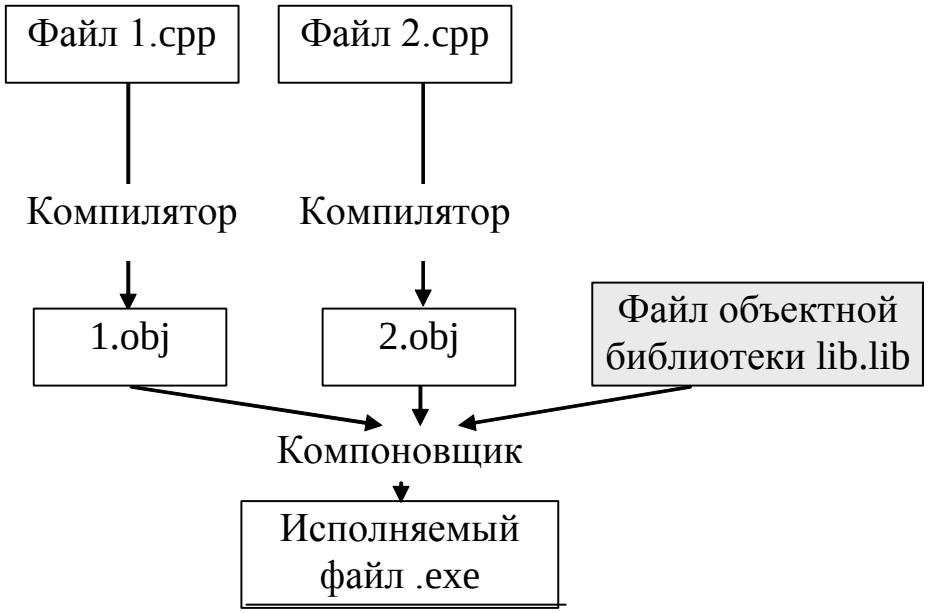

Рис. 8.2. Подключение объектной библиотеки

Следствия:

• в результате получается один самодостаточный файл, который Вы можете нести заказчику, независимо от того, имеются ли на его компьютере используемые библиотеки;

• если двум приложениям при статической компоновке понадобилась одна и та же библиотека, каждый из .exe-файлов будет располагать своей копией кода библиотеки – рис. 8.3.

У такого подхода есть неоспоримое достоинство – у пользователя просто нет возможности случайно удалить файл объектной библиотеки, так как ее код является частью исполняемого файла.

Но есть существенные недостатки:

• страничная память используется крайне неэффективно, так как каждое приложение располагает своей копией библиотеки (рис. 8.3). Очевидно, использование памяти было бы намного эффективнее, если бы в памяти находилась только одна копия, а все приложения могли бы ей пользоваться;

• не только каждое обновление собственно кода клиента требует

перекомпиляции и сборки, но и при изменении библиотечных функций Вам приходится проделывать то же самое.

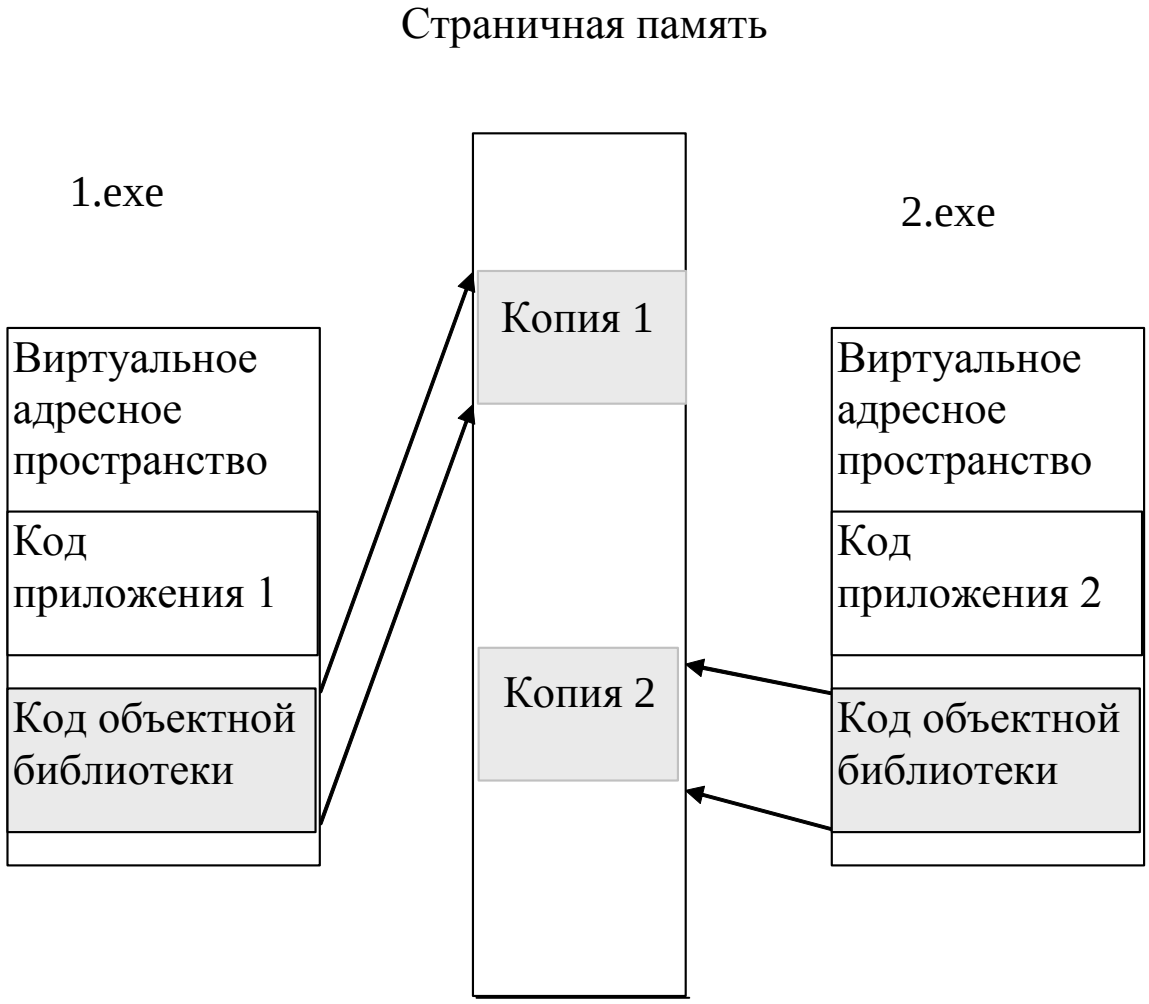

Рис. 8.3. Использование одной и той же объектной библиотеки несколькими клиентами

Рассмотрим механизм создания и использования объектных библиотек.

Создание объектной библиотеки

Объектная библиотека создается в два этапа – рис. 8.4:

- сначала компилируются все исходные .cpp модули и для каждого создается соответствующий объектный файл - .obj;
- далее программа библиотекарь (librarian) собирает из объектных модулей файл с расширением .lib.

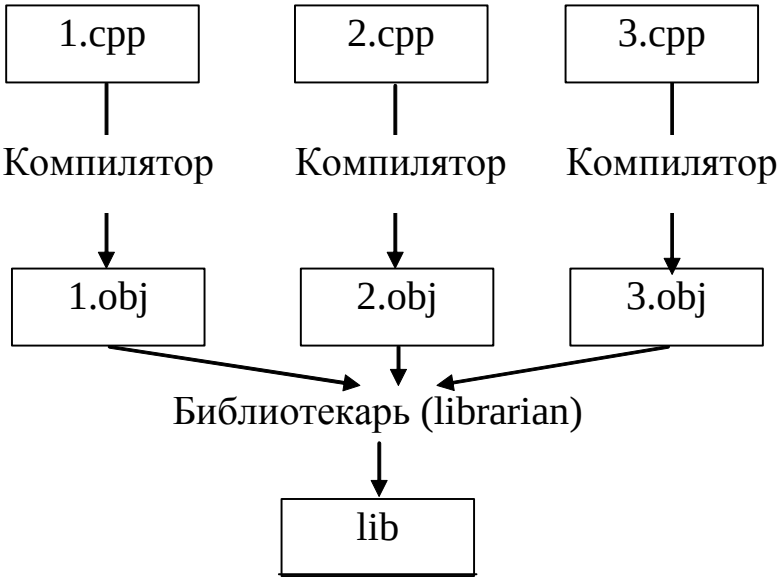

Рис. 8.4. Этапы создания объектной библиотеки

Для создания объектной библиотеки нужно в опциях командной строки компилятору задать ключ /D _LIB или в интегрированной среде разработки создать проект типа статическая библиотека. В результате будут созданы объектные модули и .lib файл с именем проекта. Это «полуфабрикат», который содержит объектный код, и соответственно называется объектной библиотекой ("Object File Library").

Что может содержать объектная библиотека

Чего точно не должна содержать объектная библиотека, так это функции main() или ее аналогов (функции WinMain()), так как это просто дополнительное хранилище функций, данных, ресурсов.

Составляющие объектной библиотеки – рис. 8.5:

- «экспортируемые» функции (для вызова клиентом);
- «экспортируемые» данные (для использования клиентом);
- внутренние функции (вспомогательные, используемые только внутри библиотеки);
- внутренние данные;
- объявления “импортируемых” функций - те функции, которые библиотека может вызывать из клиента (хотя наличие таких функций

или данных обычно свидетельствует о том, что Вы плохо логически разделили понятия);

- объявления классов и реализация методов.

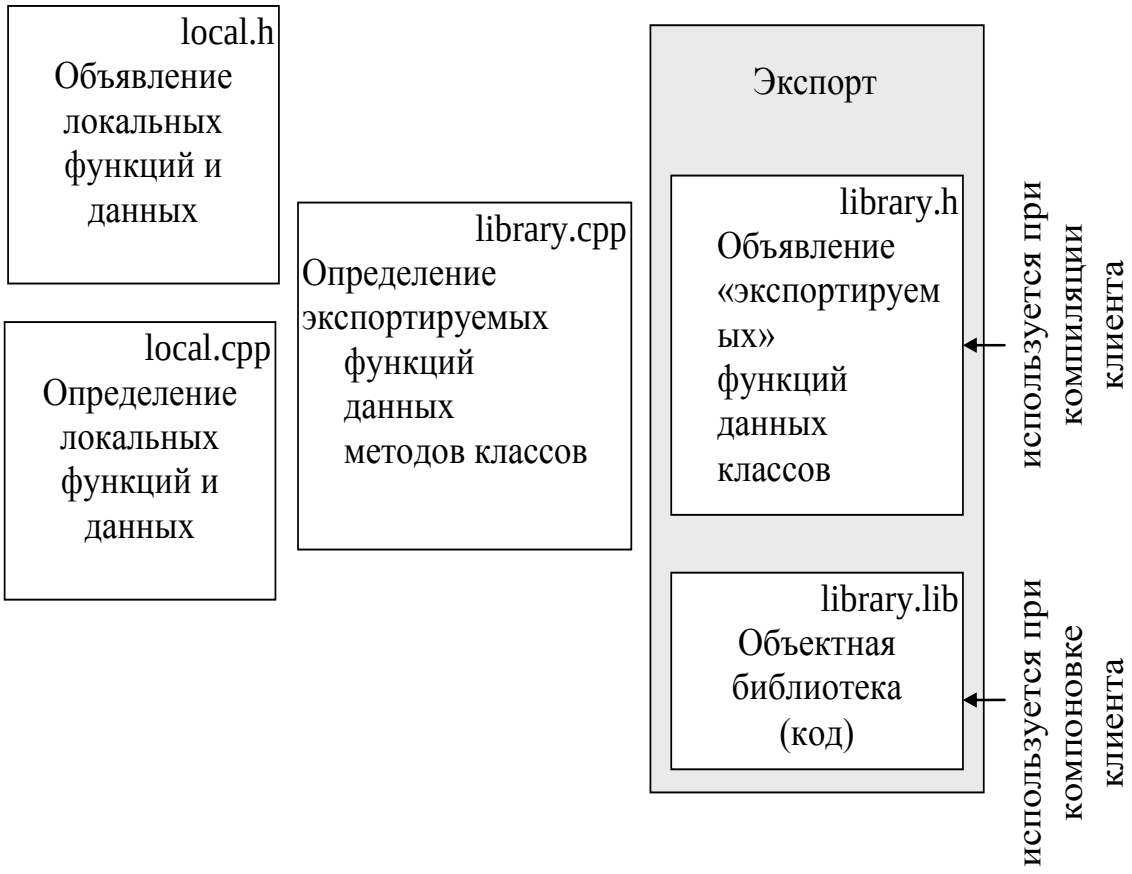

Рис. 8.5 Составляющие объектной библиотеки

Клиент, использующий объектную библиотеку

Чтобы клиент мог воспользоваться статической библиотекой, в командной строке компоновщику должны быть перечислены все используемые .lib файлы. Это можно сделать несколькими способами:

- в опциях командной строки компоновщику;
- в тексте программы с помощью директивы препроцессора #pragma.

Пример использования директивы #pragma:

#pragma comment(lib, <спецификация файла>)

Это указание компилятору поместить в объектный файл информацию о требуемом lib-файле. Компоновщик использует эту информацию при сборке исполняемого файла.

На рис. 8.6 представлено взаимодействие клиента и статической библиотеки.

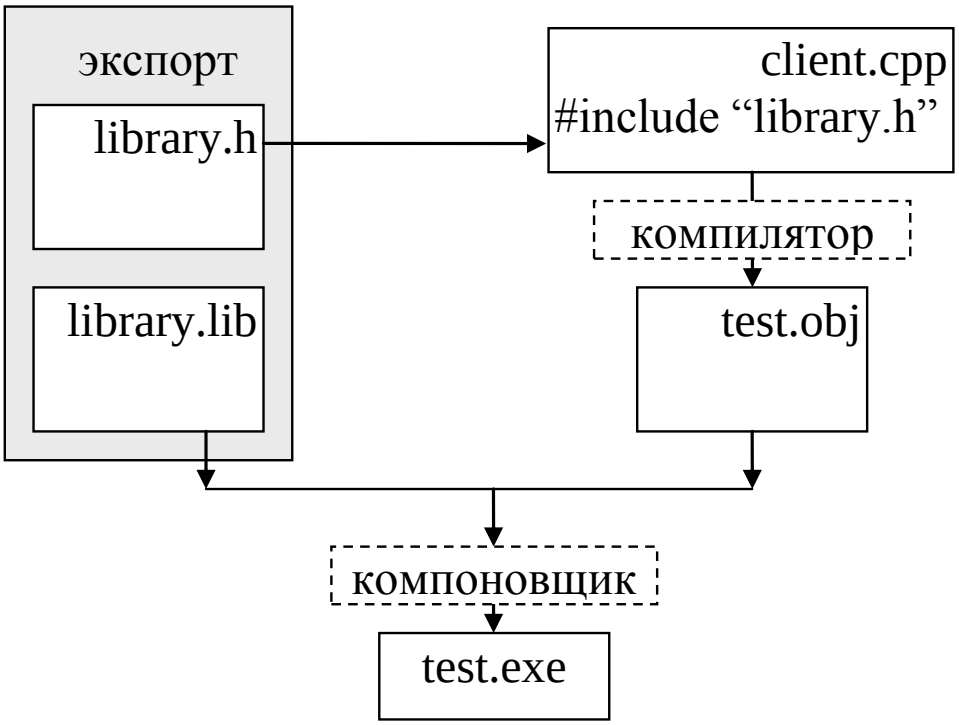

Рис. 8.6. Использование объектной библиотеки клиентом

Декорирование имен

Если клиент и библиотека созданы с помощью одного и того же инструмента, то проблем никаких не будет. Но возможны ситуации, когда:

- библиотека и клиент написаны на разных языках (правила вызова функций могут быть разными);
- разные реализации компилятора могут формировать и хранить внутренние служебные (декорированные) имена по-разному.

Поэтому при работе с библиотеками необходимо дополнительно учитывать эти особенности.

Декорирование имен применяется компилятором C++ для безопасной сборки проекта, в частности для того, чтобы компоновщик различал перегруженные функции и выявлял неуникальные определения переменных. Для этого компилятор добавляет к именам функций и переменных (хранящимся в специальной таблице в объектном модуле) дополнительные символы. Эти дополнительные символы сообщают компоновщику большое количество дополнительной информации (сколько параметров принимает функция, и какого они типа, соглашение о вызове функции и т. д.). Пример декорированных имен приведен в табл. 8.1.

Таблица 8.1

Примеры декорирования имен

`int Sum1(int,int);`	`?Sum1@@YAHHH@Z`
`int Sum2(int);`	`?Sum2@@YAHH@Z`

В результате декорирования имен компилятором C++ получаются имена функций, которые не различают программы, написанные на других языках. Поэтому, если Вы хотите, чтобы данная статическая библиотека могла быть использована клиентами, разработанными с помощью, например, компилятора C (а не C++) или написанные на языке Pascal, можно, например, отменить декорирование с помощью конструкции extern "C":

```
extern "C"  functionname /variablename ;
```

Можно также сгруппировать несколько объявлений:

```
#ifdef __cplusplus
extern "C" {
#endif
extern int libglobal;
int callappfunc(int * ptrlib);
int Sum1(int, int);
int  Sum2(int);
class A{…};
```

```
#ifdef __cplusplus
}
#endif
```

Замечание: C++ классы в большинстве случаев могут быть могут быть экспортированы только если модули импорта скомпилированы компилятором от того же производителя.

8.2. ДИНАМИЧЕСКИЕ БИБЛИОТЕКИ

Динамические библиотеки не входят в состав исполняемого модуля клиента, существуют независимо и подгружаются в адресное пространство процесса-клиента. Динамические библиотеки позволяют клиенту вызывать функции или обращаться к данным, которые не являются частью исполняемого файла клиента. Как следствие, такие функции и данные могут использовать несколько клиентов одновременно. А такое использование, в свою очередь, приводит к экономии памяти и уменьшению свопинга.

Использование механизма отображения файлов на память

Сравним действия ОС при запуске процесса и загрузке динамической библиотеки в адресное пространство процесса. При запуске приложения (.exe) ОС выполняет следующие действия:

- создает объект исполняющей системы «process»;
- вызывает функцию CreateFile() с именем исполняемого файла для того, чтобы получить доступ к содержимому:
- с помощью функции CreateFileMapping() с атрибутом SEC_IMAGE создает объект «проекция файла»;
- вызывает функцию MapViewOfFileEx() для отображения исполняемого (.EXE) файла на адресное пространство только что созданного процесса по адресу, значение которого хранится в самом исполняемом файле. Отображая исполняемый файл на адресное пространство процесса, система просматривает содержимое файла, чтобы определить, какие атрибуты защиты следует присвоить

различным страницам проецируемого образа. Например, раздел кода «.TEXT» обычно проецируется с атрибутом защиты страниц PAGE_EXECUTE_READ, тогда как раздел данных этого же файла ".DATA"— *с* атрибутом PAGE_READWRITE. То есть атрибут SEC_IMAGE заставляет систему не только спроецировать образ файла, но и автоматически подобрать подходящие атрибуты защиты страниц.

- потом создается объект первичный «поток», инициализируются регистры, адрес первого байта исполняемого кода в спроецированном представлении заносится в регистр указателя команд (Instruction Pointer - IP), и процессор может приступать к исполнению кода.

Замечания:

- если пользователь запустит второй экземпляр того же приложения, система увидит, что объект «проекция файла» для указанного исполняемого файла уже существует и не станет создавать новый объект, а просто увеличит счетчик числа пользователей. Впоследствии ОС спроецирует еще одно представление файла - на этот раз в контексте адресного пространства только что созданного второго процесса. Это позволяет эффективно использовать страничную память;
- если запущены два экземпляра одного и того же приложения, код приложения и ресурсы существуют в страничной памяти в единственном экземпляре. Естественно, стек и куча у каждого экземпляра процесса свои. Глобальные данные существуют тоже в единственном экземпляре, пока не выполняется первая операция записи. Как только один из экземпляров приложения модифицирует глобальные данные, ОС создает для каждого экземпляра свою копию – рис. 8.7.

При загрузке динамической библиотеки ОС выполняет следующие действия:

- вызывает функцию CreateFile() с именем файла динамической

библиотеки;

• с помощью функции CreateFileMapping() с атрибутом SEC_IMAGE создает объект «проекция файла»;

• вызывает функцию MapViewOfFileEx() для отображения dll-файла на адресное пространство процесса по адресу, значение которого хранится в самом dll-файле (правда, если требуемые виртуальные адреса еще «не заняты»).

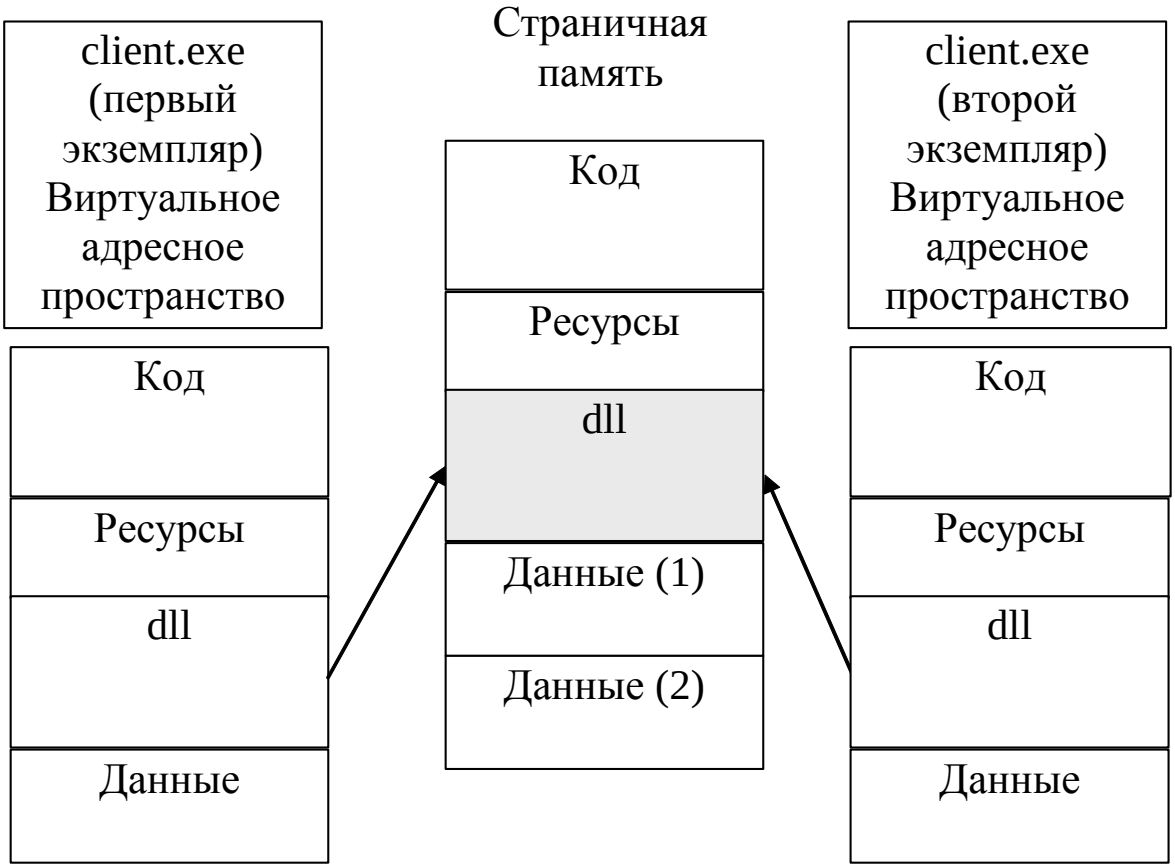

Рис. 8.7. Эффективное использование памяти посредством динамических библиотек

Замечания:

• если несколько процессов используют одну и ту же dll, то ее код и ресурсы существуют в страничной памяти в единственном

экземпляре;

• если в dll есть глобальные данные, то они тоже существуют в единственном экземпляре до первой модификации, а после первой модификации данных ОС предоставляет всем процессам, использующим эту dll, свою копию глобальных данных.

Любое Windows-приложение использует явно или неявно системные функции. Код системных функций хранится в динамических библиотеках и подгружается в адресное пространство процесса при запуске приложения. Например, основными системными библиотеками являются Kernel32.dll (управление памятью, процессами и потоками), User32.dll (поддержка пользовательского интерфейса, в том числе функции, связанные с созданием окон и передачей сообщений) и GDI32.dll (графика и вывод текста). Если бы для каждого приложения создавалась своя копия системного кода, объем любого исполняемого модуля был бы гораздо больше.

Преимущества и недостатки использования dll

Преимуществами использования динамических библиотек являются:

• экономится память за счет использования механизма отображения, так как если несколько процессов используют одну и ту же dll, на самом деле физически существует только одна копия исполняемого файла dll, а все процессы отображают ее код на свое адресное пространство. На этом базируется использование всех сервисов ОС и динамическая компоновка других библиотек (стандартной библиотеки C/C++, MFC и т. д.);

• использование системных динамических библиотек делает возможным выполнение Вашего приложения в разных средах семейства Windows без перекомпиляции, так как вся специфика заключена в реализации Win32 API, которые в свою очередь не входят в состав исполняемого файла, а подгружаются в адресное пространство процесса (или в процессе загрузки приложения или по

мере необходимости);

- так как dll может загружаться динамически в адресное пространство процесса, приложение само может во время выполнения в зависимости от каких-либо условий решить – нужно загружать соответствующий код или нет;
- приложение-клиент, использующее библиотеку, и сама dll могут быть реализовать на разных языках (VB, Си, C++, Pascal и т. д.);
- если проект разрабатывается командой, его легче координировать, если модули реализуются как отдельные независимые библиотеки. Замечание: но чем больше dll, тем больше времени уходит на инициализацию приложения (системе нужно открыть все соответствующие файлы и загрузить их в адресное пространство процесса), поэтому не стоит разбивать свое приложение на 100 библиотек только потому, что существует такая возможность;
- посредством использования dll можно эффективно использовать ресурсы, одинаковые для разных приложений. dll могут содержать шаблоны диалоговых панелей, строки, иконки, битовые изображения и т. д., которые могут использовать одновременно несколько приложений (именно так используются стандартные диалоги);
- с помощью dll можно локализовать интерфейс приложения за счет того, что код клиента – это только содержательная (вычислительная) часть, а интерфейс вынесен в dll. Например, строковые ресурсы, реализованные на разных языках, можно хранить в разных dll, подгружая только одну из них;
- dll помогают уменьшить зависимость от платформы. Разные версии Windows все-таки предлагают разную реализацию системных функций. Несмотря на это приложение-клиент в большинстве случаев может выполняться в разных семействах ОС Windows, так как системный код подгружается из соответствующих dll в адресное пространство клиента только при запуске;

- некоторые специальные возможности Windows можно реализовать только посредством dll. Например, вызов функций SetWindowsHookEx() и SetWinEventHook() возможен только при условии, что соответствующая функция (“hook notification function”) находится в dll.
- COM объекты и ActiveX компоненты обычно тоже реализуются посредством DLL.

Недостатки:

- чем больше dll использует приложение-клиент, тем больше времени уходит на инициализацию приложения;
- все необходимые dll вместе с исполняемым файлом должны быть установлены у заказчика, при этом у заказчика всегда остается возможность что-нибудь, с его точки зрения ненужное, стереть;
- когда вы устанавливаете новое программное обеспечение, и вас спрашивают «Ваша версия устарела. Не хотите ли обновить?», вы считаете, что новая версия несомненно лучше и отвечаете «Да»…А через некоторое время у вас перестает работать что-то другое (на первый взгляд не имеющее никакого отношения к обновлениям). Это означает, что в новой версии динамической библиотеки, используемой несколькими приложениями, что-то кардинально изменилось. Такая ситуация даже имеет название - «ад dll»;
- системные dll в следующих версиях ОС постоянно дополняются новыми функциями, поэтому вполне возможна такая ситуация, когда вы приносите заказчику свое приложение, а при запуске вашего приложения загрузчик ОС выдает сообщение: «Функция не найдена…». Скорее всего, это означает, что у заказчика стоит более ранняя версия ОС, и там такой функции действительно нет.

Отличия приложения и dll

С точки зрения пользователя приложение - это то, что можно запустить непосредственно, а о существовании dll пользователь может не подозревать.

С точки зрения программиста:

- назначением dll является хранение набора автономных функций, данных или ресурсов, пригодных для использования любым приложением. В dll могут находиться не только системные или библиотечные сервисы. Если Вы предполагаете, что Вашими функциями будут пользоваться другие приложения, Вы тоже можете оформить их как dll;
- имеются отличия в реализации, так как функции dll пишутся в расчете на то, что их будет вызывать приложение-клиент или другая dll, поэтому они должны быть специальным образом оформлены – для «экспорта». То же самое справедливо и для экспортируемых данных:
- исполняемые файлы приложения и dll отличаются не только расширением, но и информацией в заголовке файла Файлы с исходным кодом компилируются и компонуются так же, как и при создании .exe-файла, но вы должны указать компоновщику ключ /DLL. Это заставляет компоновщик:
  - записывать в конечный файл информацию, по которой загрузчик операционной системы определит, что данный файл является динамической библиотекой, а не приложением;
  - не искать точку входа (она не обязательна);
- принятые ограничения - в пользовательских dll обычно отсутствует цикл обработки сообщений, и не создаются окна.

С точки зрения ОС:

- при запуске приложения ОС создает объекты «process», «thread», «file», «file mapping» и выделяет новое адресное пространство. При подключении dll создаются объекты «file», «file mapping», код и данные dll загружается в уже сформированное адресное пространство;
- если в коде dll захватываются какие-нибудь ресурсы, то их владельцем является процесс.

Отличия статических и динамических библиотек

1. Код статической библиотеки подключается к исполняемому файлу во время компоновки и составляет единое целое, поэтому любые изменения (как в приложении, так и в библиотеке) требуют перекомпиляции и перекомпоновки. Код динамической библиотеки хранится в отдельном независимом файле и подключается к клиенту во время загрузки или выполнения.

2. Каждое приложение, использующее статическую библиотеку, располагает своей собственной копией библиотеки, так как статические библиотеки входят в состав исполняемого файла. В то время как код dll хранится в единственном экземпляре в страничной памяти и отображается на адресные пространства нескольких клиентов.

3. Статические библиотеки содержат только код и данные, так как они хранятся как совокупность объектных файлов. DLL могут содержать код, данные и ресурсы (битовые изображения, иконки, курсоры и т.д.), так как они хранятся как исполняемые файлы.

4. Статическую библиотеку создает библиотекарь, динамическую – компоновщик.

5. Код статической библиотеки располагается в .LIB-файле и является аналогом объектного модуля. При создании DLL компоновщик (и компилятор) создает два файла .dll (код) и .lib (библиотека импорта, которая в свою очередь содержит ссылку на файл .dll и ссылки на функции и данные внутри файла) – рис. 8.8.

Замечание: также как объектные библиотеки, библиотека импорта имеет расширение .lib и используется компоновщиком для разрешения ссылок на функции в исходном тексте программы. Однако в библиотеках импорта программного кода нет. Вместо них в библиотеках импорта находится информация, необходимая компоновщику при сборке клиента для установки таблицы ссылок внутри исполняемого файла клиента. Эта таблица предназначена для динамического связывания вызовов клиента с функциями и данными, хранящимися в dll.

Рис. 8.8. Отличия статических и динамических библиотек

Понятия, связанные с dll

Использование динамических библиотек связано со следующими понятиями: подключение к клиенту, выполнение кода dll в контексте клиента, отключение от клиента. Рассмотрим эти понятия подробнее.

1. Подключение dll к клиенту.

Для того чтобы код функции мог выполняться или к данным

можно было бы обратиться, они должны находиться в виртуальном адресном пространстве процесса-клиента. Отображение кода dll на адресное пространство клиента называется подключением.

Подключение dll к клиенту может осуществляться двумя способами:

- при неявном связывании в стартовом коде стандартной библиотеки;
- при явном связывании при вызове клиентом функции LoadLibrary().

Каким бы способом не подключалась библиотека, если еще ни один клиент не запрашивал эту dll, создаются новые объекты исполняющей системы “file” и “file mapping”, счетчики пользователей этих объектов устанавливаются в 1. Если же соответствующие объекты уже существуют, то их счетчики просто инкрементируются.

Специфика взаимодействия клиентов и dll заключается в том, что, с одной стороны, исполняемый код dll должен быть доступен каждому клиенту в его адресном пространстве, с другой стороны, именно потому, что код dll могут одновременно выполнять несколько клиентов, он должен быть оформлен с учетом множественного использования.

Замечания:

- для разных процессов одна и та же библиотека (одни и те же страницы) может быть отображена на разные виртуальные адреса;
- в Win32 для каждого процесса, использующего dll, по умолчанию создается своя копия глобальных данных.

2. Выполнение кода DLL в контексте клиента.

Как только dll спроецирована на адресное пространство процесса-клиента, ее функции и данные доступны всем потокам этого процесса. И таким образом получается, что dll просто предоставляет клиенту дополнительный код, данные и ресурсы.

Контекстом называется совокупность стека и регистров. Выполнение кода dll в контексте клиента означает:

• функция dll при вызове пользуется стеком вызвавшего ее клиентского потока и размещает в этом стеке свои локальные данные;

• созданные кодом dll объекты принадлежат процессу (сама dll ничем не владеет). Например, если код dll вызывает функцию VirtualAlloc(), резервируется регион в адресном пространстве процесса, то в случае выгрузки dll из адресного пространства процесса, регион не освободится. Так как регион принадлежит процессу, он будет освобожден в случае явного вызова функции VirtualFree() или неявно при завершении процесса.

Замечание: функции dll должны быть реентерабельными, то есть должны безопасно выполняться несколькими потоками. Если приложение многопоточное, то вполне может возникнуть следующая ситуация. Один из потоков вызвал функцию, экспортируемую из dll, и начал выполняться. Квант времени, отпущенный ОС данному потоку, истек, и ОС переключила управление на другой поток, который в свою очередь обратился к прерванной функции. Когда могут возникнуть проблемы:

• если потоки принадлежат разным процессам, то каждый процесс располагает своей копией глобальных данных, обращение к одним и тем же глобальным данным невозможно, поэтому проблем быть не может;

• если потоки принадлежат одному процессу и оперируют только стековыми локальными данными, то конфликтов тоже быть не может, так как стек у каждого потока свой;

• если же два потока одного процесса посредством одной и той же функции оперируют глобальными или динамическими данными (в куче процесса), то при реализации такой функции нужно предусмотреть защиту от таких ситуаций. Решением проблемы может быть синхронизация доступа к данным или использование локальной памяти потока (TLS – “thread locale storage”).

3. Отключение dll от клиента.

Отключение кода dll от клиента означает освобождение

соответствующих записей в таблице страниц процесса. При этом вовсе необязательно страницы, на которые отображен код dll в страничной памяти, будут освобождены, так как у этой dll могут еще оставаться другие пользователи. Страничная память, выделенная для dll, будет освобождена, только если счетчик пользователей соответствующих объектов “File” и “File Mapping” станет равным 0.

В какой момент происходит отключение:

- для неявно связанных dll при завершении процесса отключение dll от клиента производит стартовый код стандартной библиотеки Си, который вызывает функцию ExitProcess(), где счетчик пользователей соответствующих объектов “File” и “File Mapping” уменьшается на 1;
- для явно связанных библиотек код dll отключается от клиента в результате вызова функции FreeLibrary(). Если программист забыл вызвать функцию FreeLibrary(), то отключение происходит при завершении процесса по общим правилам.

Экспорт из dll

Механизм экспорта заключается в создании в исполняемом файле dll помимо кода, как такового, специальных таблиц экспорта. Посредством таких таблиц загрузчик ОС или системная функция GetProcAddress() могут определить относительные адреса экспортируемых функций и данных. Вид таблиц и механизм определения адреса приведен на рис. 8.9.

В заголовке файла с расширением dll создается структура IMAGE_EXPORT_DIRECTORY, в которой хранится информация двух видов:

- данные, которые характеризуют сам исполняемый файл (версия, дата создания и т. д.):
- раздел экспорта, который позволяет получать доступ к экспортируемым dll понятиям, в частности:

```
struct IMAGE_EXPORT_DIRECTORY{
    Поля, которые характеризуют сам исполняемый файл: где хранится имя
    исполняемого файла, версия, дата создания…

    Поля, которые характеризуют экспортируемые понятия:
    количество экспортируемых функций и данных;
    количество экспортируемых  имен (может быть больше, так как одной
    функции можно сопоставить несколько имен);
    базовое значение (base) для обращения по номерам (порядковый номер,
    обычно 1);
    указатель на массив смещений функций относительно начала dll;
    указатель на массив имен функций
    указатель на массив номеров;
    …
};
```

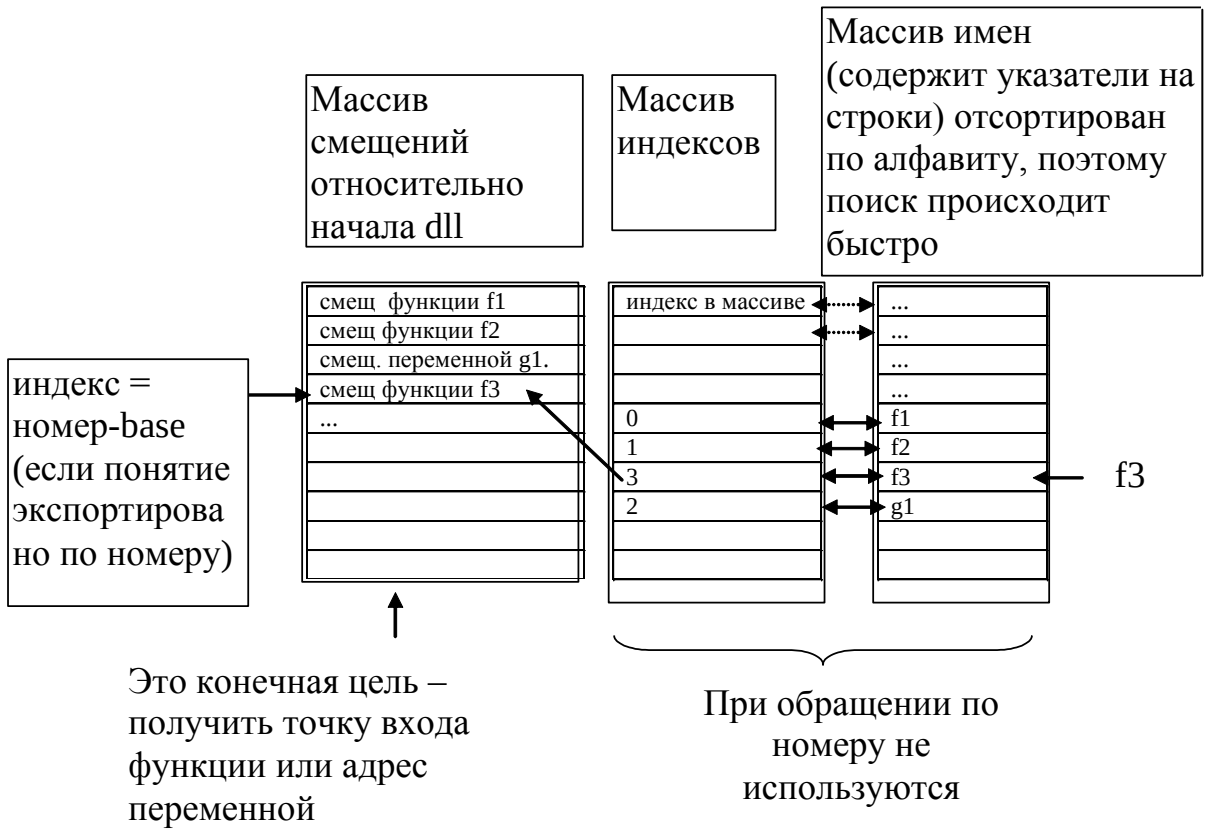

Рис. 8.9. Механизм экспорта из dll

- число экспортируемых понятий (данных и функций);
- число экспортируемых имен (которое может быть больше, так как одно и то же понятие можно экспортировать под разными именами);
- базовое значение для номеров (если номера начинаются с 100, то совершенно незачем отводить в массиве номеров неиспользуемые первые 100 элементов);
- указатели на три массива (смещений, номеров и имен).

В массиве смещений содержатся смещения функций или данных относительно начала dll. Получение этих смещений и есть конечная цель, так как, располагая этим смещением, останется только добавить виртуальный адрес, по которому загружена dll, чтобы получить адрес требуемого понятия.

Массив имен содержит указатели на строки с именами экспортируемых функций и данных. Эти строки всегда хранятся отсортированными по алфавиту, поэтому поиск происходит очень быстро.

Массив номеров является «переходным мостиком» от массива имен к массиву смещений. Каждый его элемент соответствует строке из массива имен и содержит индекс в массиве смещений.

У клиента есть две возможности получения адреса экспортируемого понятия: по номеру (такой номер принято называть “hint”) и по имени. Поэтому таблицы, сформированные в dll, должны предоставлять клиенту обе возможности.

1. Экспорт по номеру.

Это самый простой, но не самый удобный для клиента способ. В этом случае клиент должен знать номер (“hint”), который с точностью до значения “base” просто является индексом в массиве смещений. По индексу, равному разности «hint-base», из массива смещений выбирается значение и к этому значению прибавляется адрес, по которому загружена dll. Таким образом вычисляется адрес экспортируемого понятия.

Два других массива при обращении по номеру не используются.

2. Экспорт по имени.

Если клиент располагает только именем экспортируемого из dll понятия, то получение адреса происходит в несколько этапов:

- сначала в массиве имен ищется совпадение с указанным именем (поиск происходит быстро, так как все строки отсортированы);
- если найдено совпадение, по индексу совпавшей строки из массива номеров извлекается индекс в массиве смещений;
- к полученному смещению прибавляется адрес, по которому загружена dll.

Замечания:

- использовать импорт по именам удобно, но это удобство иногда оборачивается проблемами, так как:
  - клиентское приложение и dll могут быть написаны на разных языках;
  - единого стандарта на декорирование имен нет, поэтому декорированное имя может быть разным даже в зависимости от версии компилятора;
- содержимое раздела экспорта dll можно посмотреть с помощью утилиты DumpBin.

Механизм экспорта из dll (или .exe-файла)

При создании динамической библиотеки программист должен оформить экспортируемые функции, данные и классы таким образом, чтобы клиент мог ими воспользоваться. Выполняя указания программиста, компилятор и компоновщик формируют в объектных файлах и исполняемом модуле дополнительную информацию, которая дает возможность обращаться к функциям или данным по номерам или по именам другому исполняемому модулю.

В результате совместных усилий программиста, компилятора и компоновщика создается файл с расширением dll, в котором содержится код и раздел экспорта, а также вспомогательный файл с

расширением .lib, в котором никакого кода нет, зато в нем содержится информация об экспортируемых понятиях (экспортируемые номера и имена) и смещения этих понятий относительно начала dll.

Замечание: несмотря на то, что у этого вспомогательного файла расширение такое же, как у объектной библиотеки, содержимое принципиально отличается. Такой модуль называется библиотекой импорта.

Замечание: экспортируемые понятия можно посмотреть с помощью утилиты dumpbin:

```
dumpbin.exe -exports MyDll.dll
```

Программист может осуществить экспорт несколькими способами – рис. 8.10.

Рис. 8.10. Способы экспортирования понятий из dll

Замечание: механизмы можно использовать одновременно, если они друг-другу не противоречат. Рассмотрим эти возможности подробнее.

Спецификация dllexport

Ключевое слово __declspec является спецификой Microsoft и используется для того, чтобы предоставить компилятору дополнительную информацию о классе хранения. Поэтому имеет смысл только в том случае, когда вы создаете Win32 dll и пользуетесь

компилятором VC++. В частности, с помощью комбинации __declspec(dllexport) программист может дать компилятору указание о том, что программное понятие подлежит экспорту. Экспортировать можно:

- данные;
- функции;
- классы;
- отдельные методы классов.

Примеры объявлений экспортируемых понятий:

```
// Заголовочный файл mylib.h
extern __declspec(dllexport) int Global; // объявление
                    экспортируемого данного
__declspec(dllexport) int __cdecl f1(void);// объявление
                    экспортируемой функции
__declspec(dllexport) void __stdcall f2(int);// экспорт
                    функции
class __declspec(dllexport) A{  // экспорт всех методов класса
 ... объявления членов класса ...
};
```

Замечание: конструкция __declspec(dllexport) должна быть указана как при объявлении, так и при определении. Примеры определений экспортируемых понятий:

```
// Файл реализации mylib.cpp
__declspec(dllexport) int Global; //определение
                    экспортируемого данного
__declspec(dllexport) int __cdecl f1(void)
{
      //тело функции
}
И т.д.
```

Механизм экспорта:

- программист описывает экспортируемые понятия посредством __declspec(dllexport);
- компилятор, встречая конструкцию __declspec(dllexport), понимает, что функция или переменная предназначена для экспорта, поэтому для таких идентификаторов компилятор генерирует дополнительную информацию в объектном файле о том, что понятие

подлежит экспорту;

- обнаружив такую информацию, компоновщик:
  - создает .lib-файл (а если __declspec(dllexport) в объявлении нет, то lib-файл не создается) со списком идентификаторов, экспортируемых из данной dll.Файл с расширением .lib-это суррогат библиотеки, который будет связан с вызывающей (.exe) программой для разрешения внешних ссылок и создания действительных связей с файлом dll во время загрузки
  - вставляет в конечный исполняемый файл (.dll или .exe) раздел экспорта;

Использование файла определений

Файл описания модуля (“module definition”) с расширением .DEF – это текстовый файл, предназначенный для того, чтобы снабдить компоновщика информацией об экспортируемых объектах. Для того чтобы компоновщик «учитывал» файл определений, в опциях командной строки компоновщику должен быть указан ключ /DEF <спецификация файла>.

Чтобы экспортировать понятия с помощью .DEF файла, программист должен, как минимум, задать инструкции LIBRARY и EXPORTS. Инструкция LIBRARY указывает компоновщику, что следует создать динамическую библиотеку (.dll), библиотеку импорта (.lib) и сформировать в файле .dll раздел экспорта. Формат директивы LIBRARY:

```
LIBRARY [имя][BASE=адрес]
```

Где имя – это имя динамической библиотеки. Если это имя не задано в инструкции явно в .DEF-файле, то его можно также сформировать с помощью опции /OUT в командной строке компоновщику.

Поле «BASE=адрес» задает виртуальный адрес, по которому ОС должна отображать содержимое библиотеки на адресное пространство процесса. Поле является опциональным, и если оно не указано, то ОС по умолчанию пытается сделать отображение по адресу 0x10000000. Значение указываемого адреса является только рекомендацией ОС, так как на момент отображения данной

библиотеки адрес уже может быть «занят» другой библиотекой. Пример использования инструкции:

```
LIBRARY   MyLib//где MyLib – это имя файла, адрес загрузки по умолчанию
```

Второй составляющей, обеспечивающей экспорт, является инструкция EXPORTS. С инструкции EXPORTS начинается секция экспортируемых из dll имен. Таких секций в def-файле может быть несколько. Экспортируемые понятия должны быть описаны согласно следующему формату:

имя[=*внутреннее имя*] [@*номер* [NONAME]] [PRIVATE] [DATA]

Рассмотрим обязательные и опциональные поля описания экспортируемых понятий.

1. Поле «имя» является обязательным. Здесь указывается то имя, которое будет занесено компоновщиком в таблицу экспорта. Если другие (необязательные) составляющие описания экспортируемого понятия отсутствуют, например:

```
EXPORTS
MyFunc
```

то, когда компоновщик «разбирает» .def-файл, он видит:

- имя MyFunc подлежит экспорту;
- это имя с точностью до правил декорирования совпадает с именем в объектном модуле.

Поэтому компоновщик помещает в таблицу экспорта недекорированное имя MyFunc.

2. Поле «внутреннее имя» используется для экспорта понятия под псевдонимом. Например, в dll используется имя MyFunc, а экспортировать Вы хотите понятие по имени имя YourFunc. Тогда описание экспорта будет выглядеть:

```
EXPORTS
YourFunc = MyFunc
```

Это указание компоновщику занести в таблицу экспорта псевдоним YourFunc, а имя MyFunc не экспортировать.

3. Поле «@ordinal» позволяет сопоставить экспортируемому имени числовое значение (номер). Это значение будет использовано в

качестве индекса в массиве смещений раздела экспорта. Например:

```
EXPORTS
MyFunc @1
```

приводит к экспорту по недекорированному имени «MyFunc» и по номеру «1». Это означает, что клиент сможет получать адрес, пользуясь как именем, так и номером.

4. Поле NONAME указывает компоновщику экспортировать только по номеру (при попытке клиента обратиться по имени в доступе будет отказано). Например:

```
EXPORTS
MyFunc @1 NONAME//это говорит о том, что Вы не хотите
                  экспортировать имя функции – только
                  порядковый номер hint=1 (при этом само имя
                  в таблице все равно присутствует, но его не
                  разрешается использовать)
```

5. Поле PRIVATE предписывает компоновщику не помещать имя в .lib-файл, а таблице экспорта оно все равно будет. Например:

```
Sum PRIVATE
```

Замечание: такие имена можно использовать только при явном связывании библиотеки с клиентом.

6. Поле DATA дает компоновщику информацию о том, что экспортируется данное, а не функция. Например:

```
EXPORTS
MyGlobal DATA
```

Если одновременно используются поля PRIVATE и DATA, то PRIVATE должно предшествовать DATA.

Замечание: перечисленные поля можно комбинировать, например:

```
EXPORTS
YourFunc = MyFunc @1 PRIVATE
```

Без использования файлов описаний не обойтись, когда:

- требуется экспортировать недекорированное имя;
- предполагается экспорт понятий по номеру;
- необходимо экспортировать под псевдонимом;
- не следует заносить экспортируемое имя в библиотеку

импорта;

- и т. д.

Использование опций командной строки компоновщику

Посредством спецификации /export в командной строке компоновщику можно перечислить экспортируемые имена. Формат спецификации:

```
/EXPORT:имя[,@номер[,NONAME]][,DATA]
```

Формирование спецификации можно сделать двумя способами: непосредственно в командной строке компоновщику или посредством директивы препроцессора «#pragma comment», которая вставляет соответствующие указания компоновщику в объектный модуль.

Директива препроцессора #pragma пишется непосредственно в тексте программы. Синтаксис использования директивы:

```
#pragma comment(linker, "/export:имя[,@номер[,NONAME]][,DATA])
```

Пример использования:

```
#pragma comment(linker, "/export:MyFunc=_MyFunc@8")
```

Эта строка заставляет компилятор «объяснить» компоновщику, что функция с именем *MyFunc* должна быть экспортирована с той же точкой входа (смещением), что и функция с декорированным именем *_MyFunc@8*. Этот способ менее удобен, так как Вы явно должны сформировать и указать декорированное имя. Кроме того, такая dll экспортирует на самом деле два символа для одной и той же функции. Но позволяет обойтись без .def-файла.

Импорт функций и переменных клиентом из dll

Чтобы клиентское приложение могло вызывать функцию из dll или обращаться к данным dll, код dll (образ dll файла) должен быть спроецирован на адресное пространство клиентского процесса. Существуют два способа связывания кода dll с кодом клиента: явное связывание (во время выполнения клиента) и неявное (во время загрузки клиента). И в одном, и в другом случае для импорта понятий из dll используется раздел экспорта, содержащийся в dll, но механизмы получения адресов импортируемых функций и данных клиентом отличаются.

Замечание: разные клиенты могут использовать одну и ту же dll любым способом. Но эти механизмы не взаимозаменяемые, так как действия программиста и ОС разные.

Неявное (implicit) связывание dll с клиентом

Неявное связывание – это самый распространенный способ проецирования образа dll на адресное пространство процесса-клиента.

Иногда неявное связывание называют статической загрузкой dll (не путать со статическими библиотеками). При неявном связывании все требуемые клиенту динамические библиотеки отображаются на адресное пространство процесса при запуске клиента. Такое использование динамической библиотеки требует наличия файла библиотеки импорта (.lib) и заголовочного файла с прототипами импортируемых функций и объявлениями импортируемых данных и классов.

Механизм импорта функций и данных из dll.

1. Программист пишет текст программы и обращается к импортируемым понятиям как, если бы импортируемые функции или данные присутствовали в коде клиента. Для облегчения работы компоновщика программист может (это необязательно) описать импортируемые понятия посредством специфического для Microsoft атрибута класса хранения __declspec(dllimport). Такое описание предоставит компилятору дополнительную информацию о том, что функции или данные находятся в dll. Пример:

```
__declspec(dllimport) int MyFunc(int, const char*);
__declspec(dllimport) extern int Global;
```

Кроме того, программист обязан указать компоновщику все используемые библиотеки импорта (.lib). Это можно сделать непосредственно с помощью опции /LIB <спецификация файла> или с помощью директивы #pragma в тексте программы, например:

```
@pragma comment(lib, <спецификация файла>)
```

2. При компиляции файлов клиента компилятор использует объявления импортируемых понятий, предоставленные динамической библиотекой посредством заголовочного файла, и генерирует

объектный модуль. Если в объявлении внешнего по отношению к компилируемому файлу понятия компилятор встречает конструкцию __declspec(dllimport), он:

- формирует в объектном модуле дополнительную информацию о том, что данный идентификатор находится в dll;
- генерирует косвенный вызов функции (при обычном вызове внешней по отношению к данному модулю функции компоновщик подставляет адрес функции, а при косвенном адрес безусловного перехода на «заглушку»).

Эта дополнительная информация необязательна, но существенно сокращает время работы компоновщика, а косвенные вызовы унифицируют передачу управления на неизвестный на этапе компоновки адрес импортируемого понятия.

3. Компоновщику в командной строке необходимо указать все библиотеки импорта (.lib), соответствующие требуемым динамическим библиотекам. Компоновщик:

- собирает воедино код объектных модулей клиента, но не располагает кодом динамических библиотек. Поэтому компоновщик может сформировать адресные части только для клиентских данных и функций, а адреса импортируемых из dll понятий еще не известны. Не найдя определения таких имен в объектных модулях, компоновщик анализирует файлы библиотек импорта (.lib);
- внедряет в исполняемый файл специальную таблицу импорта, в которой фигурируют все импортируемые функции и данные, и частично ее заполняет – табл. 8.2, используя информацию .lib-файлов. Каждая строка таблицы содержит имя dll файла, имя импортируемой функции или переменной, ее номер. Адреса данных и функций, импортируемых из dll, на этом этапе еще неизвестны, поэтому в качестве адреса в соответствующей строке таблицы фигурирует «заглушка», а формирование адреса «откладывается» на время загрузки приложения. Таблица импорта впоследствии при загрузке клиента будет сформирована окончательно, а при выполнении будет

работать как «переходный мостик» между кодом клиента и кодом dll;

- в команду обращения к импортируемому понятию компоновщик вставляет в качестве адресной части адрес «заглушки» в таблице импорта;
- только в том случае, когда компоновщик не обнаруживает определения понятия в объектных модулях и не находит соответствия в библиотеках импорта, он выдает ошибку о неопределенной внешней функции или внешнем данном.

Таблица 8.2

Импорт понятий из dll при неявном связывании

Имя файла динамической библиотеки	Номер	Экспортируемое имя	Адрес
MyDll.dll	1 Положительное целое значение (hint), задаваемое при экспорте по номеру. Используется в качестве индекса в таблице смещений раздела экспорта dll	MyFunc Используется при экспорте по имени. Присутствует в массиве имен раздела экспорта dll	«Заглушка» для адреса. Само значение будет сформировано при загрузке приложения
OtherDll.dll	5	OtherFunc	«Заглушка»
…			

Замечание: содержимое таблицы импорта клиентского приложения можно посмотреть с помощью утилиты «dumpbin»:

dumpbin.exe <имя приложения>.exe /IMPORTS

4. При запуске приложения загрузчик ОС анализирует таблицу импорта и:

• пробует найти все перечисленные dll и посредством механизма отображения файлов на память загрузить их в адресное пространство приложения;

• пробует найти все импортируемые из dll функции или данные, формирует их адреса в таблице импорта (к адресу загрузки прибавляет относительный адрес от начала dll).

5. При выполнении программы адрес любого импортируемого понятия извлекается из таблицы импорта. Всякий раз, когда приложение ссылается на один из таких идентификаторов, сгенерированный компилятором код выбирает его адрес из таблицы и таким образом косвенно осуществляет вызов функции или обращение к данному.

Механизм неявного связывания кода клиентского приложения с кодом dll представлен на рис. 8.11.

Замечания:

• в таблице импорта указывается только имя dll, так как на разных компьютерах требуемая dll может находиться в разных директориях, поэтому поиск dll загрузчик ОС осуществляет в следующем порядке:

- каталог, содержащий исполняемый файл клиента (файл .exe);
- текущий каталог процесса;
- системный каталог Windows;
- основной каталог Windows;
- каталоги, указанные в переменной окружения PATH:
- если файл не найден, система выводит сообщение и немедленно завершает клиентский процесс;

• при неявном связывании разработчик динамической библиотеки обязательно должен предоставить клиенту, помимо собственно библиотеки (.dll), библиотеку импорта (.lib). Библиотека импорта - это мостик между клиентом и dll (передаточный вектор), «невидимый» программисту, поэтому при таком способе связывания

программист пользуется функциями и данными из dll, как будто они располагаются в самом клиенте;

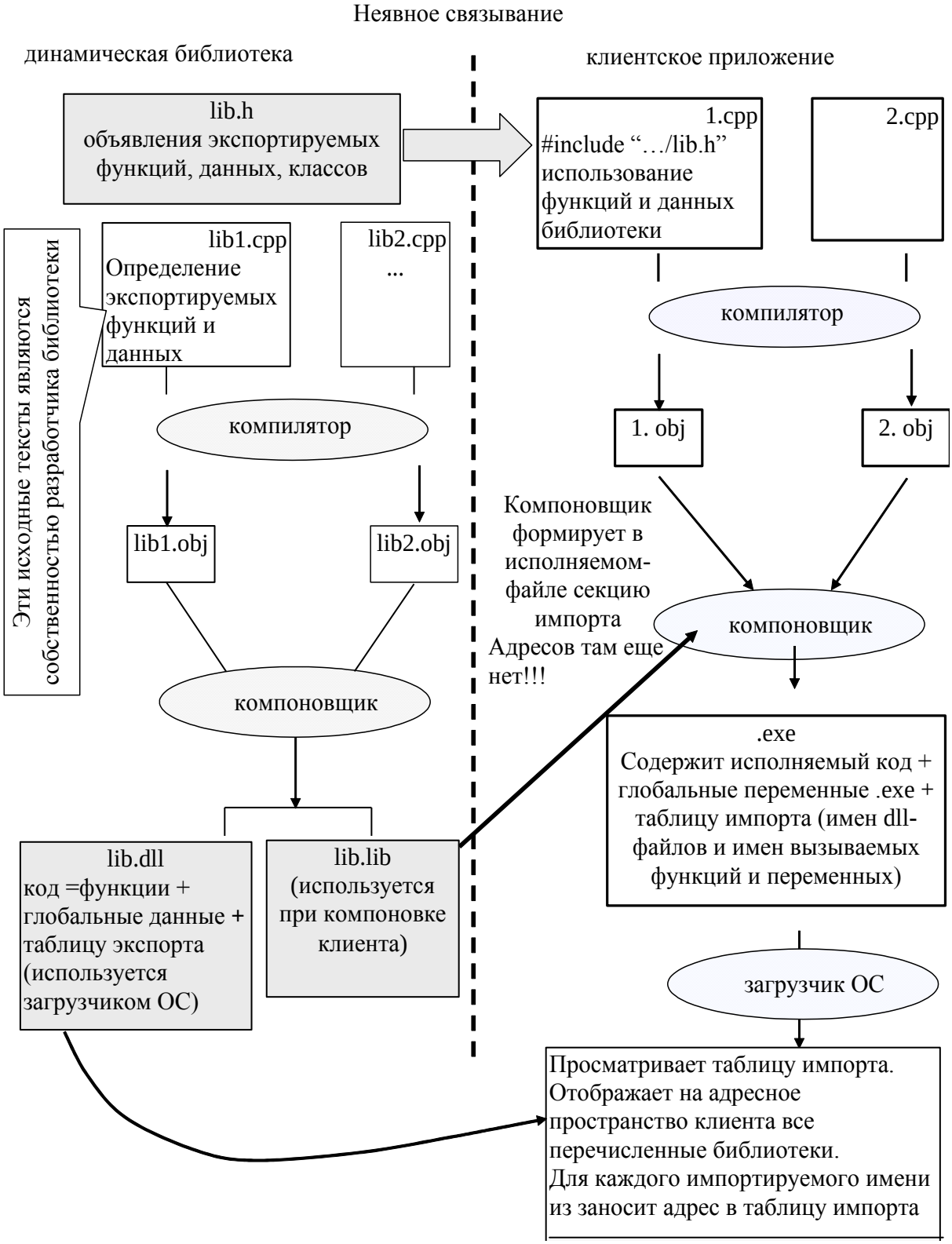

Рис. 8.11. Использование dll посредством неявного связывания

• библиотеки, спроецированные на адресное пространство процесса таким методом, не отключаются от него до завершения клиентского процесса;

• так как одна dll может в свою очередь импортировать функции и данные из другой (то есть сам dll-модуль может иметь свою секцию импорта), при загрузке клиентского приложения загрузчик ОС осуществляет итеративный процесс анализа таблицы импорта, поиска и отображения нужных dll. Этот итеративный процесс можно прервать посредство ключа /DELAYLOAD <имя библиотеки>.

Заголовочный файл dll

Обычно при неявном связывании разработчик dll предоставляет клиенту файлы с расширениями: .dll, .lib и .h (заголовочный файл), в котором содержатся прототипы всех экспортируемых из dll функций, объявления переменных и классов. Со стороны библиотеки все эти понятия должны быть экспортируемыми (это можно осуществить в частности с помощью атрибута класса хранения __declspec(dllexport)), а со стороны клиента те же самые понятия следует описать (это необязательно, но упрощает работу компоновщика) как импортируемые посредством __declspec(dllimport). При создании двух отдельных файлов описаний для экспорта и для импорта программист все время должен следить за синхронизацией изменений (это неудобно и может порождать ошибки). Для того чтобы один и тот же заголовочный файл можно было использовать при компиляции как самой библиотеки, так и клиента, использующего данную библиотеку, его можно оформить с помощью директив условной трансляции следующим образом – рис. 8.12.

Тогда при подключении заголовочного файла “lib.h” имя MY_DLL является для препроцессора определенным, и вместо идентификатора LINKDLL подставляется __declspec(dllexport), и все объявленные понятия становятся экспортируемыми. А когда тот же заголовочный файл подключается клиентом, имя MY_DLL для

препроцессора не определено, поэтому вместо идентификатора LINKDLL подставляется __declspec(dllimport), и все объявленные понятия становятся импортируемыми.

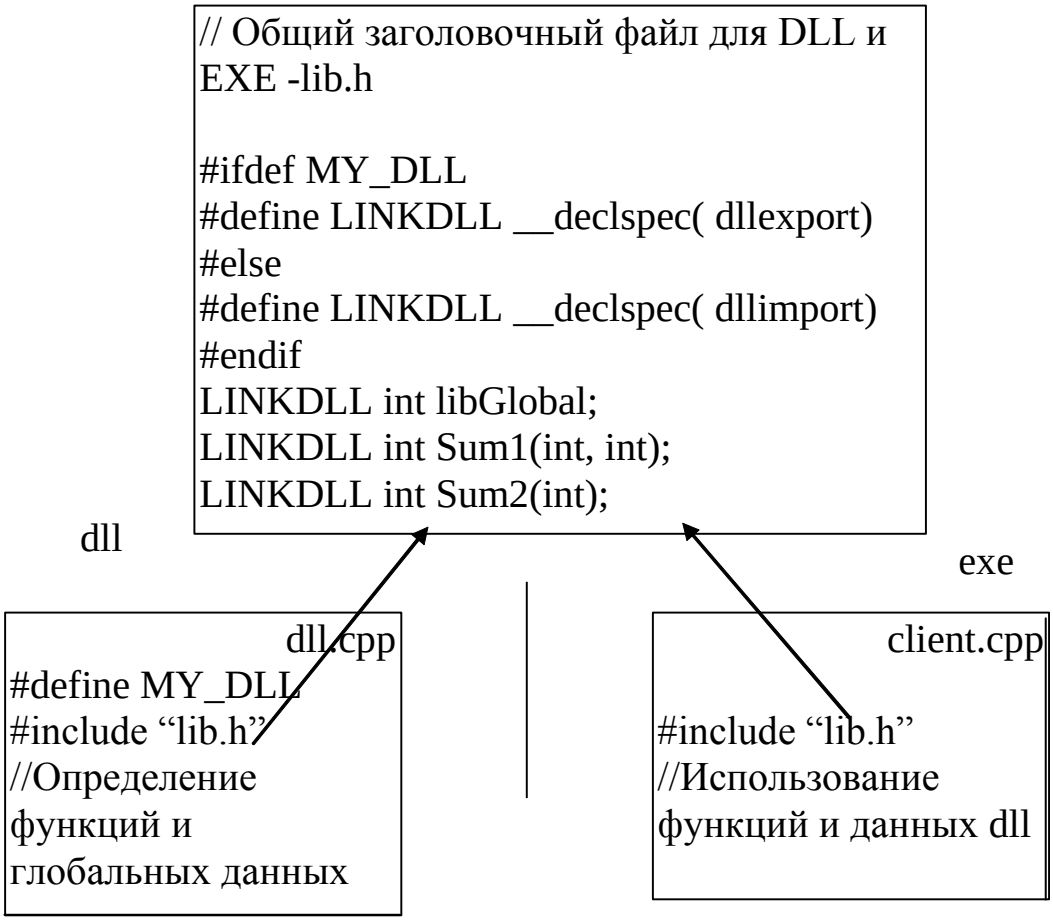

Рис. 8.12. Использование заголовочного файла dll

Замечание: обычно имя (в приведенном примере MY_DLL) делается определенным не с помощью директивы #define, а в опциях командной строки компилятору посредством ключа /D “MY_DLL”

Явное (explicit) связывание dll с клиентом

Большинство приложений использует неявное связывание, так как это самый простой способ использования динамических библиотек. Но этот способ имеет ряд недостатков:

- все dll загружаются при загрузке клиентского приложения;
- если хотя бы одна dll не найдена системой, приложение

запущено не будет.

Поэтому, несмотря на то, что реализация явного связывания гораздо сложнее, в некоторых случаях без этого способа не обойтись:

- приложение не знает заранее имени dll, а получает или формирует его в процессе выполнения;
- программные объекты dll используются крайне редко в зависимости от исключительных условий, поэтому совершенно незачем всегда тратить на них память;
- если данной dll не существует, хочется продолжить выполнение приложения с использованием других средств (альтернативный вариант выполнения приложения), но при неявном связывании в этом случае инициализация приложения будет прекращена ОС;
- в зависимости от условий нужно для одних и тех же целей в процессе выполнения загружать разные dll.

Явное (иногда называемое динамическим) связывание используется в том случае, когда Вы хотите загружать dll только непосредственно перед использованием ее функций, данных или ресурсов. После использования данную dll можно выгрузить из памяти, экономя память во время выполнения Вашей программы. При этом в случае, когда dll не найдена, программист может предусмотреть альтернативный вариант действий.

Механизм неявного связывания представлен на рис. 8.13. При явном связывании разработчик библиотеки обязан предоставить клиенту только файл с расширением .dll, а также может облегчить жизнь (это не обязательно) программисту клиентского приложения посредством заголовочного файла. Клиент не использует библиотеку импорта (.lib), так как поиск адреса импортируемого понятия осуществляется системой только по запросу клиента. И в большинстве случаев заголовочный файл с объявлениями экспортируемых библиотекой понятий тоже не нужен, так как обращение к таким данным и функциям осуществляется по-другому.

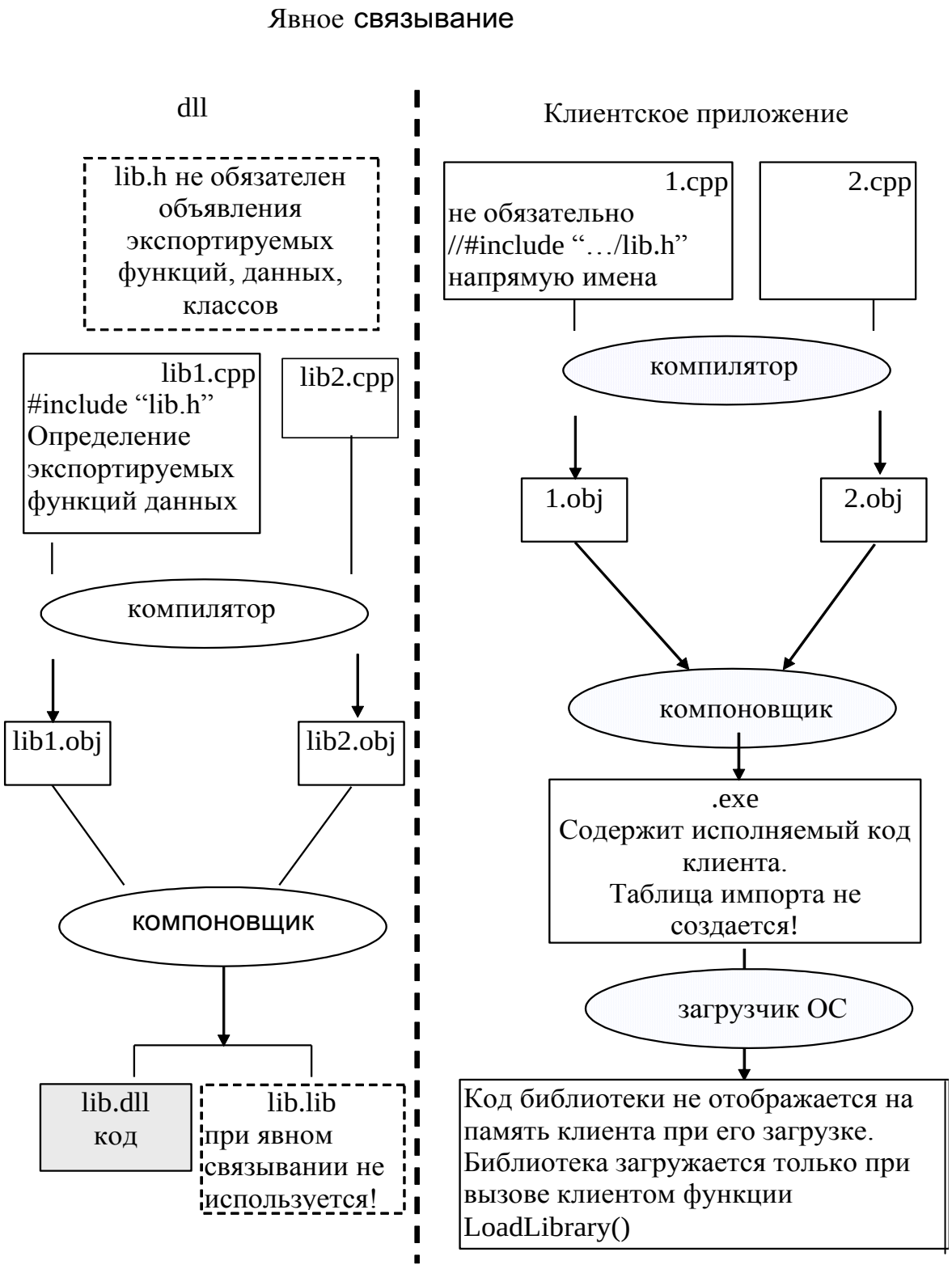

Рис. 8.13. Использование dll посредством явного связывания

Разработчик библиотеки предоставляет клиенту только файл .dll

Последовательность действий программиста для доступа к импортируемым функциям или данным выглядит следующим образом.

1. Образ dll проецируется на адресное пространство процесса явным способом, для чего клиент должен вызвать функцию LoadLibrary() или LoadLibraryEx(). Прототипы этих системных функций:

```
HMODULE LoadLibrary( //возвращаемое значение – дескриптор
                     загруженной dll (соответствует виртуальному
                     адресу отображения) или 0 в случае неудачи
LPCTSTR lpLibFileName);//спецификация dll файла.
                     Замечание: Если в строке указан путь, но
                     там исполняемого файла нет, функция
                     возвращает 0. Если только имя файла,
                     система использует ту же стратегию поиска,
                     что и при неявном связывании
```

Вторая функция обладает дополнительными возможностями.

```
HMODULE LoadLibraryEx(LPCTSTR lpLibFileName,
                HANDLE hFile,//зарезервирован и должен быть 0
                DWORD dwFlags);//либо 0, либо комбинация
                    флагов: DONT_RESOLVE_DLL_REFERENCES,
                    LOAD_LIBRARY_AS_DATAFILE или
                    LOAD_WITH_ALTERED_SEARCH_PATH
```

Флаг DONT_RESOLVE_DLL_REFERENCES указывает системе спроецировать dll на адресное пространство процесса, но не вызывать точку входа dll – функцию DllMain() и автоматически не загружать другие dll, используемые данной.

Флаг LOAD_LIBRARY_AS_DATAFILE предоставляет ОС информацию о том, что dll содержит только данные, поэтому при проецировании dll на адресное пространство процесса ОС не будет вызывать DllMain(). При этом система не тратит время на подготовку к исполнению какого-либо кода. Рекомендуется указывать этот флаг, если dll содержит только ресурсы. При этом получая возвращаемое значение, приложение - клиент может использовать его в системных функциях (типа LoadIcon()) для доступа к ресурсам.

Флаг LOAD_WITH_ALTERED_SEARCH_PATH – заставляет

дополнить стратегию поиска файла dll (в соответствии с параметрами реестра DllDirectory).

Замечания:

- если библиотека имеет расширение .dll, то в спецификации файла расширение можно опустить, любое другое расширение нужно указывать явно;
- если запрашиваемая библиотека уже загружена в страничную память (явно или неявно), то объекты исполняющей системы “file” и “file mapping” не создаются, а вызов любой из функций лишь увеличивают счетчики пользователей.

2. После того, как библиотека загружена, нужно получить из нее адрес требуемой функции. Для этого используется системная функция GetProcAddress(). Прототип функции:

```
FARPROC GetProcAddress(//возвращаемое значение – адрес данного
                       или функции, в случае неудачи - 0
                  HMODULE, // дескриптор библиотеки, полученный в
                       результате предыдущего вызова
                  LPCSTR); // экспортируемое имя функции или
                       данного
```

Замечания:

- имя функции должно соответствовать имени в таблице экспорта dll;
- импорт по номеру осуществляется этой же функцией, для приведения типа параметра используется макрос MAKEINTRESOURCE;
- для получения адреса импортируемого данного используется также функция GetProcAddress();
- тип возвращаемого значения имеет смысл указателя на функцию, которая не принимает параметром, и возвращает целое.

3. Получив адрес импортируемого понятия, программист должен объяснить компилятору, каким образом этим адресом следует пользоваться. FARPROC – это псевдоним указателя на функцию вида

```
int (FAR WINAPI * FARPROC) ()
```

Импортируемая функция может иметь совершенно другой вид, поэтому программист должен явно привести возвращаемый адрес к определенному типу (указать тип возвращаемого значения, соглашение по вызову, количество и типы принимаемых параметров).

Если возвращаемый адрес – это адрес переменной, то возвращаемое значение необходимо привести к указателю требуемого типа.

Замечание: в случае явного связывания компилятор не может проверить корректность приведения типа, поэтому ошибок он не выдаст, а при обращении могут возникнуть непредсказуемые ситуации или ошибки времени выполнения.

4. Располагая типизированным указателем, можно вызывать функцию или обращаться к данным.

5. После использования требуется отключить dll от адресного пространства клиента с помощью системной функции FreeLibrary (). Прототип:

```
BOOL FreeLibrary( //возвращаемое значение в случае неудачи 0
                HMODULE hinstDLL);//дескриптор dll
```

Эта функция декрементирует счетчики пользователей объектов “file” и “file mapping”. Если при этом счетчик пользователей объекта “file mapping” обнулились, система освободит страницы в страничной памяти, занятые образом dll.

Рассмотрим пример. Пусть динамическая библиотека экспортирует функцию и переменную.

Библиотека (с помощью extern "C" отменяем декорирование имен, поэтому экспортироваться функция и переменная будут по указанным программистом именам):

```
extern "C" __declspec(dllexport) int nGlobal; //экспорт
                    переменной
extern "C" __declspec(dllexport) int sum(int x, int y)
                    //экспорт функции
{
    return x+y;
}
```

Клиент:

```
int main()
{
    //Загружаем библиотеку
    HMODULE hDLL = LoadLibrary("MyDll.dll"); //расширение можно
                   опустить
    if(hDLL==0) {//не удалось найти dll
        return -1;
    }
    //Пользуемся экспортируемой переменной
    FARPROC pFar = GetProcAddress(hDLL,"nGlobal"); //получаем
                   адрес переменной
    if(pFar!=0) //если адрес удалось получить
    {
        int* p = reinterpret_cast< int*>( pFar);//приводим к
                 нужному типу
        *p = 55; //используем
    }
    //Пользуемся экспортируемой функцией
    pFar = GetProcAddress(hDLL,"Sum");//получаем адрес функции
    if(pFar!=0) //если адрес удалось получить
    {
        int (*pSum)(int,int) = reinterpret_cast< int
                 (*)(int,int)>(pFar); //приводим к нужному
                 типу
        int a=1, b=2;
        int c=pSum(a,b);//вызов библиотечной функции
                 посредством указателя
    }
    //Отключаем библиотеку
    FreeLibrary(hDLL);
    return 0;
}
```

Замечание: в приведенном примере системная функция GetProcAddress() в обоих случаях вернет ненулевые указатели, так как в библиотеке декорирование имен было отменено с помощью extern "C". Поэтому в массиве имен раздела экспорта библиотеки имена переменной и функции были сформированы так, как задал их в тексте

программы программист. Если бы декорирование отменено не было, то экспорт был бы осуществлен посредством внутреннего декорированного имени, и функция GetProcAddress() в таблице экспорта соответствия не обнаружила бы. Декорированное имя указывать достаточно сложно, да и правила декорирования могут различаться в разных реализациях. Поэтому у разработчика библиотеки имеются следующие возможности экспортировать имена:

- отмена декорирования с помощью extern "C";
- использование файла определений (.def) для
  - отмены декорирования,
  - экспорта под псевдонимом,
  - экспорта по номеру;
- использование директивы #pragma для тех же целей.

Пример экспорта по номеру с помощью файла определений.

Библиотека.

Файл library.cpp

```
int nGlobal;
int sum(int x, int y)
{
    return x+y;
}
```

Файл library.def

```
LIBRARY   MyDll
EXPORTS
dllGlobal=nGlobal @10 DATA
sum @11
```

Клиент:

```
int main()
{
    //Загружаем библиотеку
    HMODULE hDLL = LoadLibrary("MyDll.dll"); //расширение можно
                    опустить
    if(hDLL==0) {//не удалось найти dll
        return -1;
    }
```

```
    //Пользуемся экспортируемой переменной
    FARPROC pFar = GetProcAddress(hDLL,"nGlobal");
                   //возвращается нулевое значение, так как
                   такого экспортируемого имени нет
    pFar = GetProcAddress(hDLL,"dllGlobal"); // получаем адрес
                   переменной, так как переменная
                   экспортируется под псевдонимом
    pFar = GetProcAddress(hDLL,MAKEINTRESOURCE(10)); //получаем
                   адрес переменной, так как она
                   экспортируется по номеру


    …
    //Пользуемся экспортируемой функцией
    pFar = GetProcAddress(hDLL,"sum");//получаем адрес функции,
                   так как функция экспортируется по не
                   декорированному имени
    pFar = GetProcAddress(hDLL, MAKEINTRESOURCE(11));//получаем
                   адрес функции, так как функция
                   экспортируется по номеру
    …
    FreeLibrary(hDLL);
    return 0;
}
```

Замечание: даже если программист с помощью extern "C" отменил декорирование имен, имена функций с соглашением о вызове __stdcall хранятся в объектном и исполняемом файлах с учетом объема передаваемых параметров. А значит, и экспортируются под такими именами. Например, для функции

```
_declspec(dllexport) __stdcall int sm(int, int);
```

экспортироваться будет имя _sum@8, где 8 – это количество байтов, резервируемое компилятором для передачи параметров в стеке. Так как это соглашение достаточно популярно, рано или поздно программисту придется решать эту проблему, указывая экспортируемое имя как sum@8, или отменяя декорирование другими способами.

Точка входа динамической библиотеки

У Win32 динамической библиотеки может быть единственная «точка входа» функция DllMain(), но она не обязательна! Система

вызывает ее с разными параметрами в сугубо информационных целях и не один раз на протяжении «жизни» dll. Вызов функции DllMain() происходит в следующих случаях:

- при отображении кода dll на память процесса-клиента (в случае неявного связывания при запуске приложения, в случае явного связывания при вызове функции LoadLibrary());
- при отключении кода dll от процесса-клиента (в случае неявного связывания при завершении приложения, в случае явного связывания при вызове функции FreeLibrary());
- когда в процессе запускается новый поток (не первичный);
- когда в процессе корректно завершается любой поток.

Функция DllMain() может быть использована для инициализации/деактивации ресурсов, используемых как процессом в целом, так и любым потоком процесса. В документации по использованию системных сервисов (Platform SDK) утверждается, что функция DllMain() должна выполнять лишь простые виды инициализации. Простыми действиями разработчики ОС считают в этом случае настройку локальной памяти потока (TLS), создание объектов исполняющей системы, открытие файлов и т. д. В функции DllMain() следует избегать обращений к сервисам, связанным с User, Shell, ODBC, COM, RPC и сокетами, потому что соответствующие DLL могут быть еще не инициализированы. Кроме того, подобные функции могут в свою очередь вызывать функцию LoadLibrary() или LoadLibraryEx() и тем самым приводить к взаимной блокировке.

Замечания:

- существуют еще две функции LibMain() и WEP(), которые можно использовать в качестве точки входа dll, но они считаются устаревшими, не гарантируют безопасной работы, поэтому не рекомендованы для использования;
- так как DllMain() используется для инициализации/очистки, если эти действия не нужны, можно эту функцию не реализовывать (при этом нужно указать компоновщику ключ в командной строке

/NOENTRY).

Если функция DllMain() присутствует, то она должна иметь определенную структуру. Рассмотрим пример использования этой функции для подсчета количества процессов-клиентов, использующих данную dll.

// dll.cpp

```
    #pragma data_seg ("MyShared")
    __declspec(allocate("MyShared"))volatile LONG dll_count;
                    //попросили компоновщик поместить
                    переменную в секцию с именем MyShared


BOOL WINAPI DllMain(//возвращаемое значение: если система
                    вызывает DllMain() с параметром
                    DLL_PROCESS_ATTACH, функция должна
                    возвращать TRUE, если инициализация прошла
                    успешно, иначе FALSE. Во всех остальных
                    случаях возвращаемое значение системой
                    игнорируется
        HINSTANCE hinstDLL,//дескриптор dll. Это значение имеет
                    смысл виртуального адреса, по которому файл
                    dll проецируется на адресное пространство
                    процесса. Замечание: обычно это значение
                    далее можно использовать в функциях,
                    загружающих ресурсы (DialogBox() или
                    LoadString())
        DWORD fdwReason,//причина (уведомление), по которой
                    система вызвала эту функцию. Может
                    принимать одно из значений:
                    DLL_PROCESS_ATTACH, DLL_THREAD_ATTACH,
                    DLL_THREAD_DETACH, DLL_THREAD_DETACH
        LPVOID lpvReserved)//если fdwReason==DLL_PROCESS_ATTACH
                    ненулевое значение означает неявную
                    компоновку (при явной ==0), если
                    fdwReason==DLL_PROCESS_DETACH ненулевое
                    значение lpvReserved означает, что
                    DllMain() была вызвана при завершении
                    процесса (если в результате FreeLibrary() -
                    lpvReserved равно нулю)
{
    #pragma comment(linker, "/SECTION:MyShared,RWS");// задали
                    способ использования переменной. Ее
                    значение можно читать, изменять и разделять
                    всем экземплярам dll
    switch(fdwReason)
```

```
{
    case DLL_PROCESS_ATTACH:   //вызывается только один раз
                на протяжении жизни процесса сразу после
                того, как DLL спроецирована на его адресное
                пространство
    //здесь может быть «глобальная» инициализация,
                например, создать кучу (heap),открыть файл
                и т.д. В нашем примере будем считать
                клиентов dll
    {
        LONG all = InterlockedIncrement(&dll_count);
                //простейший механизм, который не позволяет
                более, чем одному потоку модифицировать
                разделяемую переменную
        char ar[50];
        sprintf(ar, "Одновременно используют библиотеку %d
                процессов", all);
        MessageBox(0,"Появился еще один клиент", ar,
                MB_OK);
}
        return TRUE;
    case DLL_PROCESS_DETACH:   // вызывается только один
                раз на протяжении жизни процесса, когда DLL
                отключается от адресного пространства
                процесса (выгружается)
        //Здесь может быть «глобальная» деактивация
                освободить кучу (heap), закрыть файл и т.д.
                В нашем примере декрементируем счетчик
                клиентов dll
    {
        LONG all = InterlockedDecrement(&dll_count);
        char ar[50];
        sprintf(ar, "Осталось %d процессов", all);
        MessageBox(0,"Один клиент отказался от dll", ar,
                MB_OK);
    }
        break;

    case DLL_THREAD_ATTACH: //в текущем процессе создан
                новый вторичный поток, после чего система
                вызывает DllMain() для всех DLL,
                подключенных к процессу
```

```
            //«локальная» инициализация, например, связанная с
                    TLS
            break;
        case DLL_THREAD_DETACH: //любой из потоков завершился
                    корректно (не аварийно)
            //«локальная» деактивация
    }
    return TRUE;//используется только для DLL_PROCESS_ATTACH
}
```

Поясним варианты вызова функции DllMain() с разными значениями параметра fdwReason.

Если fdwReason имеет значение DLL_PROCESS_ATTACH, то это означает, что код dll спроецирован на адресное пространство процесса при загрузке приложения или в результате первого вызова функции LoadLibrary(). При этом код функции DllMain() при неявном связывании выполняется в контексте первичного потока, а при явном – в контексте потока, который вызвал функцию LoadLibrary(). Если тот же процесс в дальнейшем еще раз вызовет функцию LoadLibrary(), то функция DllMain() будет вызвана с параметром DLL_THREAD_ATTACH в контексте соответствующего потока.

Если fdwReason имеет значение DLL_PROCESS_DETACH, то это означает, что образ dll отключается от адресного пространства клиента в результате завершения приложения или такого вызова функции FreeLibrary(), при котором счетчик пользователей обнуляется. Если процесс завершается аварийно вызовом функции TerminateProcess(), то ОС не вызывает функцию DllMain() с DLL_PROCESS_DETACH. Это означает, что ни одна из спроецированных на адресное пространство процесса dll не получит возможности осуществить деактивирующие действия.

Если fdwReason имеет значение DLL_THREAD_ATTACH, то это означает, что в процессе создан новый поток (не первичный), при этом в контексте этого потока система вызывает для каждой отображенной библиотеки функцию DllMain(), давая возможность библиотеке произвести инициализирующие действия для

создаваемого потока. Например, динамическая версия стандартной библиотеки C создает служебный блок данных, чтобы многопоточные приложения могли безопасно использовать ее функции.

Значение DLL_THREAD_DETACH формируется при вызове функции DllMain() в результате корректного завершения потока. Система просматривает все образы dll-файлов, спроецированные в данный момент на адресное пространство процесса, и в каждой из этих dll вызывает функцию DllMain() со значением DLL_THREAD_DETACH в контексте завершающегося потока. Если при отключении dll продолжают исполняться другие потоки, то для них функция DllMain() со значением DLL_THREAD_DETACH не вызывается.

Замечание: если функция DllMain() была вызвана с параметром DLL_PROCESS_ATTACH, не смогла выполнить инициализацию и вернула FALSE, то система все равно позже вызовет функцию DllMain() с параметром DLL_PROCESS_DETACH, поэтому пишите код деактивации таким образом, чтобы ненароком не уничтожать то, что и создано не было.

Действия ОС по отображению dll на адресное пространство клиента при явном и неявном связывании

На рис. 8.14 и рис. 8.15 в виде блок-схем изображена разница в последовательности действий ОС при неявном и явном связывании кода динамической библиотеки с кодом клиентского приложения.

Динамические библиотеки ресурсов

Одним из назначений динамических библиотек является хранение ресурсов. Выделение ресурсов в отдельную библиотеку преследует две цели:

- для экономии памяти в случае использования одних и тех же ресурсов несколькими приложениями;
- для упрощения замены ресурсов, если требуется их локализация.

Например, в зависимости от предпочтений пользователя один и

тот же текст требуется отображать на разных языках. Такие строки удобно хранить в качестве ресурсов в отдельных динамических библиотеках и загружать по желанию пользователя в коде приложения или предоставлять выбор ОС в зависимости от текущих региональных настроек. При явной загрузке соответствующей библиотеки программист может использовать следующие системные функции: LoadString() или более низкоуровневые системные средства – функции FindResourceEx(), LoadResource(), and LockResource().

Рис. 8.14. Этапы отображения динамической библиотеки на адресное пространство клиента при неявном связывании

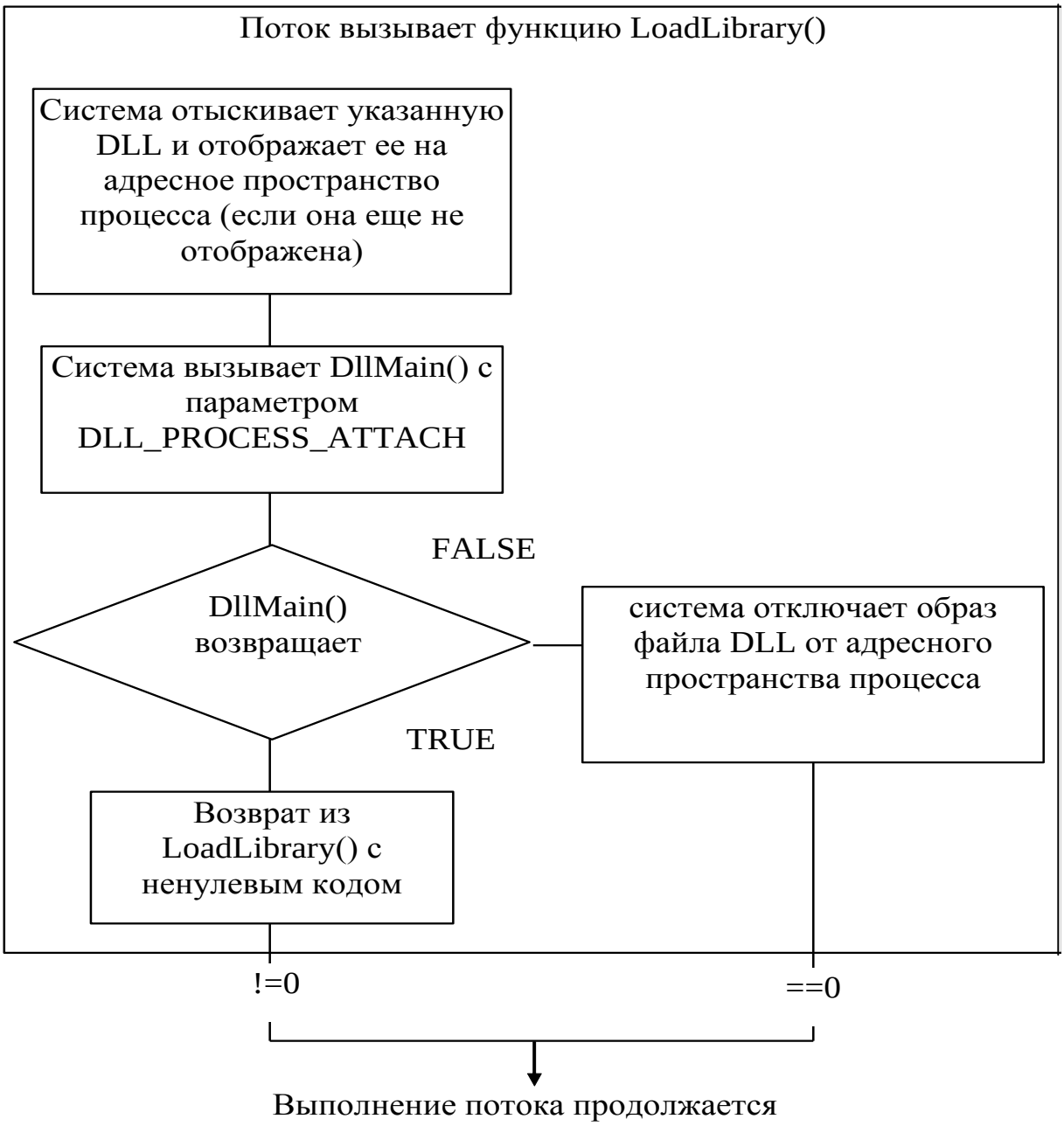

Рис. 8.15. Этапы отображения динамической библиотеки на адресное пространство клиента при явном связывании

Приведем пример получения и использования иконки из ресурсов динамической библиотеки.

```
{
    HWND hwnd = < дескриптор окна >;
    HMODULE hLibrary = LoadLibrary("имя динамической
                    библиотеки"); //загружаем библиотеку
```

```
    if(hLibrary != 0) //удалось найти соответствующий файл и
                загрузить библиотеку
    {
        HICON hIco = LoadIcon(hLibrary, MAKEINTRESOURCE(целое
                значение идентификатора иконки));
        if(hIco != 0) // если соответствующий ресурс в dll
                найден
        {
            HDC hdc = GetDC(hwnd); //получили контекст
                устройства
            DrawIcon(hdc, 0, 0, hIco); //нарисовали иконку в
                левом верхнем углу окна
            ReleaseDC(hwnd, hdc); //освободили контекст
                устройства
        }
        FreeLibrary(hLibrary); //освободили библиотеку
    }
}
```

9. ПОТОКИ

В ОС Windows приложение описывается двумя понятиями: процесс и поток. Процесс является контейнером для всех ресурсов, используемых приложением, кроме учета времени выполнения. Можно сказать, что процесс описывает статику приложения. А время для выполнения выделяется системой не процессу, а потокам, поэтому можно сказать, что поток описывает динамику приложения. Следовательно, на самом деле, все то, что мы называем межпроцессным взаимодействием, было бы корректнее назвать взаимодействием между потоками. Потоки могут выполняться как в разных процессах, так и принадлежать одному и тому же процессу. При этом следует учитывать:

- задача предоставления потоку необходимого процессорного времени возлагается исключительно на операционную систему, и ее в общем случае не интересует – относятся переключаемые потоки к одному процессу или к разным;
- для действительно многопроцессорной (многоядерной) системы система сама (та ее часть, которая называется планировщиком потоков) распределяет потоки по отдельным процессорам. В распоряжении программиста есть несколько функций, посредством которых он может предписать использование конкретных ядер для выполнения – функции SetProcessAffinityMask(), SetThreadAffinityMask() или указать свое предпочтение – функция SetThreadIdealProcessor();
- адресное пространство процесса защищено, но все потоки одного процесса находятся в едином адресном пространстве, поэтому, с одной стороны, имеют доступ ко всем ресурсам процесса, но, с другой стороны, могут одновременно требовать доступа к одним и тем же ресурсам, поэтому программист должен решать проблемы синхронизации;

- потоки разных процессов тоже могут пользоваться одними и теми же системными ресурсами (файлами, устройствами ввода/вывода), доступ к которым тоже должен быть синхронизирован;
- нельзя сделать никаких предположений относительно очередности выполнения потоков с одинаковым приоритетом, поэтому если важна очередность выполнения задач, поэтому и в этом случае необходимо решать проблемы синхронизации;
- так как у каждого потока свой стек, с локальными переменными проблем возникнуть не может, но если несколько потоков одного процесса разделяют глобальные данные, то опять требуется синхронизация доступа к этим данным или предоставление каждому потоку своей копии глобальных данных посредством механизма локальной памяти потоков (TLS - Thread Local Storage).

Синхронизация в многозадачной и многопроцессорной системе реализуется на двух уровнях:

- исполняющей системой (инициируется и используется исключительно системой);
- пользователем (инициируется пользовательским приложением, а реализуется на уровне системы).

9.1. ПОНЯТИЯ, СВЯЗАННЫЕ С ПОТОКАМИ

Перечислим термины и понятия, связанные с потоками.

Первичный и вторичные потоки

Первичный поток создается автоматически системой при запуске приложения. В первичном потоке выполняется функция main() или WinMain().

Вторичные потоки создаются системой по запросу программиста. Для создания потока приложение должно вызвать соответствующую системную функцию – CreateThread().

В обоих случаях система создает объект исполняющей системы – “thread”.

Контекст потока

Контекст потока – это набор регистров процессора, стек для работы в пользовательском режиме, стек для работы и режиме ядра и собственная (служебная) область памяти, предназначенная для использования подсистемами, библиотеками периода выполнения и DLL – рис. 9.1. Для описания всех этих данных используется недокументированная в справочной системе структура CONTEXT (ее формат можно посмотреть в заголовочном файле Winnt.h).

Рис. 9.1. Контекст потока

Замечание: контекст потока аппаратно зависим, так как регистры определяются типом процессора.

Виды потоков

Потоки бывают:

- рабочими,
- интерфейсными.

Обычно в приложении существует только один поток, отвечающий за поддержку пользовательского интерфейса и взаимодействие с пользователем. Интерфейсный поток создает окна и способен принимать и обрабатывать сообщения (содержит цикл обработки сообщений).

В рабочем потоке не принято использовать цикл обработки

сообщений и создавать окна. При создании многопоточных программ наиболее часто используются именно рабочие потоки. Дело в том, что необходимость в нескольких «каналах» приема и обработки сообщений возникает достаточно редко, в то время как во многих приложениях часто используются вспомогательные потоки, позволяющие вести фоновую обработку данных или осуществлять ввод/вывод.

Замечание: поток пользовательского интерфейса, как правило, должен иметь более высокий приоритет, чем рабочие потоки. Это нужно для того, чтобы он как можно более оперативно реагировал на действия пользователя.

9.2. СОСТОЯНИЕ ПОТОКА

Во время функционирования поток может находиться в разных состояниях.

Состояние выполнения

Потоки, находящиеся в состоянии выполнения, называются активными. Если система, точнее компонент ОС, который называется планировщиком потоков, выбрала поток для выполнения, она выделяет ему квант времени. Совершенно необязательно потоку удастся использовать весь выделенный интервал времени для выполнения. Поток будет выполняться, пока не произойдет одно из событий:

- действительно закончится квант времени;
- поток завершился, не успев израсходовать отведенный ему квант времени;
- поток перешел в состояние ожидания;
- появился другой поток с более высоким приоритетом, поэтому в системах с вытесняющей многозадачностью отдается предпочтение более высокоприоритетному потоку;
- пуста очередь сообщений, цикл обработки которой реализует

данный поток, поэтому система «отбирает» остаток кванта.

Замечания:

- только в многопроцессорных системах несколько потоков могут одновременно находиться в состоянии выполнения;
- величина кванта может быть разной для потоков с разным приоритетом и может отличаться даже в зависимости от версии одной и той же ОС, например, в версиях Professional и Server.

Состояние готовности

В состоянии готовности находятся потоки, выполнению которых не препятствуют никакие дополнительные условия, и поток может в любой момент продолжить выполнение.

Система может переводить выполняющийся поток в состояние готовности, если:

- истек квант времени;
- закончилось состояние ожидания потока (поток «проснулся»);
- активизируется поток с более высоким приоритетом.

Только что созданный поток при условии, что он не был создан в отложенном (suspended) состоянии, тоже находится в состоянии готовности.

Состояние ожидания

Состояние ожидания (простоя) означает, что поток может продолжить выполнение только при выполнении какого-нибудь условия:

- поток отправлен в спячку на определенный интервал времени;
- поток ожидает установки в сигнальное состояние одного или нескольких объектов;
- поток создан в состоянии ожидания (suspended).

Состояние завершен

Если весь код потока выполнен, то система исключает этот поток из планирования, но соответствующий объект исполняющей системы “thread” продолжает существовать, пока существует хотя бы один открытый дескриптор этого объекта. Посредством дескриптора

объекта можно получить достаточно много информации об уже завершившемся потоке, например, получить код завершения, узнать, сколько времени поток работал в пользовательском режиме или в режиме ядра. Такое состояние называется состоянием «завершен». В этом состоянии поток остается, пока не будет закрыт последний дескриптор объекта «thread».

9.3. ПЕРЕКЛЮЧЕНИЕ ПОТОКОВ

При любых изменениях, связанных с выполнением потока, «вмешивается» система (диспетчер потоков). Это происходит в следующих случаях:

- когда поток переходит из одного состояния в другое;
- при изменении приоритета потока;
- по каким-то причинам системе понадобилось продолжить выполнение потока на другом процессоре (ядре).

Приоритет потока может измениться:

- явно при вызове программистом системной функции SetThreadPriority();
- неявно самой системой.

В любой ситуации, когда речь идет о переключении потоков (истек квант времени, поток отправлен «в спячку» и т. д.) система должна решить: какой поток должен выполняться следующим. Когда выбор сделан, система «переключает контекст». Поясним действия системы при переключении контекста:

- сохраняет контекст выполняемого потока,
- загружает контекст выбранного потока;
- передает управление выбранному потоку.

Для того чтобы поток мог выполняться, ОС должна выделить ему время. Следует отметить, что в разных системах алгоритмы выделения квантов времени потокам – разные. Но выделение времени всегда:

- связано с понятием активности потока;
- зависит от приоритета потока.

Время выделяется только активным потокам. То есть, если поток находится в состоянии ожидания, то и времени ему не выделяется, он исключается из диспетчирования.

Выделение времени всегда связано с понятием приоритета потока. Если существуют несколько потоков с одинаковым приоритетом, то время выделяется тому, кто дольше всех ждал. Алгоритмы распределения времени между потоками с разными приоритетами отличаются в разных системах. Но в любом случае предпочтение отдается потоку с более высоким приоритетом. В ОС семейства NT это предпочтение выражается в различных значениях выделяемого кванта, что дает низкоприоритетным потокам возможность для выполнения.

Замечание: сама система, оказывается, тоже интенсивно использует многопоточность, например, нулевой уровень приоритета присваивается в системе особому потоку обнуления страниц, обнуляющему свободные страницы при отсутствии других потоков, требующих внимания со стороны ОС. Ни один поток, кроме него, не может иметь нулевой уровень. Есть еще специальный поток для приема информации от мыши и клавиатуры, дальнейшего преобразования этой информации в сообщения и распределения их по очередям приложений.

9.4. ПОТОКОВАЯ ФУНКЦИЯ

Каждый поток начинает выполнение с некоей входной функции. В первичном потоке таковой является стартовый код, предоставляемый стандартной библиотекой, из которого в свою очередь вызывается функция main() или WinMain(). Если программист хочет создать вторичный поток, он в первую очередь должен предоставить точку входа - потоковую функцию. Прототип

потоковой функции:

```
DWORD WINAPI ThreadFunc( //возвращаемое значение – это код
                         завершения потока
                PVOID p); //смысл параметра определяется
                    программистом, то есть все, что угодно,
                    помещающееся в 4 байта
```

Несмотря на название, это не обычная функция. Специфика потоковой функции:

- в отличие от псевдо точки входа первичного потока, у которой должно быть одно из предопределенных имен: main, wmain, WinMain или wWinMain, функцию потока можно назвать как угодно;
- у функций main(), WinMain() и т. д. предопределены не только имена, но и количество, а также типы параметров, у потоковой функции должен быть единственный четырехбайтовый параметр, смысл которого определяется программистом;
- порядок вызова обычных функций определяет программист, потоковую функцию запускает система.

Сравним вызов обычной функции и запуск потока – табл. 9.1.

Таблица 9.1

Отличия вызова функции и запуска потока

Вызов обычной функции	Запуск потока
Вызывающая функция и вызываемая пользуются одним и тем же стеком.	Стек у каждого потока (порождающего и порожденного) свой
Последовательность выполнения вызывающей и вызываемой функций предопределена и гарантируется компилятором.	Последовательность выполнения порождающего и порожденного потоков определяется системой, они относительно независимы.

Окончание табл. 9.1

Вызов обычной функции	Запуск потока
Параметры передаются в том же стеке, поэтому компилятор гарантирует, что локальные данные вызывающей функции существуют, пока выполняется вызываемая. Поэтому безопасно передавать адреса локальных переменных.	Параметр (один) формируется в стеке потоковой функции. Данные, передаваемые потоковой функции должны гарантированно существовать, пока ими пользуется порожденный поток. Если в качестве параметра передается адрес локальных для порождающего потока данных, то программист должен обеспечить существование этих данных, пока ими пользуется потоковая функция.

На рис. 9.2 схематически представлен запуск потока.

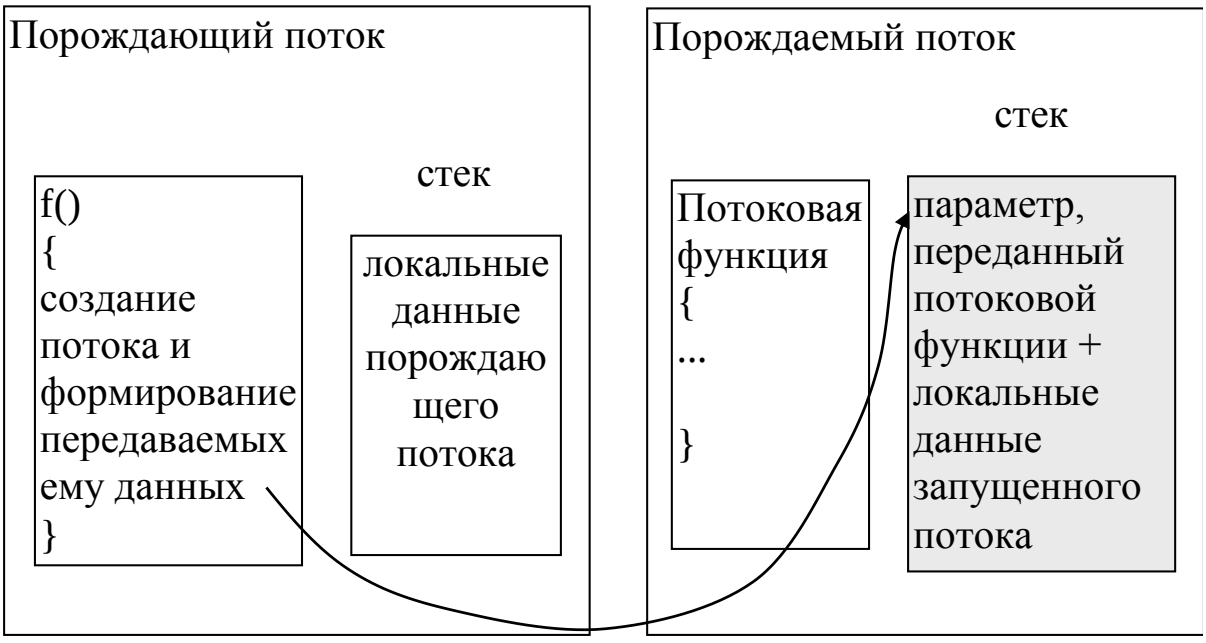

Рис. 9.2. Передача данных потоку

Нормальное выполнение потока продолжается до тех пор, пока не завершится его потоковая функция. Завершение приложения (естественное) происходит при завершении всех потоков.

Функция потока должна возвращать код завершения потока. Код завершения первичного потока становится кодом завершения процесса. Код завершения вторичного потока можно получить (располагая дескриптором потока) с помощью функции GetExitCodeThread().

Рекомендация: если только речь не идет об использовании общих данных, функции потоков должны, по мере возможности, обходиться локальными переменными, так как они создаются в индивидуальном стеке каждого потока и никаким образом не конфликтуют.

9.5. СОЗДАНИЕ ПОТОКА

Создавать потоки можно с помощью системной функции CreateThread() или средствами стандартной библиотеки, если программист пишет приложение на языке C/C++.

Системная функция CreateThread()

Системная функция CreateThread() создает в адресном пространстве процесса вторичный поток для выполнения. При создании потоков в Win32 также как и при порождении процессов, не существует отношений родительский/дочерний поток. Прототип функции:

```
HANDLE CreateThread(//возвращаемое значение==0 - ошибка
  LPSECURITY_ATTRIBUTES lpsa, // если 0- защита по умолчанию и
                    дочерние процессы не наследуют данный поток
  SIZE_T cbStack, // размер стека или 0 (в случае 0
                    используется информация, встроенная
                    компоновщиком в EXE-файл)
  LPTHREAD_START_ROUTINE lpStartAddr, // указатель на
                    потоковую функцию (точку входа) потока. Не
                    может быть нулевым! С этой функции начинает
                    свое выполнение запущенный поток.
  LPVOID lpvThreadParm, // значение или указатель на любые
                    данные. Используется по замыслу
```

```
                    программиста. Эти данные должны
                    гарантированно существовать пока их
                    использует потоковая функция, так как стек
                    у каждого потока свой!
  DWORD fdwCreate, // дополнительные флаги создания:
                    0 (создается в состоянии готовности,
                    поэтому сразу же включается системой в
                    планирование)
                    CREATE_SUSPENDED - поток создается
                    "остановленным", то есть ожидающим запуска.
                    Для того чтобы запустить такой поток, кто-
                    то должен вызвать функцию ResumeThread()
  LPDWORD lpIDThread// сюда функция вернет идентификатор (в
                    NT, если программисту не нужно значение,
                    можно передать 0)
);
```

Замечание: для того, чтобы создать поток в адресном пространстве другого процесса, следует воспользоваться функцией CreateRemoteThread().

Что происходит при вызове функции CreateThread():

- создается объект исполняющей системы «thread»;
- счетчик пользователей объекта устанавливается в 1;
- в таблице дескрипторов процесса создается соответствующая строка;
- система создает и инициализирует для нового потока структуру CONTEXT;
- создается стек потока;
- значения параметров lpStartAddr и lpvThreadParm заносятся на самый верх стека;
- если только поток не создавался в приостановленном (CREATE_SUSPENDED) состоянии, поток готов к выполнению. Это не означает, что система сразу же разрешит ему выполняться. Поток будет просто учтен в планировании. А выполняться начнет только тогда, когда система выберет его на выполнение;
- перед возвратом из функции счетчик пользователей объекта "thread" увеличивается еще на единицу и в таблице дескрипторов процесса создается еще одна запись для того же объекта,

соответственно, в распоряжение порождающего потока система предоставляет еще один открытый дескриптор, посредством которого порождающий поток может взаимодействовать с порожденным.

Рассмотрим пример создания потока посредством функции CreateThread(). Следует помнить о том, что порождающий и порожденный потоки относительно независимы, поэтому не исключена ситуация, связанная с передачей адресов локальных данных, при которой:

- порождающий поток завершится раньше порожденного;
- та функция, в стековом кадре которой сформированы локальные данные для передачи потоку, закончится раньше, чем порожденный поток сможет этими данными воспользоваться (при этом область стека порождающего потока может быть использована для локальных переменных другой функции).

С учетом вышесказанного следует обратить внимание на то, каким образом программист может обеспечить существование данных, передаваемых потоковой функции.

```
//Потоковая функция в качестве параметра принимает адрес
                    строки для отображения
DWORD WINAPI ThreadFunc( LPVOID lpParam )
{
    MessageBox(0,(const char*) lpParam,”Thread Message Box”,
                    MB_OK);
    return 0;
}
//порождение вторичного потока
{
    DWORD Id; //сюда будет возвращен идентификатор созданного
                    потока
    HANDLE h = CreateThread(
    NULL,                           // безопасность по умолчанию
    0,                              // размер стека по умолчанию
    ThreadFunc,                     // указатель на потоковую
                    функцию
    “QWERTY”,                  // в качестве параметра передать
                    потоковой функции адрес строкового
```

```
                    литерала. Память под этот массив символов
                    будет выделена не в стеке, а в области
                    кода, поэтому строка будет гарантированно
                    существовать до завершения приложения
    0,                          // по умолчанию поток
                    создается в состоянии готовности
    &Id);                 // адрес переиенной для формирования
                    идентификатора потока
    if (h == 0) //если поток создать не удалось, выдать
                    диагностику
        MessageBox( 0,"Fail","Primary Thread", MB_OK);
    else // сказать заранее, когда поток получит процессорное
                    время, нельзя, но планировщик уже учитывает
                    его
    CloseHandle( h ); //если взаимодействовать с порожденным
                    потоком не требуется, то дескриптор следует
                    закрыть
}
```

Способы обеспечения существования данных, адреса которых передаются порождаемому потоку:

• глобальные и статические данные существуют до завершения приложения, поэтому адреса таких данных передавать безопасно;

• порождающий поток может создать для порождаемого данные динамически, при этом освобождение памяти после использования данных возлагается на порожденный поток;

• если порождающий поток передает порождаемому адрес своих локальных данных, то единственной гарантией существования таких данных является ожидание порождающим потоком завершения порожденного (например, с помощью функции ожидания WaitForSingleObject()).

Потоки и стандартная библиотека Си

Если программист пишет многопоточное приложение на языке C/C++, он должен учитывать особенности реализации стандартной библиотеки. Так как стандартная библиотека Си создавалась в те времена, когда такого понятия как многопоточность еще не было, то исходно она не была рассчитана на использование одновременно несколькими потоками. А именно в стандартной библиотеке

присутствуют:

- глобальные переменные, например, errno;
- нереентерабельные функции, например, malloc().

Так как система «ничего» не знает о проблемах стандартной библиотеки, в некоторых случаях для создания потоков использовать системную функцию CreateThread() небезопасно

Для обеспечения безопасности:

- за каждым потоком требуется закрепить свою копию глобальных переменных стандартной библиотеки (например, у каждого потока должная быть своя переменная errno). Кроме того, понадобится какой-то механизм, который позволит каждому потоку ссылаться на свою копию и не трогать чужую;
- доступ к общим данным (например, выделение памяти в куче) должен быть синхронизирован;
- требуется обеспечить буфер между вызовом системной функции CreateThread() и собственно началом выполнения потоковой функции для того, чтобы дать возможность стандартной библиотеке подготовить для порожденного потока копию своих глобальных данных, а также осуществить другие инициализирующие действия.

Многопоточная версия стандартной библиотеки «обертывает» нереентерабельные функции стандартной библиотеки синхронизирующими примитивами. Таким образом, например, гарантируется целостность кучи (heap) при одновременной попытке двух и более потоков одного процесса динамически выделить память.

Запуск потока осуществляется не системной функцией CreateThread(), а функцией стандартной библиотеки _begmthreadex (), при вызове которой:

- для запускаемого потока создается внутренняя служебная структура данных посредством механизма локальной памяти потока (TLS), в которой формируются копии глобальных переменных;
- в эту служебную структуру данных также заносятся адрес потоковой функции и передаваемый параметр:

• вызывается системная функция CreateThread(), но в качестве указателя на потоковую функцию ей передается адрес другой функции стандартной библиотеки _threadstart(), а в качестве передаваемого данного – адрес сформированной служебной области памяти потока;

• порожденный поток начинает свое выполнение с функции _threadstart(), которая сначала осуществляет все инициализирующие действия, а только потом потоковую функцию, передавая ей предназначенный параметр;

• после завершения потоковой функции управление снова возвращается функции _threadstart(), которая в свою очередь вызывает функцию стандартной библиотеки _endthreadex(), передавая ей код завершения потоковой функции;

• функция стандартной библиотеки _endthreadex()осуществляет деактивирующие действия (в частности освобождение локальной памяти потока) и вызывает системную функцию ExitThread(), передавая ей код завершения потока.

Смысл параметров функции _beginthreadex() такой же, как и у системной функции CreateThread(), но типы параметров вводятся посредством базовых типов языка Си, а не псевдонимов, введенных разработчиками Windows. Прототип функции:

```
uintptr_t _beginthreadex (возвращаемое значение: 0 – неудача,
                    ненулевое значение – дескриптор созданного
                    объекта
                    void *Security, // указатель на дескриптор
                    защиты
                    unsigned StackSize, // размер стека или 0
                    unsigned (__stdcall *StartAddress)(void*),
                    // адрес потоковой функции
                    void *ArgList, // данные для потоковой функции
                    unsigned Initflag, // параметры создания
                    потока 0 или CREATE_SUSPENDED
                    unsigned *ThrdAddr); // указатель на
                    переменную для идентификатора потока или 0
```

Пример создания потока с помощью функции _beginthreadex():

```
#include <process.h>
```

```
//Потоковая функция
uintptr_t __stdcall ThreadFuncEx( void* p )

{
    MessageBox(0,(char*) p,"BeginThreadEx Message Box", MB_OK);
    return 0;
}
//Порождающий поток
int main()
{
    //Запуск потока средствами стандартной библиотеки
    unsigned int id;
    HANDLE hThreadEx =
                    (HANDLE)_beginthreadex(0,1024,ThreadFuncEx,
        "Proba",0,&id);
    if(hThreadEx)
    {
        CloseHandle(hThreadEx);
    }
    WaitForSingleObject(hThreadEx,INFINITE);
}
```

Замечание: для поддержки многопоточности стандартной библиотекой используются две версии стандартной библиотеки: для статической компоновки объектная библиотека LIBCMT.LIB, для динамического подключения библиотека импорта MSVCRT.LIB и собственно динамическая библиотека MSVCRT**.dll.

9.6. ЗАВЕРШЕНИЕ ПОТОКА

Выполнение потока заканчивается в следующих случаях:

- при возврате управления из потоковой функции;
- при вызове из кода потока системной функции ExitThread() или функции стандартной библиотеки _endthreadex();
- при завершении потока с помощью системной функции TerminateThread();

• при завершении процесса с помощью системных функций ExitProcess() или TerminateProcrss().

Естественное завершение потока при возврате управления из потоковой функции гарантирует не только корректное освобождение всех ресурсов (это делает система), но и вызов деструкторов для всех объектов, созданных потоком, если приложение написано на C++ (это делает компилятор). При любом завершении потока система:

• освобождает память, занятую стеком данного потока;

• закрываются дескрипторы объектов User (окон и программных ловушек), которыми владеет поток;

• заменяет код завершения данного потока со STILL_ACTIVE на возвращаемое потоковой функцией значение;

• объект «thread» переводится в свободное (сигнальное) состояние:

• счетчик пользователей объекта «thread» уменьшается на единицу и, если открытых дескрипторов больше нет, память, занятая объектом, освобождается;

• если данный поток является последним активным потоком в процессе, завершается и сам процесс.

Замечание: при вызове из любого другого потока системной функции TerminateThread() для аварийно завершаемого потока не будут произведены все корректные завершающие действия: обработчики завершения, предусмотренные программистом, стек не будет освобожден и подключенные к процессу динамические библиотеки не получат уведомления о завершении потока.

9.7. ЗАДЕРЖКА И ВОЗОБНОВЛЕНИЕ ПОТОКОВ

Выполнение любого потока можно «приостановить», то есть исключить его из планирования:

• непосредственно при создании (с помощью параметров функций CreateProcess() или CreateThread());

• при выполнении;

• до возникновения указанного события.

Создание «отложенного» потока

Каждый объект «thread» имеет счетчик приостановок (простоев) и может выполняться только при нулевом значении этого счетчика.

В предыдущем разделе уже упоминалось о том, что создаваемый поток можно «попридержать», передав в функции CreateProcess() или CreateThread() флаг CREATE_SUSPENDED. В этом случае система создает объект исполняющей системы, формирует стек потока и инициализирует контекст (то есть элементы структуры CONTEXT). Однако счетчику простоев объекта «поток» присваивается начальное значение 1 (а не 0), а это значит, что система не собирается подключать поток к процессору для выполнения. Чтобы такой поток начал работать, нужно из другого потока вызвать системную функцию ResumeThread().

Системные функции SuspendThread() и ResumeThread()

Располагая дескриптором объекта “thread”, выполнение отдельного потока можно приостановить с помощью:

```
DWORD SuspendThread(HANDLE hThread); // возвращает предыдущее
                    значение счетчика простоев
```

Как только для потока вызвана эта функция, система отправляет поток «в спячку», то есть исключает его из планирования. В этом состоянии поток будет находиться, пока кто-нибудь не вызовет для него функцию ResumeThread(). Прототип функции:

```
DWORD ResumeThread(HANDLE hThread);//возвращает предыдущее
                    значение счетчика простоев
```

Специфика использования функций:

• если поток был «n»-раз задержан, он должен быть «n»-раз возобновлен, только тогда система выделит ему процессорное время;

• любой поток, обладающий открытым дескриптором «усыпляемого» потока, может вызвать функцию SuspendThread() и приостановить выполнение указанного в качестве параметра (в частности сам поток тоже может приостановить свое выполнение);

• возобновить сам себя поток не может.

Системные функции Sleep() и SleepEx()

Системная функция Sleep() приостанавливает выполнение вызвавшего ее потока на заданное число миллисекунд. Прототип функции:

```
void Sleep(DWORD dwMilliseconds); //интервал времени, в
                течение которого поток будет исключен из
                планирования
```

Специфика использования функции Sleep():

• вызывая функцию Sleep(), поток добровольно отказывается от остатка выделенного ему кванта времени;

• система прекращает выделять потоку процессорное время на период, равный или больший, заданного интервала, так как Windows не является системой реального времени. Уснувший поток будет только включен в планирование по истечении указанного периода. А выполняться такой поток продолжит тогда, когда ОС сочтет нужным предоставить ему квант времени;

• программист может вызвать функцию Sleep() и передать в качестве параметра значение INFINITE, вообще исключает поток из планирования, но это не очень практично (лучше корректно завершить поток, освободив его стек и объект);

• можно вызвать функцию Sleep() и передать в dwMilliseconds нулевое значение. Это будет означать, что поток сам отказывается от остатка своего кванта времени и заставит систему подключить к процессору другой ожидающий поток. Однако система может снова запустить прерванный поток, если других планируемых потоков с тем же приоритетом нет.

Пример использования функции Sleep(). Пока пользователь не нажмет любую клавишу, порожденный будет выдавать сигнал с указанным интервалом времени.

```
//Потоковая функция
DWORD WINAPI BeepFunc(LPVOID p)
{
    while(true)
```

```
    {
        MessageBeep(-1); // выдаем звуковой сигнал
        Sleep(1000); //на секунду усыпляем поток
    }
    return 0;
}

int main()
{
    HANDLE hThread = CreateThread(0,0,BeepFunc, 0,0,0);
                    //безопасность – по умолчанию, стек – по
                    умолчанию, параметр потоковой функции 0,
                    создаваемый поток готов к выполнению,
                    идентификатор не нужен
    if(hThread) //если удалось создать поток
    {
        _getch(); //ждем нажатия любой клавиши
        TerminateThread(hThread, 0); //завершаем порожденный
                    поток
        CloseHandle(hThread); // закрываем дескриптор
    }
    return 0;
}
```

Функция SleepEx() обладает по отношению к функции Sleep() дополнительными возможностями. Прототип функции:

```
DWORD WINAPI SleepEx(
    DWORD dwMilliseconds, //смысл параметра такой же, как в
                    функции Sleep()
    BOOL bAlertable //если параметр FALSE, то функция ведет
                    себя также как Sleep(), все асинхронные
                    вызовы процедур игнорируются.
                    при значении параметра TRUE поток
                    пробуждается в следующих случаях: истек
                    указанный интервал времени, или в том же
                    потоке завершилась операция расширенного
                    ввода/вывода, или произошел асинхронный
                    вызов процедуры.
);
```

Возвращаемое значение равно нулю, если истек указанный интервал времени.

10. ПРИОРИТЕТЫ ПРОЦЕССОВ И ПОТОКОВ В WIN32

Уровни приоритета процессам и потокам присваивает система, исходя из пожеланий программиста и прав пользователя. Уровни задаются в диапазоне от 0 (самый низший приоритет) до 31 (самый высший). Значения приоритетов формируются в два этапа:

- на первом этапе процессу присваивается определенный класс приоритета – абсолютный приоритет, который говорит системе, какой приоритет нужен данному процессу по сравнению с другими выполняемыми процессами;
- на втором этапе потокам, принадлежащим этому процессу, присваиваются относительные уровни приоритета (относительно приоритета процесса).

10.1. КЛАССЫ ПРИОРИТЕТА ПРОЦЕССОВ

Класс приоритета процесса система формирует, исходя из прав пользователя. В некоторой степени диспетчер задач (Task Manager) позволяет пользователю изменять класс приоритета выполняющегося процесса (вкладка «Processes», колонка «Base Priority»). Пользователь может изменить приоритет, выбрав процесс и указав другой класс в подменю «Set Priority» контекстного меню. Удастся ли действительно повысить приоритет, зависит от прав пользователя.

Win32 поддерживает 6 классов приоритетов, перечисленных в табл. 10.1. Приведенные константы можно указать в качестве параметра функции CreateProcess() при создании процесса.

Замечание: соответствующая константа – это не абсолютное значение приоритета, а флаг, который позиционно задает значение приоритета, поэтому указание произвольного значения, например 7, будет иметь непредсказуемые последствия!

Таблица 10.1

Классы приоритета процесса

Класс приоритета	Флаг в CreateProcess()	Уровень	Описание
Idle (простаива-ющий)	IDLE_PRIORITY_CLASS 0x40	4	Потоки такого процесса выполняются только когда системе больше нечего делать и вытесняются потоками с остальными приоритетами. Приоритет idle идеален для приложений, занимающихся мониторингом системы.
Normal (обычный)	NORMAL_PRIORITY_CLASS 0x20	8	Большинство приложений выполняются с таким приоритетом.
High (высокий)	HIGH_PRIORITY_CLASS 0x80	13	Процесс выполняет критичные ко времени операции. Потоки такого процесса вытесняют потоки normal-или idle. Следует обращаться с большой осторожностью – такой процесс может «забить» все время процессора. Например, с таким приоритетом выполняется диспетчер задач для того, чтобы у него была возможность принудительно (функцией TerminateProcess()) завершить «зависший» процесс.

Продолжение табл. 10.1

Класс приоритета	Флаг в CreateProcess()	Уровень	Описание
			Замечание: высокий класс приоритета вовсе не означает, что такое приложение использует все процессорное время. Большую часть времени диспетчер задач простаивает, не тратя времени. Зато, когда пользователь начинает взаимодействовать с диспетчером задач, система немедленно предоставляет ему время для выполнения.
Realtime (реального времени)	REALTIME_PRIORITY_CLASS 0x100	24	Наивысший возможный приоритет. Потоки такого процесса вытесняют все остальные, включая потоки ОС, выполняющие важные системные задачи. Такой поток должен быть очень коротким, иначе мышь перестанет отвечать или начнутся неприятности с диском. Классом приоритета realtime почти никогда пользоваться не стоит.

Окончание табл. 10.1

Класс приоритета	Флаг в CreateProcess()	Уровень	Описание
			Класс приоритета realtime используют только в программах, напрямую обращающихся к оборудованию, или если требуется выполнить быстротечную операцию, которую нельзя прервать ни при каких обстоятельствах.
	ABOVE_NORMAL_PRIORITY_CLASS 0x8000		Windows 2000/XP - выше NORMAL_PRIORITY_CLASS, но ниже HIGH_PRIORITY_CLASS
	BELOW_NORMAL_PRIORITY_CLASS 0x4000		Windows 2000/XP: выше IDLE_PRIORITY_CLASS, но ниже NORMAL_PRIORITY_CLASS

Предостережение: будьте осторожны при выборе конкретного класса приоритета! Вызывая функцию CreateProcess(), большинство приложений должны либо использовать флаг NORMAL_PRIORITY_CLASS, либо вообще не указывать класс приоритета. В последнем случае система обычно присваивает процессу класс normal, и только в том случае, когда родительский процесс имеет класс idle или below normal, дочерний процесс получит такой же класс приоритета в наследство.

Понятие активного процесса

На одном процессоре в каждый момент времени может

выполняться только один поток. Процесс, чей поток выполняется, считается активным процессом (foreground), а остальные процессы – фоновыми (background).

Замечание: процессы с обычным классом приоритета ведут себя несколько иначе, чем процессы остальных классов. В Windows NT, когда обычный процесс становится активным, система увеличивает кванты времени, выделяемые всем его потокам. Причина изменений приоритета в активных процессах очень проста: система добивается того, чтобы они быстрее реагировали на пользовательский ввод. Рассмотрим пример: активный процесс – редактирование текста, фоновый – печать. Если бы приоритеты потоков не менялись, то два normal-процесса (как фоновый, так и активный) – оба одинаково конкурировали бы за процессорное время, и пользователь, набирая текст в активном приложении, заметил бы, что текст появляется на экране какими-то рывками. Но так как система повышает уровни приоритета потоков активного процесса, они «вытесняют» потоки фонового процесса.

Изменение класса приоритета процесса

Если при запуске дочернего процесса родительский процесс может явно задать приоритет дочернего, то при запуске процесса посредством какой-нибудь утилиты (например, проводника), которая будет в данном случае играть роль родительского процесса, дочерний процесс создается с обычным (normal) приоритетом. Но запускающая утилита ведь не знает, что делает запущенный процесс, и как часто надо его потокам выделять процессорное время. Поэтому в системе предусмотрена возможность изменения класса приоритета самим выполняемым процессом с помощью функции:

```
BOOL SetPriorityClass(HANDLE hProcess, DWORD fdwPriority);
```

где параметр fdwPriority теоретически (с точностью до прав пользователя) может принимать значения IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS,
ABOVE_NORMAL_PRIORITY_CLASS,
BELOW_NORMAL_PRIORITY_CLASS HIGH_PRIORITY_CLASS и

REALTIME_PRIORITY_CLASS.

Узнать приоритет процесса можно с помощью функции:

```
DWORD GetPriorityClass(HANDLE hProcess); //возвращаемое
                значение – это одна из указанных в таблице
                констант
```

Замечание: система далеко не всегда позволит приложению изменить приоритет своего процесса на желаемый. Например, для того, чтобы установить приоритет HIGH_PRIORITY_CLASS, пользователь должен запускать приложение с правами Администратора или Power User.

10.2. ОТНОСИТЕЛЬНЫЙ ПРИОРИТЕТ ПОТОКА

Сформировав абсолютный приоритет процесса, система назначает потокам этого процесса относительные приоритеты. Обратите внимание на то, что при создании потока его приоритет нигде не указывается. Это означает, что в момент создания потока значение его приоритета соответствует классу приоритета процесса. Например, для обычного процесса с классом приоритета NORMAL_PRIORITY_CLASS поток создается с абсолютным приоритетом 8, а для процесса с HIGH_PRIORITY_CLASS - 13.

Далее уровень приоритета отдельного потока можно и повысить, и понизить функцией:

```
BOOL SetThreadPriority(
        HANDLE hThread, //дескриптор потока
         int nPriority); //относительное значение
                приоритета
```

Перечень констант, соответствующих относительным приоритетам потоков, приведен в табл. 10.2. Следует знать, что приведенные константы – это уже не битовые маски, а значения: 0, -1, +1, -2, +2.

Таблица 10.2

Относительные приоритеты потока

Значение относительного приоритета	Описание
THREAD_PRIORITY_ LOWEST (-2)	Приоритет потока должен быть на 2 единицы ниже класса приоритета процесса
THREAD_PRIORITY_BELOW_ NORMAL (-1)	Приоритет потока должен быть на 1 единицу ниже класса приоритета процесса
THREAD_PRIORITY_ NORMAL (0)	Приоритет потока должен соответствовать классу приоритета процесса
THREAD_PRIORITY_ABOVE_ NORMAL (+1)	Приоритет потока должен быть на 1 единицу выше класса приоритета процесса
THREAD_PRIORITY_HIGHEST (+2)	Приоритет потока должен быть на 2 единицы выше класса приоритета процесса
THREAD_PRIORITY_TIME_ CRITICAL (15 или 31)	Приоритет потока =15, если приоритет процесса – idle, normal, high; и =31, если приоритет процесса – realtime
THREAD_PRIORITY_IDLE (1 или 16)	Приоритет потока =1, если приоритет процесса – idle, normal, high; и =16, если приоритет процесса – realtime

В табл. 10.3 приведены значения уровней приоритетов потоков в зависимости от приоритета процесса.

Таблица 10.3

Значения приоритетов потока в зависимости от приоритета процесса

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

Замечания:

- задать уровень приоритета 0 невозможно. Это связано с тем, что нулевой приоритет зарезервирован системой для потока обнуления страниц, и никакой другой поток не может иметь такой приоритет;
- уровни 17-21 и 27-30 пользовательским приложениям тоже недоступны. Программист может пользоваться ими, только если пишет драйвер устройства, работающий в режиме ядра;
- уровень приоритета потока в процессе с классом real-time не может опускаться ниже 16;
- уровень приоритета потока в процессе с любым другим классом не может подниматься выше 15.

Пример задания приоритета потока

Так как при создании потока приоритет не указывается, то по умолчанию поток создается с приоритетом, равным приоритету процесса. Чтобы повысить или понизить приоритет потока, можно создать поток в приостановленном состоянии (установив флаг CREATE_SUSPENDED), изменить его приоритет на желаемый

(функцией SetThreadPriority()), а затем перевести его в состояние готовности (функцией Resume Thread()):

```
HANDLE hThread = CreateThread(NULL, 0, ThreadFunc, NULL,
                 CREATE_SUSPENDED, 0);
SetThreadPriority(hThread, THREAD_PRIORITY_IDLE);
ResumeThread(hThread);
CloseHandle(hThread);
```

Сказать заранее, когда поток получит процессорное время, нельзя, но планировщик уже учитывает его новый приоритет.

Динамическое изменение уровней приоритетов потока

Уровень приоритета, получаемый комбинацией относительного приоритета потока и класса приоритета процесса, которому принадлежит данный поток, называется базовым уровнем приоритета потока.

Иногда система изменяет уровень приоритета потока, обычно это бывает связано с потоками, которые обрабатывают пользовательский ввод. Этот новый уровень приоритета, временно установленный системой, называется динамическим приоритетом потока. Нижней границей динамического приоритета является базовым приоритетом потока, а верхнюю границу определяет исполняющая система, исходя из вида работы, выполняемой потоком. На рис. 10.1 представлен диапазон изменения динамического приоритета потока для процесса с приоритетом IDLE_PRIORITY_CLASS.

Например, пользователь нажимает какую-нибудь клавишу. В ответ на действие пользователя:

- система помещает в очередь потока сообщение;
- поток получает процессорное время для обработки сообщения;
- при этом система временно повышает уровень его приоритета на несколько единиц;
- процессор исполняет поток в течение отведенного кванта, а по его истечении система снижает приоритет потока на 1;
- далее потоку вновь выделяется квант времени, по окончании

которого система опять снижает уровень приоритета на 1;

• и так до тех пор, пока динамический уровень приоритета не опустится до базового.

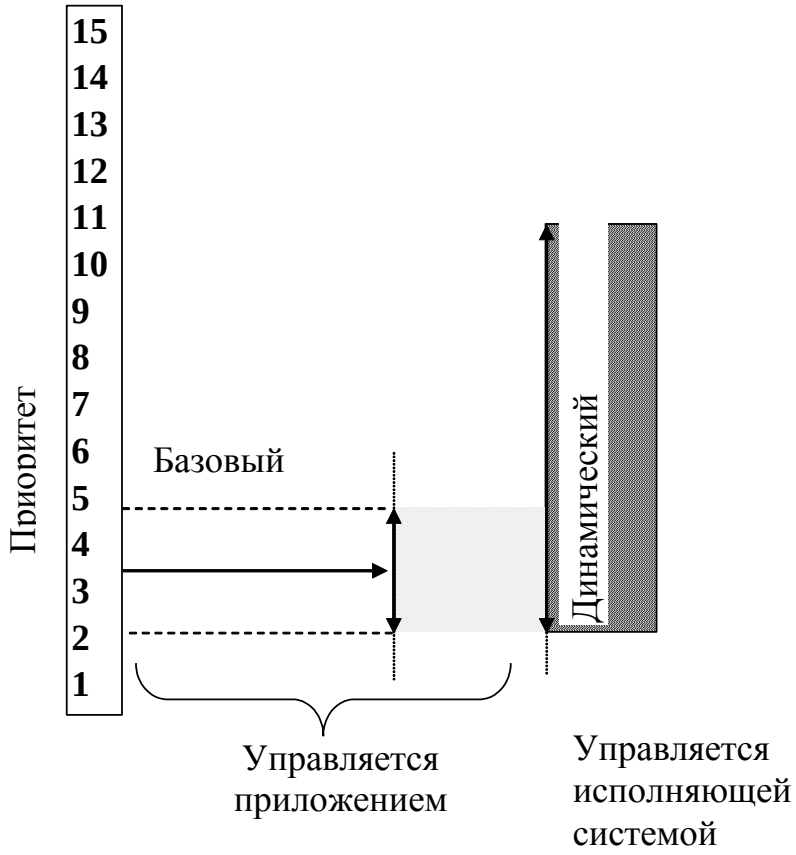

Рис. 10.1. Динамическое изменение приоритета потока

Замечания:

• динамический уровень приоритета никогда не опускается ниже базового;

• потоки с уровнями приоритетов реального времени (от 16 до 31) системой никогда не меняются;

• система не допускает динамического повышения приоритета потока выше уровня 15.

Так как динамическое изменение приоритета системой может отрицательно сказаться на производительности некоторых приложений, разработчики Windows добавили две функции, позволяющие отключать этот механизм: SetProcessPriorityBoost() и SetThreadPriorityBoost().

10.3. ДЕМОНСТРАЦИЯ ПЕРЕКЛЮЧЕНИЯ ПОТОКОВ СИСТЕМОЙ

Покажем на простых примерах, как влияют приоритеты потоков на предоставление им процессорного времени. Будем использовать одну и ту же потоковую функцию для создания трех потоков. Каждый поток выводит на консоль свою букву. Смена символа означает переключение потока системой.

```
//Потоковая функция
const int DELAY = 100000; //константа для организации
                    программной задержки (подбирается в
                    зависимости от частоты процессора)
const int NUMBER = 100; //количество символов, выводимых
                    каждым потоком
DWORD WINAPI DrawFunc(LPVOID p)
{
    char c= reinterpret_cast<char>(p);
    for(int I = 0; I<NUMBER; I++)
    {
        std::cout<<c;
        for(int i=0; i<DELAY; i++){;} //имитация деятельности
    }
    return 0;
}
```

Запуск потоков с одинаковым приоритетом:

```
int main()
{
    SetProcessAffinityMask(GetCurrentProcess(), 1);
                    //предписали всем потокам выполняться
                    только на первом ядре
    HANDLE arh[3];
    arh[0] =
                    CreateThread(0,0,DrawFunc,reinterpret_cast<
                    LPVOID>('A'),0,0);
    arh[1] =
                    CreateThread(0,0,DrawFunc,reinterpret_cast<
                    LPVOID>('B'),0,0);
```

```
    arh[1] =
                    CreateThread(0,0,DrawFunc,reinterpret_cast<
                    LPVOID>('C'),0,0);
    for(int i=0; i<3; i++) CloseHandle(arh[i]);
    return 0;
}
```

На экране увидим что-нибудь, похожее на:

AAABBBCCCAAABBBCCC…

Запуск потоков с разными приоритетами:

```
int main()
{
    SetProcessAffinityMask(GetCurrentProcess(), 1);
                    //предписали всем потокам выполняться
                    только на первом ядре
    HANDLE arh[3];
    arh[0] = CreateThread(0,0,DrawFunc,
                    reinterpret_cast<LPVOID>('A'),
        CREATE_SUSPENDED,0);
    arh[1] =
                    CreateThread(0,0,DrawFunc,reinterpret_cast<
                    LPVOID>('B'),
        CREATE_SUSPENDED,0);
    arh[2] =
                    CreateThread(0,0,DrawFunc,reinterpret_cast<
                    LPVOID>('C'),
        CREATE_SUSPENDED,0);
    //Придержим все потоки, пока не поменяли их приоритет
    SetThreadPriority(arh[0], THREAD_PRIORITY_HIGHEST);
    SetThreadPriority(arh[1], THREAD_PRIORITY_LOWEST);
    SetThreadPriority(arh[2], THREAD_PRIORITY_LOWEST);
    //Возобновим выполнение потоков
    for(int i=0; i<3; i++) ResumeThread(arh[i]);
    //Дождемся, пока все потоки завершатся
    WaitForMultipleObjects(3,arh,INFINITE);
    for(int i=0; i<3; i++) CloseHandle(arh[i]);
}
```

На экране увидим что-нибудь, похожее на:

AAAAAA…AAAABBBCCCBBBCCC…

11. СИНХРОНИЗАЦИЯ ПОТОКОВ

В предыдущих примерах все запущенные потоки выполнялись псевдо параллельно, так как на одном процессоре в один момент времени реально может выполняться только один поток. Переключение потоков осуществляла система, отдавая предпочтение потокам с более высоким приоритетом.

11.1. СИНХРОНИЗАЦИЯ ПОТОКОВ. ХУДШЕЕ, ЧТО МОЖНО СДЕЛАТЬ

Если бы синхронизирующих объектов не было, а ОС не умела отслеживать особые события, потоку пришлось бы самостоятельно синхронизировать себя с другими потоками, применяя грубый метод (на подобие приведенного ниже). Но поскольку в систему встроена поддержка синхронизации объектов, никогда не применяйте этот метод!

Рассмотрим нерекомендуемый способ и отметим его недостатки. Суть метода в том, что поток синхронизирует себя с каким-либо событием в другом потоке, постоянно анализируя значение глобальной переменной, доступной обоим потокам. Пусть вызывающий поток в некоторый момент времени не может продолжать выполнение, пока порожденный поток не закончит какие-то действия и не устанавливает флаг gFlag в значение TRUE:

```
BOOL gFlag = FALSE; //глобальный флаг, синхронизирующий потоки
//Потоковая функция
DWORD WINAPI MyThread(LPVOID ptr)  //функция потока
{
    ... //выполняем какие-то вычисления
    gFlag = TRUE;   //с этого момента может продолжаться
                    порождающий поток
    ...
    return 0;
}
//Порождающий поток
```

```
int main()
{
    SetProcessAffinityMask(GetCurrentProcess(), 1);
                    //предписали всем потокам выполняться
                    только на первом ядре
    ...
    CreateThread(…MyThread,…);
    ...
    while(gFlag == FALSE); //ждем установки флага
    //продолжаем выполнение
}
```

Чем плох такой подход?

- вызывающий поток при таком решении проблемы не исключается из планирования, а когда система выделяет ему квант времени, занимается бесполезной работой (проверкой условия продолжения цикла), то есть система продолжает выделять ему процессорное время на “ожидание” за счет других потоков, скорее всего занимающихся чем-то более полезным;
- если у порождающего потока приоритет выше, чем у порожденного, последнему придется очень долго ждать, когда, наконец, система выделит потоку с более низким приоритетом квант на выполнение (причем, если бы низкоприоритетным потокам иногда не выделялся бы квант времени, порождающему потоку пришлось бы ждать бесконечно);
- если потоки выполняются в разных процессах, тогда эта переменная должна быть разделяемой;
- так как два потока пользуются одной и той же переменной, стоит запретить компилятору оптимизировать работу с такой переменной (квалификатор volatile).

Вывод: потоки следует синхронизировать, отправляя их «в спячку» до выполнения некоторого условия! Если поток указывает системе, что он «ожидает» наступления некоторого события, система до выполнения условия ожидания исключает поток из планирования, то есть не выделяет ему времени для выполнения!

11.2. ПРОСТЕЙШИЕ СПОСОБЫ СИНХРОНИЗАЦИИ

Следует отметить, что при использовании средств синхронизации приоритеты потоков отходят на второй план.

Для очень простых способов синхронизации предусмотрены системные средства, которые не поддерживаются объектами исполняющей системы.

Функции семейства «InterLocked»

Для некоторых простейших действий предусмотрены функции семейства «InterLocked». Единственное назначение этих функций – указанным способом изменять значение четырехбайтовой переменной (типа LONG), которой одновременно могут пользоваться несколько потоков. Но использование этих функций гарантирует, что на время модификации значения поток, вызвавший такую функцию, получает к переменной монопольный доступ, и система не разрешит никакому другому потоку в этот интервал времени произвести свои действия над переменной. Атомарность операции гарантируется и в том случае, когда во время выполнения этой операции квант времени истек, и даже в том случае, когда в системе появился поток с более высоким приоритетом. Назначением одной из функций этого семейства InterlockedIncrement() является увеличение значения по указанному в качестве параметра адресу на единицу. Прототип:

```
LONG __cdecl InterlockedIncrement(//возвращает новое значение
                    переменной
    LONG volatile *Addend //указатель на модифицируемое
                    значение
);
```

Замечание: существуют ограничения на инкрементируемую таким образом переменную: ее адрес должен быть кратен 4 байтам, то есть, выровнен на границу LONG.

Критические секции

Критические секции (critical sections) - это часть кода, который может выполняться одновременно только одним потоком. Критическая секция:

• не является объектом исполняющей системы;

• дескриптор для нее не формируется;

• память выделяется в адресном пространстве процесса.

Критическая секция является эффективным синхронизирующим средством потоков, но лишь в пределах одного процесса.

Последовательность действий программиста для синхронизации потоков посредством критической секции.

1. Сформировать экземпляр структуры типа CRITICAL_SECTION. Структурная переменная может быть глобальной, локальной или динамической – главное, чтобы она существовала, пока ею будут пользоваться потоки. Поля этой структуры недокументированы. Пользоваться ею можно только посредством соответствующих системных функций.

```
CRITICAL_SECTION cs; //поля этой структуры не
                     документированы. Пользоваться ею можно
                     только посредством соответствующих
                     системных функций
```

2. Прежде чем синхронизировать потоки с помощью критической секции, нужно ее инициализировать вызовом системной функции InitializeCriticalSection():

```
InitializeCriticalSection(&cs);
```

3. Перед каждым выполнением фрагмента кода, который должен выполняться только одним потоком, следует вызвать функцию EnterCriticalSection().

```
EnterCriticalSection(&cs);
```

Если критическая секция «свободна», данный поток ее «занимает» и продолжает выполняться. Если эта критическая секция уже «занята» другим потоком, система усыпляет данный поток до освобождения критической секции, передавая остаток кванта другим потокам. Анализ состояния критической секции заключается в проверке счетчика ссылок (одного из полей структуры):

• если счетчик нулевой, функция увеличивает его значение на единицу и разрешает выполнение потока дальше;

• если значение счетчика отлично от нуля, но секция захвачена

тем же самым потоком, который вторично вызвал функцию EnterCriticalSection(), счетчик увеличивается на единицу, выполнение потока продолжается;

• если значение счетчика не равно нулю, а секция захвачена другим потоком, данный поток переводится системой в состояние ожидания.

Это означает, что только один поток может выполнять код, который «защищен» критической секцией.

4. После выполнения защищенного блока кода следует вызвать функцию LeaveCriticalSection().

```
LeaveCriticalSection(&cs);
```

Эта функция уменьшает счетчик ссылок. Если после декремента счетчик становится равным нулю (это означает освобождение критической секции), система пробуждает один из потоков, ожидающих посредством функции EnterCriticalSection() ее освобождения.

Замечание: если один и тот же поток несколько раз вызвал функцию EnterCriticalSection(), он должен столько же раз вызвать функцию LeaveCriticalSection(), чтобы освободить секцию.

5. Когда надобность в синхронизации потоков отпадает, следует вызвать функцию

```
DeleteCriticalSection(&cs);
```

Эта функция освобождает все ресурсы, включенные в критическую секцию.

Замечание: есть еще одна функция, связанная с критическими секциями:

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

Если при вызове этой функции указанная критическая секция не занята никаким потоком, она передает права на критическую секцию вызвавшему потоку и возвращает TRUE. Если же критическая секция уже занята, функция возвращает FALSE, но не приостанавливает выполнение потока. Эта функция может быть использована для

проверки состояния критической секции.

Рассмотрим пример использования критической секции. Пусть критическая секция позволяет только одному из потоков выводить на консоль «свой» символ.

```
const int DELAY = 100000; //константа для организации
                    программной задержки
const int NUMBER = 10; //количество символов, выводимых каждым
                    потоком
CRITICAL_SECTION cs; //глобальная переменная, которой можно
                    пользоваться как в порождающем, так и в
                    порожденном потоке
//Потокоаая функция получает в качестве параметра код символа
DWORD WINAPI CSFunction(LPVOID p)
{
    char c= (char)p; //преобразуем тип параметра к фактическому
                    типу
    EnterCriticalSection(&cs); //защищаем вывод NUMBER символов
                    критической секцией
    for(int I=0;I<NUMBER;I++)
    {
        std::cout<<c;
        for(int i=0;i<DELAY; i++); //программная задержка
                    (имитация деятельности)
    }
    LeaveCriticalSection(&cs); //освобождаем критическую секцию
    return 0;
}
//Порождающий поток:
int main()
{
    InitializeCriticalSection(&cs);
    HANDLE ar[2];
    //запускаем потоки
    ar[0] = CreateThread(0,0,CSFunction,
                    reinterpret_cast<LPVOID>('A'), 0,0);
    ar[1] = CreateThread(0,0,CSFunction,
                    reinterpret_cast<LPVOID>('B'), 0,0);
    WaitForMultipleObjects(2,ar,true,INFINITE); // ждем, пока
                    оба потока завершатся
    CloseHandle(ar[0]);
```

```
    CloseHandle(ar[1]);
    DeleteCriticalSection(&cs); //освобождаем критическую
                    секцию
}
```

Первый поток, который вызывает функцию EnterCriticalSection(), занимает критическую секцию и продолжает выполняться. Второй поток также вызывает функцию EnterCriticalSection(), но так как секция уже занята, система переводит второй поток в состояние ожидания. Когда первый поток освобождает секцию посредством функции LeaveCriticalSection(), система пробуждает второй поток, и он продолжает выполнение. На экране картинка будет примерно такой:

```
AAAAAAAAAABBBBBBBBBB
```

11.3. СВОБОДНОЕ СОСТОЯНИЕ ОБЪЕКТА ИСПОЛНЯЮЩЕЙ СИСТЕМЫ

Для отправления потоков «в спячку» (исключения из диспетчеризации) и для их «пробуждения» при выполнении некоторого условия система использует следующие понятия:

- большинство объектов исполняющей системы могут находиться в двух состояниях свободное/занятое (signaled|nonsignaled);
- система предоставляет программисту механизмы, которые позволяют отслеживать переход объекта в свободное состояние (например, функции ожидания);
- некоторые объекты предназначены исключительно для обеспечения механизма синхронизации («event», «mutex» и т. д.);
- на время ожидания система исключает поток из диспетчеризации, не тратя на него процессорное время.

Большинство объектов исполняющей системы могут находиться в одном из двух состояний: свободен (signaled) и занят (nonsignaled). Для объектов разного типа свободное состояние имеет разный смысл.

Для некоторых (основных) типов объектов описание свободного состояния приведено в табл. 11.1.

Таблица 11.1

Переход объектов в свободное состояние

Тип объекта	Переходит в сигнальное состояние, когда	Ожидающие данный объект потоки
Process	Завершается его последний поток	Все могут продолжаться (пробуждаются)
Thread	Находится в занятом состоянии на всем протяжении своей “жизни”, а в свободное состояние переводится системой, когда его выполнение заканчивается	Все могут продолжаться
File	Завершается операция ввода/вывода	Все могут продолжаться
Event (notification type) с ручным сбросом	Поток устанавливает объект в сигнальное состояние с помощью SetEvent(), а в занятое - ResetEvent()	Все могут продолжаться
Event (synchronization type) с автоматическим сбросом	Поток устанавливает объект в сигнальное состояние с помощью SetEvent(), а в занятое - автоматически переводит функция ожидания	Один поток может продолжаться, событие переходит в занятое состояние
Semaphore	Если счетчик свободных ресурсов n>0	Продолжить выполнение могут n ожидающих потоков
Timer (notification type) ожидаемый таймер с ручным сбросом	Истекает заданный интервал времени	Все могут продолжаться

Окончание табл. 11.1

Тип объекта	Переходит в сигнальное состояние, когда	Ожидающие данный объект потоки
Timer (synchronization type) ожидаемый таймер синхронизации	Истекает заданный интервал времени	Один поток может продолжаться, таймер переводится системой в занятое состояние
Mutex	Поток устанавливает объект в свободное состояние с помощью ReleaseMutex()	Один поток может продолжаться

11.4. ФУНКЦИИ ОЖИДАНИЯ

Потоки «усыпляют» себя до освобождения какого-либо синхронизирующего объекта с помощью специальных функций ожидания. Функция принимает в качестве параметра один или несколько дескрипторов объектов исполняющей системы. Любая из функций проверяет состояние объекта (объектов) и, если объект (все объекты или некоторые из указанных) находится в занятом состоянии, отправляет вызвавший ее поток в спячку. Поток исключается из планирования до перехода ожидаемого функцией объекта в свободное состояние. Система отслеживает этот переход и пробуждает поток, то есть осуществляет возврат управления из функции ожидания.

Функции, ожидающие освобождения одного объекта

Функции, предназначенные для ожидания перехода одного объекта в свободное состояние: WaitForSingleObject(), WaitForSingleObjectEx() и SignalObjectAndWait(). Рассмотрим прототип и механизм работы функции WaitForSingleObject().

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

Возвращает управление в следующих случаях:

- указанный объект перешел в свободное состояние;
- истек указанный интервал времени.

В качестве параметра dwMilliseconds могут выступать два особых значения – табл. 11.2.

Таблица 11.2

Специфические значения тайм аута в функциях ожидания

dwMilliseconds	Описание
0	Функция только проверяет состояние объекта (занят/свободен) и сразу же возвращает управление
INFINITE	Время ожидания бесконечно. Если объект так и не освободится, поток останется в неактивном состоянии и никогда не получит процессорного времени.

В соответствии с причинами, по которым поток продолжает выполнение, функция WaitForSingleObject() может возвращать одно из следующих значений – табл. 11.3.

Таблица 11.3

Варианты возвращаемых функцией одиночного ожидания значений

Возвращаемое значение	Описание
WAIT_OBJECT_0	Объект перешел в свободное состояние
WAIT_TIMEOUT	Объект не перешел в свободное состояние, но интервал времени истек
WAIT_ ABANDONED	Ожидаемый объект является мьютексом, который не был освобожден владеющим им потоком перед окончанием этого потока. Объект мьютекс автоматически переводится системой в состояние свободен. Такая ситуация называется «отказ от мьютекса»
WAIT_FAILED	Произошла ошибка. Чтобы узнать ее код, вызовите функцию GetLastError()

Функция WaitForSingleObjectEx() обладает по сравнению с предыдущей функцией дополнительными возможностями. Кроме перечисленных выше случаев, эта функция пробуждает поток, когда в очередь потока ставится процедура завершения ввода/вывода (I/O completion routine) или происходит асинхронный вызов процедуры (APC). Прототип функции:

```
DWORD WINAPI WaitForSingleObjectEx(
    HANDLE hHandle,
    DWORD dwMilliseconds,
    BOOL bAlertable //если задается значение TRUE и поток
                    находится в состоянии ожидания, то в
                    контексте потока выполняется процедура
                    завершения или асинхронная процедура
);
```

В дополнении к перечисленным кодам завершения добавляется еще продолжение табл. 11.3.

Продолжение табл. 11.3

Возвращаемое значение	Описание
WAIT_IO_ COMPLETION	Ожидание было прервано одним или несколькими асинхронными вызовами процедур

Функция SignalObjectAndWait() позволяет вызвавшему эту функцию потоку автоматически установить один объект в свободное состояние, а ждать освобождения другого объекта. Прототип функции:

```
DWORD WINAPI SignalObjectAndWait(
    HANDLE hObjectToSignal, //дескриптор объекта, который
                    требуется перевести в свободное состояние
    HANDLE hObjectToWaitOn, //дескриптор объекта, который
                    требуется перевести в занятое состояние
    DWORD dwMilliseconds, //интервал в миллисекундах
    BOOL bAlertable //аналогично предыдущей функции
);
```

Функции множественного ожидания

Функции множественного ожидания WaitForMultipleObjects(), WaitForMultipleObjectsEx(), MsgWaitForMultipleObjects(), и

MsgWaitForMultipleObjectsEx() можно использовать для ожидания перехода в свободное состояние одного и более объектов. В качестве одного из параметров задается массив дескрипторов ожидаемых объектов. В зависимости от указанных параметров функции возвращают управление в следующих случаях:

- один из указанных объектов или все перешли в свободное состояние;
- истек указанный интервал времени.

```
DWORD WaitForMultipleObjects(DWORD nCount, //количество
                    объектов в массиве описателей объектов, на
                    который указывает lpHandles (не более
                    MAXIMUM_WAIT_OBJECTS, определенной как 64)
                        CONST HANDLE* lpHandles,.// указатель
                    на массив описателей синхронизирующих
                    объектов (возможно разного типа). Объект
                    можно указать в массиве только один раз,
                    иначе произойдет ошибка, даже если объект
                    идентифицируют два разных описателя.
                        BOOL bWaitAll, ..// если TRUE – функция
                    ждет освобождения всех объектов из списка,
                    если FALSE – ждет освобождения хотя бы
                    одного объекта из списка. В последнем
                    случае функция сканирует массив описателей
                    (начиная с нулевого элемента), и первый же
                    освободившийся объект прерывает ожидание.
                        DWORD dwTimeOut);// интервал ожидания в
                    миллисекундах
```

В табл. 11.4 приведены возвращаемые функцией значения.

Таблица 11.4

Варианты возвращаемых функцией множественного ожидания значений

Возвращаемое значение	Описание
от WAIT_ OBJECT_0 до (WAIT_OBJECT_ 0 + nCount - 1)	При ожидании всех объектов это значение указывает на то, что все объекты освободились. При ожидании одного из объектов это значение представляет собой наименьший индекс (в массиве lpHandles) описателя объекта, перешедшего в свободное состояние.

Окончание табл. 11.4

Возвращаемое значение	Описание
WAIT_TIMEOUT	Интервал истек, а условия, заданные bWaitAll-параметром не выполнены.
от WAIT_ ABANDONED_0 до (WAIT_ ABANDONED_0 + nCount - 1)	При ожидании всех объектов это значение указывает на то, что все объекты перешли в свободное состояние и, по крайней мере, один объект-мьютекс был переведен в свободное состояние из-за отказа от него.
WAIT_FAILED	Произошла ошибка. Чтобы узнать ее код, вызовите функцию GetLastError()

Функции MsgWaitForMultipleObjects() и MsgWaitForMultipleObjectsEx() позволяют оконному приложению, ожидая освобождения одного или нескольких объектов, обрабатывать все или некоторые сообщения посредством функций GetMessage() или PeekMessage(). Прототип:

```
DWORD MsgWaitForMultipleObjects(
  DWORD nCount,           // количество «ожидаемых»
                     дескрипторов
  CONST PHANDLE pHandles, // адрес массива дескрипторов
  BOOL bWaitAll,          // TRUE ждет освобождения всех
                     объектов плюс получения какого-нибудь
                     события для нашего приложения,
                     FALSE – выполнения хотя бы одного из
                     перечисленных условий
  DWORD dwMilliseconds,  // интервал ожидания или INFINITE
  DWORD dwWakeMask       // критерий «пробуждения»:
                     QS_ALLEVENTS – все события,
                     QS_MOUSE – мышиные
                     QS_KEY – клавиатура и т.д.
);
```

Пример использования функции MsgWaitForMultipleObjects(). Ждем, пока порожденный поток завершится, но обрабатываем свои сообщения.

```
{
    //запускаем поток (потоковая функция ThreadFunction()
```

```
    HANDLE hThread = CreateThread(0,0,ThreadFunction, 0, 0, 0);
    while(true) //бесконечный цикл, который прерывается при
                завершении запушенного потока
    {
        DWORD res = MsgWaitForMultipleObjects(1,//количество
                ожидаемых объектов
            &hThread,//указатель на массив «ожидаемых»
                дескрипторов, в нашем случае дескриптор
                единственный
            FALSE, //поток должен пробуждаться, когда ожидаемый
                объект перешел в свободное состояние или
                приложению поступило событие
            INFINITE,//ждем бесконечно
            QS_ALLEVENTS);//управление возвращается при любом
                событии
        if(res==WAIT_OBJECT_0) // если значение лежит в
                диапазоне от WAIT_OBJECT_0 до
                (WAIT_OBJECT_0+количество ожидаемых
                объектов -1), то освободился один из
                ожидаемых объектов, в нашем случае
                завершился ожидаемый поток
            break;// прервали бесконечный цикл
        // иначе поступило сообщение. Обрабатываем его
        MSG m;
        while(PeekMessage(&m,0,0,0,PM_REMOVE))//пока в очереди
                есть сообщения – извлекаем из очереди
        {
            DispatchMessage(&m);//отправляем на обработку
                сообщение
        }
    }
    // Закрывайте описатели, как только необходимость в них
                отпадает!!!
    CloseHandle(hThread);
}
```

Замечание: функции WaitForMultipleObjectsEx() и MsgWaitForMultipleObjectsEx() обладают дополнительными возможностями.

Тревожные функции ожидания

Функции MsgWaitForMultipleObjectsEx(), SignalObjectAndWait(), WaitForMultipleObjectsEx(), WaitForSingleObjectEx(), SleepEx()

являются расширением обычных функций ожидания. Главное отличие: поддерживают асинхронный режим и перекрывающийся ввод/вывод, то есть эти функции могут возвращать управление в следующих случаях:

- если истек заданный интервал времени;
- если ожидаемые объекты перешли в свободное состояние (кроме функции SleepEx());
- если завершаются назначенные процедуры завершения.

11.5. СИНХРОНИЗИРУЮЩИЕ ОБЪЕКТЫ

В среде, позволяющей использовать несколько потоков одновременно, очень важно синхронизировать их деятельность. Для этого в ОС, базирующихся на Win32, предусмотрен целый ряд синхронизирующих объектов. Каждый объект может находиться в одном из двух состояний: свободном (signaled) и занятом (nonsignaled). В общем случае поток синхронизирует себя с другим так: он «засыпает», и ОС, не выделяя ему процессорного времени, приостанавливает его выполнение. Но, прежде чем заснуть, поток посредством синхронизирующего объекта сообщает системе, какое событие должно произойти, чтобы его исполнение возобновилось. Как только указанное событие произойдет, поток вновь будет включен системой в планирование.

Замечание: синхронизирующие объекты могут быть неименованными или именованными. Именование позволяет легко использовать объекты для синхронизации потоков, выполняющихся в разных процессах. Неименованные объекты менее удобно, но тоже можно использовать в разных процессах посредством наследования или дублирования дескриптора. Обычно неименованные объекты используют для синхронизации потоков в одном процессе.

Мьютексы (mutexes)

Слово mutex происходит от «mutually exclusive” (взаимно

исключающий). Эти объекты весьма похожи на критические секции за исключением того, что с их помощью можно синхронизировать доступ к данным со стороны нескольких процессов. Мьютексы полезны, когда нужно обеспечить, чтобы в каждый момент времени только один поток мог модифицировать данные или пользоваться каким-либо разделяемым ресурсом.

Специфика мьютекса: объект-мьютекс отличается от других синхронизирующих объектов исполняющей системы тем, что занявшему его потоку передаются права на владение им. Прочие синхронизирующие объекты могут быть либо свободны, либо заняты – и все. А мьютекс способен еще и запоминать, какому потоку он принадлежит. Это означает, что только поток-владелец может его освободить (в отличие от других синхронизирующих объектов, которые может занять один поток, а освободить другой).

Рассмотрим последовательность действий программиста и соответствующие системные средства для синхронизации потоков посредством мьютекса.

1. Сначала необходимо создать новый объект мьютекс или получить доступ к уже существующему объекту с помощью функции CreateMutex(). Прототип функции:

```
HANDLE CreateMutex(
                LPSECURITY_ATTRIBUTES lpMutexAttributes,
                    //защита и возможность наследования
                BOOL bInitialOwner, //TRUE - позволяет
                    вызывающему потоку немедленно вступить во
                    владение мьютексом при его создании
                    (Замечание: этот флаг игнорируется, если
                    мьютекс уже существует). Мьютекс
                    оказывается в занятом состоянии, поэтому
                    любой другой поток, ожидающий данный
                    мьютекс, будет приостановлен, пока поток,
                    создавший этот объект, не освободит его
                    Если параметр равен FALSE, это означает,
                    что мьютекс не принадлежит ни одному из
                    потоков, поэтому первый же поток из числа
                    ожидающих этот объект может занять его и
                    тем самым продолжить свое выполнение
                LPCTSTR lpName ); //имя объекта или 0
```

Мьютекс создается с правами доступа MUTEX_ALL_ACCESS и может быть использован в любом потоке любого процесса. При успешном завершении функция возвращает процессо-зависимый описатель мьютекса, иначе NULL. Проверить, действительно ли при вызове функции был создан новый объект мьютекс, можно вызовом функции GetLastError(). Если функция GetLastError() вернет нуль, значит, был создан новый объект-мьютекс, а если значение ERROR_ALREADY_EXISTS, значит, новый объект не создан, а возвращен описатель уже существующего объекта-мьютекса с тем же именем.

Замечание: если Вы знаете, что существует мьютекс с данным именем, и Вы хотите сделать этот объект доступным другим процессам, есть способ получить описатель уже существующего мьютекса:

```
HANDLE OpenMutex( DWORD dwDesiredAccess, //флаги доступа:
                    общие для всех объектов (DELETE,
                    READ_CONTROL, SYNCHRONIZE, WRITE_DAC, and
                    WRITE_OWNER) + специфические для мьютекса
                    (MUTEX_ALL_ACCESS и MUTEX_MODIFY_STATE)
                  BOOL bInheritHandle,  //если TRUE, дочерний
                    процесс наследует возвращаемый описатель
                    мьютекса
                  LPCTSTR lpName); //указатель на строку с именем
                    мьютекса
```

При вызове функции OpenMutex() система сканирует существующие объекты-мьютексы, проверяя – нет ли среди них объекта с именем, указанным в lpName. Обнаружив таковой, она создает описатель объекта, специфичный для данного процесса, и возвращает его вызвавшему потоку. В дальнейшем любой поток из данного процесса может использовать этот описатель при вызове любой функции, требующей такой описатель. А если объекта с таким именем нет, функция возвращает NULL.

После того, как мьютекс перешел в занятое состояние, любой поток, ожидающий этот мьютекс посредством функции ожидания, будет отправлен в спячку.

2. После выполнения кода, защищенного мьютексом, объект-мьютекс требуется освободить. Для перевода мьютекса в свободное состояние используется функция:

```
BOOL ReleaseMutex(HANDLE hObject);
```

Замечание: если вызвавший функцию ReleaseMutex() поток не является владельцем указанного мьютекса, функция возвращает FALSE.

3. Когда мьютекс больше не нужен, следует вызвать функцию:

```
BOOL CloseHandle(HANDLE hObject);  //где hObject – описатель
                    мьютекса, возвращенный функцией
                    CreateMutex()
```

Рассмотрим пример использования синхронизации двух потоков посредством мьютекса.

```
const int NUMBER = 10; //количество символов, выводимых каждым
                    потоком
HANDLE hMutex; //глобальный дескриптор предназначен для
                    использования в порождающем и порожденных
                    потоках
//Потоковая функция получает в качестве параметра код символа
DWORD WINAPI MutexFunc(LPVOID p)
{
    WaitForSingleObject(hMutex,INFINITE);//если мьютекс
                    свободен, то он переводится в занятое
                    состояние и выполнение потока продолжается.
                    Если мьютекс занят, то поток засыпает
    char c= (char)p;
    for(int I = 0; I<NUMBER; I++)
    {
        std::cout<<c;
        Sleep(100);
    }
    ReleaseMutex(hMutex); //мьютекс переводится в свободное
                    состояние, это означает, что один из
                    ожидающих потоков может захватить мьютекс и
                    продолжить выполнение
    return 0;
}
//Порождающий поток
int main()
```

```
{
    HANDLE ar[2];
    hMutex = CreateMutex(0,FALSE,0);//создаем свободный
                    неименованный мьютекс
//Порождаем потоки
    arh[0] =::CreateThread(0,0,MutexFunc,(LPVOID)'A',0,0);
    arh[1] =::CreateThread(0,0,MutexFunc,(LPVOID)'B',0,0);
    WaitForMultipleObjects(2,arh,TRUE,INFINITE); //ждем
                    завершения обоих потоков для того, чтобы
                    закрыть дескриптор мьютекса
    CloseHandle(hMutex);
    CloseHandle(arh[0]);
    CloseHandle(arh[1]);
    return 0;
}
```

Замечание: с объектом-мьютексом сопоставляется счетчик, фиксирующий, сколько раз данный объект передавался во владение потоку. Поэтому, если поток вызовет функцию WaitForSingleObject() для уже принадлежащего ему объекта-мьютекса, он продолжит выполнение (так как система определит, что поток уже владеет этим мьютексом). При этом счетчик мьютекса увеличится на 1. А значит, чтобы перевести мьютекс в свободное состояние потоку необходимо соответствующее число раз вызвать функцию ReleaseMutex(). Если Вы помните, функции EnterCriticalSection() и LeaveCriticalSection() действуют по отношению к критическим секциям аналогичным образом.

Замечание: объект-мьютекс отличается от других синхронизирующих объектов тем, что занявшему его потоку передаются права на владение им. Прочие синхронизирующие объекты могут быть либо свободны, либо заняты – и все. А мьютексы способны еще и запоминать, какому потоку они принадлежат. Отказ от объекта-мьютекса происходит, когда ожидавший его поток захватывает этот объект (переводя его в занятое состояние), а потом завершается, «забыв» вызвать функцию ReleaseMutex(). В таком случае получается, что мьютекс занят и никогда не освободится,

поскольку другой поток не сможет этого сделать. Однако система не допускает подобных ситуаций и, заметив, что произошло, автоматически переводит мьютекс в свободное состояние. Поэтому потоки, ожидающие данный объект посредством функций ожидания, получают возможность захватить его, а соответствующая функция ожидания возвращает WAIT_ABANDONED вместо WAIT_OBJECT_0. Посредством возвращаемого значения можно понять, что мьютекс освобожден некорректно. Это обычно указывает на то, что в исходном коде программы есть «bug». Выяснить, что сделал с защищаемыми данными завершенный поток – владелец мьютекса, увы, невозможно.

Сравнение мьютексов и критических секций

Мьютексы и критические секции одинаковы в том, как они влияют на планирование ждущих потоков, но различны по некоторым другим характеристикам. Сравнение этих синхронизирующих средств представлено в табл. 11.5.

Таблица 11.5

Сравнение критических секций и мьютексов

Характеристики	Мьютекс	Критическая секция
Быстродействие	Малое	Высокое
Возможность использования за границами процесса	Да для именованных мьютексов, при использовании неименованных нужно обеспечивать механизмы разделения объектов процессами	Нет
Тип используемых переменных	HANDLE hmfx;	CRITICAL_SECTION cs;

Окончание табл. 11.5

Характеристики	Мьютекс	Критическая секция
Инициализация	hmtx = CreateMutex (NULL, FALSE, NULL);	InitializeCriticalSection(&cs);
Очистка	CloseHandle(hmtx);	DeleteCriticalSection(&cs);
Бесконечное ожидание	WaitForSingleObject (hmtx, INFINITE);	EnterCrittcalSection(&cs);
Проверка состояния	WaitForSingleObject (hmtx, 0);	TryEnterCriticalSection (&cs);
Ожидание в течение произвольного периода времени	WaitForSingleObject (hmtx, dwMilliseconds);	Невозможно
Освобождение	ReleaseMutex(hmtx);	LeaveCriticalSecliun(&cs);
Возможность параллельного ожидания других объектов ядра	Да (с помощью WaitForMultipleObjects или аналогичной функции)	Нет

Семафоры

Если мьютекс дает право выполняться только одному ожидающему потоку, семафор позволяет выполняться одновременно нескольким потокам. Специфика семафора заключается в том, что он содержит два дополнительных данных:

- максимальное количество потоков, которые могут выполняться одновременно (система не допускает присвоения отрицательных значений);
- текущее количество потоков, которые могут выполняться одновременно, то есть, сколько потоков, ожидающих семафор, могут

продолжить выполнение (значение счетчика может быть изменяться в диапазоне от 0 до максимального числа потоков).

Оба значения используются системой для перевода семафора из состояния «занят» в состояние «свободен» и наоборот определены следующие правила:

• когда счетчик текущего числа потоков становится больше 0, семафор переходит в свободное состояние;

• если этот счетчик равен 0, семафор занят;

• при каждом вызове функции ожидания система проверяет значение счетчика текущих потоков. Если значение больше нуля, система уменьшает счетчик на 1, а поток продолжает выполнение. Если счетчик семафора равен нулю, система отправляет поток в спячку. Поток находится в состоянии ожидания, пока значение счетчика не станет больше 0.

Замечание: семафоры, в отличие от мьютексов, не передаются во владение какому-либо одному потоку. Это означает, что один поток может занять семафор (уменьшив его счетчик на 1), а другой – освободить семафор (увеличив его счетчик).

Последовательность действий программиста при использовании семафора для синхронизации потоков.

1. Объект семафор создается функцией:

```
HANDLE CreateSemaphore( // возвращается дескриптор объекта или
                        0 в случае неудачи
                        LPSECURITY_ATTRIBUTES
                        lpSemaphoreAttributes, //защита и
                        возможность наследования
                        LONG lInitialCount, // позволяет задать -
                        сколько потоков изначально могут
                        выполняться. Это значение должно лежать в
                        диапазоне 0- lMaximumCount. Семафор
                        свободен, если его счетчик больше нуля и
                        занят, если счетчик равен 0.
                        LONG lMaximumCount, // сколько максимально
                        потоков могут выполняться (значение должно
                        быть больше нуля)
                        LPCTSTR lpName); //если lpName равен 0,
                        семафор создается без имени, и им можно
```

```
                    пользоваться только в пределах одного
                    процесса.
```

Замечание: если семафор с указанным именем уже существует, параметры lInitialCount и lMaximumCount игнорируются.

Для того чтобы узнать был ли создан новый объект, или возвращен описатель уже существующего семафора – воспользуйтесь функцией GetLastError(). Если был создан новый объект, функция вернет 0, если объект уже существовал, вернет ERROR_ALREADY_EXISTS. Доступ к существующему объекту можно также получить с помощью функции:

```
HANDLE WINAPI OpenSemaphore(
       DWORD dwDesiredAccess,
       BOOL bInheritHandle,
       LPCTSTR lpName
);
```

2. После выполнения защищенного семафором блока кода, поток должен увеличить значение счетчика текущего числа ресурсов, вызывая функцию ReleaseSemaphore().

```
BOOL ReleaseSemaphore( HANDLE hSem, //дескриптор семафора
        LONG lReleaseCount, //на сколько единиц увеличить
                    счетчик, обычно на 1
        PLONG pPreviousCount); //указатель на переменную, в
                    которой функция вернет предыдущее значение
                    счетчика, или 0, если предыдущее значение
                    не важно
```

Функция просто складывает величину lReleaseCount со значением счетчика текущего числа потоков, не допуская превышения максимального значения.

3. Когда семафор больше не нужен, не забудьте вызвать функцию:

```
BOOL CloseHandle(HANDLE hObject); //где hObject – описатель
                    семафора, возвращенный функцией
                    CreateSemaphore()
```

Рассмотрим пример использования объекта семафор для синхронизации трех потоков.

```
const int DELAY = 100000; //константа для организации
                    программной задержки
```

```
const int NUMBER = 10; //количество символов, выводимых каждым
                    потоком
HANDLE hSema; //глобальный дескриптор предназначен для
                    использования в порождающем и порожденных
                    потоках
//Потоковая функция получает в качестве параметра код символа
DWORD WINAPI SemaFunc(LPVOID p)
{
    WaitForSingleObject(hSema,INFINITE);// если текущий счетчик
                    семафора больше нуля, счетчик
                    декрементируется, и поток продолжает
                    выполнение. Если счетчик семафора равен
                    нулю, система отправляет поток в спячку, до
                    того, как какой-нибудь другой поток не
                    освободит семафор (то есть увеличит его
                    счетчик)
    char c= (char)p;
    for(int I = 0; I<NUMBER; I++)
    {
        std::cout<<c;
        //программная задержка для имитации деятельности
        for(int i=0; i<DELAY; i++){;}
    }
    LONG count;
    ReleaseSemaphore(hSema,1,&count); //увеличить счетчик на 1,
                    сформировать предыдущее значение счетчика в
                    переменной count
    return 0;
}
//Порождающий поток
int main()
{
    HANDLE ar[3];
hSema = CreateSemaphore(0, //безопасность по умолчанию, объект
                    не наследуется
                    2, // изначально два потока могут
                    выполняться
                    2, // максимально выполняться могут тоже
                    два потока
                    0);// неименованный объект
//Порождаем потоки
    char symbol='A';
```

```
    for(int i=0; i<3: i++)
    {
        arh[i] = CreateThread(0,0,SemaFunc,(LPVOID)symbol,0,0);
        symbol++;
    }
    WaitForMultipleObjects(3,arh,TRUE,INFINITE); //ждем
                    завершения всех потоков для того, чтобы
                    закрыть дескриптор семафора
    for(int i=0; i<3: i++)
    {
        CloseHandle(arh[i]); //закрываем дескрипторы потоков
    }
    CloseHandle(hMutex); //закрываем дескриптор мьютекса
    return 0;
}
```

Результат работы будет выглядеть примерно так:

AAAAABBBBBAAAAABBBBBCCCCCCCCCC

Поясним результат. Первые два потока, которые вызвали функцию ожидания WaitForSingleObject(), получили возможность продолжить выполнение. Если они выполняются на одном ядре, то система выделяет им поочередно квант времени. В результате значение счетчика текущих потоков стало равным 0. Поэтому третий поток, при вызове функции WaitForSingleObject(), был переведен в состояние ожидания. И только после того, как один из выполняющихся потоков вызвал функцию ReleaseSemaphore(), спящий поток был переведен в состояние готовности и продолжил выполнение.

События (events)

Событие – это самая примитивная разновидность синхронизирующих объектов. Событие просто уведомляет об окончании какой-либо операции.

Существуют два типа событий:

- с ручным сбросом (manual reset events) – при переходе в свободное состояние позволяют возобновить выполнение сразу всех ожидающих потоков и остаются в свободном состоянии, пока не

будет вызвана функция ResetEvent();

• с автоматическим (autoreset events) сбросом – позволяют возобновить выполнение только одного ожидающего потока и переводятся системой автоматически в занятое состояние.

Замечание: события с автоматическим сбросом работает аналогично мьютексу, но предоставляет дополнительную интересную возможность посредством функции PulseEvent().

Рассмотрим последовательность действий программиста при синхронизации потоков посредством события.

1. Объекты-события создаются функцией CreateEvent().

```
HANDLE CreateEvent( //возвращается дескриптор объекта или 0 в
                    случае неудачи
          LPSECURITY_ATTRIBUTES lpEventAttributes, //защита и
                    возможность наследования
          BOOL bManualReset, //TRUE – создастся событие со
                    сбросом вручную, FALSE – с автосбросом
          BOOL bInitialState, // начальное состояние объекта-
                    события: TRUE – свободен, FALSE – занят
          LPCTSTR lpName ); //имя объекта или 0
```

Замечание: если объект-событие с указанным именем уже существует, то возвращаемый описатель характеризует существующий объект-событие, а поля bManualReset и bInitialState игнорируются, так как они уже были установлены при создании объекта. Для того чтобы узнать – был ли создан новый объект или будет использоваться уже существующий – вызовите функцию GetLastError(). Если объект уже существовал, функция вернет ERROR_ALREADY_EXISTS, иначе нуль.

Замечание: если вы хотите получить дескриптор существующего объекта-события, пользуйтесь функцией

```
HANDLE OpenEvent( DWORD dwDesiredAccess,
     BOOL bInheritHandle,
     LPCTSTR lpName );
```

2. Поток ожидает события посредством любой функции ожидания. События с автоматическим сбросом при наступлении события переводятся функциями ожидания снова в занятое состояние.

Таким образом, если в какой-то момент времени событие с автоматическим сбросом оказывается свободным, то только один из потоков, ожидающих данный объект-событие, пробуждается, а остальные продолжают спать. События с ручным сбросом функциями ожидания автоматически в занятое состояние не переустанавливаются. Если на момент перехода объекта-события в свободное состояние, существует несколько спящих (ожидающих это событие) потоков, система пробуждает все ожидающие данное событие потоки.

3. После выполнения блока кода, защищенного событием, объект необходимо перевести в свободное состояние функцией:

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

При вызове этой функции все потоки, ожидающие событие с ручным сбросом, могут продолжить выполнение. Если используется событие с автоматическим сбросом, один поток может продолжить выполнение, для всех остальных система переводит событие снова в занятое состояние. Если ожидающих потоков нет, событие остается в свободном состоянии.

4. События с ручным сбросом остаются в свободном состоянии, пока какой-нибудь поток явно не сбросит событие в занятое состояние вызовом функции:

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

5. Для того, чтобы позволить всем ожидающим потокам продолжить выполнение, и в то же время после того, как они все переведены системой в состояние готовности, установить объект-событие в занятое состояние, можно использовать функцию:

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

6. Когда объект-событие больше не нужен, не забудьте вызвать функцию:

```
CloseHandle(HANDLE hObject);    //где hObject – описатель
                    события, возвращенный функцией
                    CreateEvent()
```

Рассмотрим пример использования события с автоматическим сбросом для синхронизации потоков.

```
const int DELAY = 100000; //константа для организации
                    программной задержки
const int NUMBER = 10; //количество символов, выводимых каждым
                    потоком
HANDLE hAutoEvent; //глобальный дескриптор предназначен для
                    использования в порождающем и порожденных
                    потоках
//Потоковая функция получает в качестве параметра код символа
DWORD WINAPI AutoEventFunc(LPVOID p)
{
    WaitForSingleObject(hAutoEvent,INFINITE);//если объект
                    свободен, то переводится в занятое
                    состояние и выполнение продолжается. Если
                    объект занят, то поток засыпает
    char c= (char)p;
    for(int I = 0; I<NUMBER; I++)
    {
        std::cout<<c;
        for(int i=0; i<DELAY; i++){;}
    }
    SetEvent(hAutoEvent);//событие переводится в свободное
                    состояние, это означает, что один из
                    ожидающих потоков может продолжить
                    выполнение, а объект будет снова переведен
                    системой в занятое состояние
    return 0;
}
//Порождающий поток
int main()
{
    HANDLE ar[2];
    hAutoEvent= CreateEvent(0, //безопасность по умолчанию и
                    объект не наследуется
                    FALSE, // с автосбросом
                    TRUE, // изначально событие свободно
                    0);// объект неименованный
//Порождаем потоки
    arh[0] =::CreateThread(0,0, AutoEventFunc,(LPVOID)'A',0,0);
    arh[1] =::CreateThread(0,0, AutoEventFunc,(LPVOID)'B',0,0);
    WaitForMultipleObjects(2,arh,TRUE,INFINITE); //ждем
                    завершения обоих потоков для того, чтобы
                    закрыть дескриптор события
```

```
    CloseHandle(hMutex);
    CloseHandle(arh[0]);
    CloseHandle(arh[1]);
    return 0;
}
```

Результат работы:

```
AAAAAAAAAAABBBBBBBBBBB
```

Рассмотрим пример использования события с ручным сбросом для синхронизации потоков.

```
const int DELAY = 100000; //константа для организации
                    программной задержки
const int NUMBER = 10; //количество символов, выводимых каждым
                    потоком
HANDLE hManualEvent; //глобальный дескриптор предназначен для
                    использования в порождающем и порожденных
                    потоках
//Потоковая функция получает в качестве параметра код символа
DWORD WINAPI ManualEventFunc(LPVOID p)
{
    WaitForSingleObject(hManualEvent,INFINITE); //если объект
                    свободен, то выполнение продолжается,
                    событие остается свободным,
                    если объект занят, то поток засыпает

    char c= (char)p;
    for(int I = 0; I<NUMBER; I++)
    {
        std::cout<<c;
        for(int i=0; i<DELAY; i++){;}
    }
    return 0;
}
//Порождающий поток
int main()
{
    HANDLE ar[2];
    hManualEvent = CreateEvent(0, //безопасность по умолчанию и
                    объект не наследуется
                    FALSE, // с ручным сбросом
```

```
                    TRUE, // изначально объект занят
                    0);// объект неименованный
//Порождаем потоки
    arh[0] =::CreateThread(0,0,
                    ManualEventFunc,(LPVOID)'A',0,0);
    arh[1] =::CreateThread(0,0,
                    ManualEventFunc,(LPVOID)'B',0,0);
    while(true)
    {
        cout<<"Запустить потоки (Y/N) ?";
        char yesno:
        cin>> yesno:
        if((yesno=='Y') || (yesno=='y')) break; //ждем, пока
                    пользователь не захочет запустить все
                    потоки
    }
    SetEvent(hManualEvent);//разрешили всем потокам выполняться

    WaitForMultipleObjects(2,arh,TRUE,INFINITE); //ждем
                    завершения обоих потоков для того, чтобы
                    закрыть дескриптор события
    CloseHandle(hManualEvent);
    CloseHandle(arh[0]);
    CloseHandle(arh[1]);
    return 0;
}
```

Пока пользователь не подтверждает запуск потоков, потоки находятся в состоянии ожидания. Как только пользователь нажмет ‘y’ или ‘Y’, оба потока проснутся и, если они выполняются на одном ядре, то система будет переключать их, поочередно выделяя квант времени каждому:

AAAAAABBBBBAAAAABBBBBB

Таймеры ожидания (waitable timers)

Ожидаемые таймеры – это объекты исполняющей системы, самостоятельно переходящие в свободное состояние в определенное время или регулярно через заданные промежутки времени. Появились в ОС Windows NT 4, поэтому значение константы _WIN32_WINNT должно быть больше или равно 0x0400. Ожидаемые таймеры бывают

двух видов – рис. 11.1:

• таймеры с ручным сбросом, которые остаются в свободном состоянии, пока не вызвана функция SetWaitableTimer();

• таймеры с автоматическим сбросом (или таймеры синхронизации), которые автоматически переводятся системой в занятое состояние.

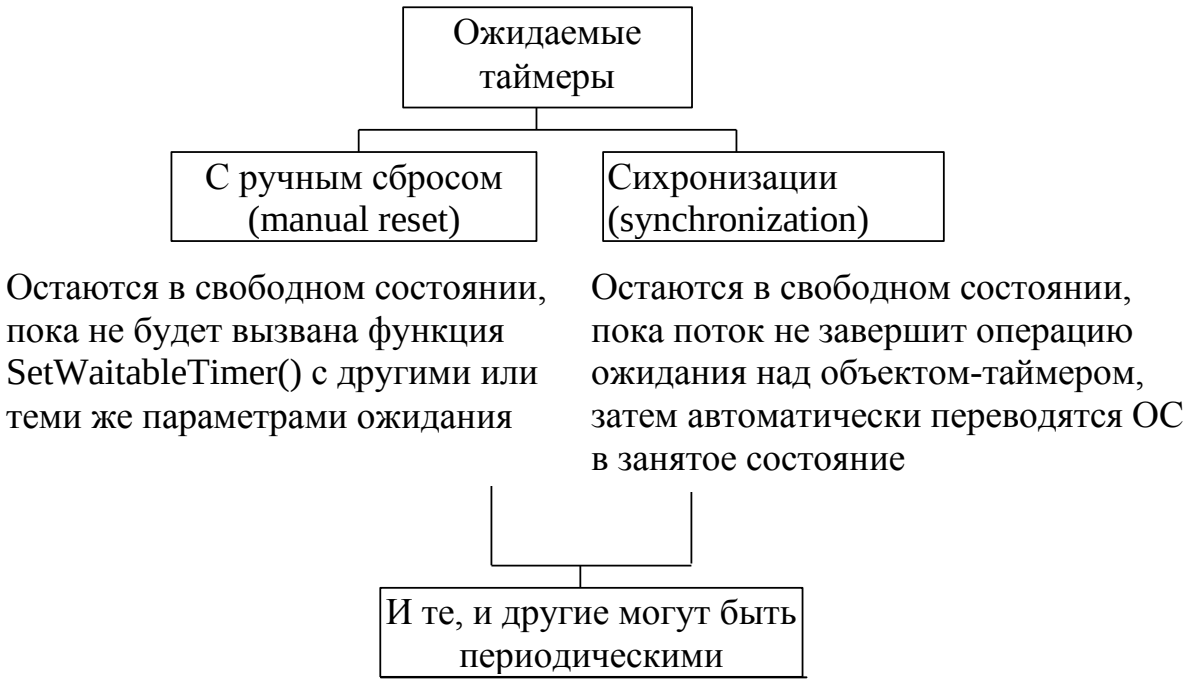

Рис. 11.1. Виды ожидаемых таймеров

Последовательность действий программиста при использовании ожидаемого таймера для синхронизации потоков.

1. Создание ожидаемого таймера с помощью функций CreateWaitableTimer() или CreateWaitableTimerEx().Таймер создается в неактивном состоянии.

```
HANDLE CreateWaitableTimer( //возвращаемое значение –
                    дескриптор созданного или уже существующего
                    объекта или 0 в случае неудачи
    LPSECURITY_ATTRIBUTES lpsa, //защита и возможность
                    наследования
    BOOL bManualReset,//TRUE – с ручным сбросом, то есть если
                    таймер переходит в свободное состояние,
```

```
                    пробуждаются все потоки, ожидающие этот
                    таймер. Перевод таймера в занятое состояние
                    осуществляется только новым вызовом
                    SetWaitableTimer() возможно с другими
                    параметрами
                    FALSE – синхронизации, то есть, если таймер
                    переходит в свободное состояние,
                    пробуждается только один поток, а таймер
                    снова автоматически переводится в занятое
                    состояние системой
    LPCTSTR lpszTimerName); //имя объекта или 0
```

Можно получить доступ к существующему таймеру посредством

```
HANDLE OpenWaitableTimer(DWORD dwDesiredAccess,
BOOL bInheritHandle,
LPCTSTR lpszTimerName);
```

2. Настройку и запуск таймера осуществляет функция SetWaitableTimer(). Она же используется для нового запуска таймера с теми же или новыми параметрами.

```
BOOL SetWaitableTimer(HANDLE hTimer,
const LARGE_INTEGER* pDueTime,//когда таймер должен
                    запуститься (перейти в свободное состояние
                    первый раз) в UTC (coordinated universal
                    time – в 100нс интервалах). Если величина
                    положительная, то абсолютное время, если
                    отрицательная, то относительное после
                    вызова данной функции. Замечание: Если на
                    момент вызова функции этот момент времени
                    уже прошел, то таймер сразу переходит в
                    свободное состояние
LONG lPeriod,//0 – срабатывает один раз, >0 в мс – период, <0
                    – ошибка и функция вернет FALSE
PTIMERAPCROUTINE pfnCompletionRoutine, //необязательный
                    указатель на функцию завершения (если
                    функция завершения не используется, то 0)
LPVOID lpArgToCompletionRoutine,.//необязательные данные,
                    которые передаются процедуре завершения или
                    0
BOOL fResume);// Обычно FALSE. Флаг полезен на компьютерах с
                    режимом сна – когда таймер сработает,
                    компьютер выйдет из режима сна (если он там
                    находился)
```

Периодический таймер будет срабатывать каждый раз по

истечении указанного периода, пока не будет вызвана одна из функций CancelWaitableTimer() или SetWaitableTimer() с другими параметрами.

3. Для сброса таймера (установки таймера в неактивное состояние) используется функция:

```
BOOL CancelWaitableTimer(HANDLE hTimer);
```

4. Когда таймер больше не нужен, следует закрыть его дескриптор с помощью функции:

```
CloseHandle(HANDLE hObject); //где hObject – описатель
                    таймера, возвращенный функцией
                    CreateWaitableTimer()
```

Рассмотрим пример использования таймера с автоматическим сбросом. Требуется примерно через 10 секунд после запуска программы с периодом 500 мс выдавать звуковой сигнал. «Бибиканье» должно продолжаться, пока пользователь не нажмет любую клавишу.

```
HANDLE hT; //дескриптор таймера
//Потоковая функция с помощью функции ожидания отслеживает
                    периодические срабатывания таймера и выдает
                    звуковой сигнал
DWORD WINAPI BeepFunc(LPVOID p)
{
    while(true)
    {
        WaitForSingleObject(hT, INFINITE); //при срабатывании
                    таймера поток пробуждается
        MessageBeep(-1); //и выдает звуковой сигнал
    }
    return 0;
}

int main()
{
    //Создание таймера
    hT=CreateWaitableTimer(0,FALSE,0); //синхронизации,
                    неименованный
    if(hT!=0)
```

```
    {
        //Настройка таймера
        LARGE_INTEGER time; //время запуска
        time.QuadPart = -10000000LL; //относительно момента
                    вызова функции SetWaitableTimer()
        SetWaitableTimer(hT,&time,500,0,0,FALSE); //через
                    интервал time от момента вызова, с
                    интервалом 500 мс, процедура не задана,
                    данные для процедуры не заданы
        //Запуск потока
        HANDLE hThread = CreateThread(0,0,BeepFunc, 0,0,0);
                    //безопасность – по умолчанию, стек – по
                    умолчанию, параметр потоковой функции 0,
                    создаваемый поток готов к выполнению,
                    идентификатор не нужен
        if(hThread) //если удалось создать поток
        {
            _getch(); //ждем, пока пользователь не нажмет любую
                    клавишу
            TerminateThread(hThread, 0); //завершаем бибикающий
                    поток
            CloseHandle(hThread); // закрываем дескриптор
                    порожденного потока
            CloseHandle(hT); // закрываем дескриптор таймера
        }
    }
    return 0;
}
```

Рассмотрим использование процедуры завершения. Требуется с помощью процедуры завершения заданное количество раз вывести время срабатывания ожидаемого таймера.

Четвертый параметр функции SetWaitableTimer() - это указатель на функцию вида:

```
VOID CALLBACK TimerAPCProc(
   LPVOID lpArg,   // указатель на данные (пятый параметр
                   SetWaitableTimer)
   DWORD dwTimerLowValue,      // младшая часть 64-разрядного
                   значения времени (когда таймер перешел в
                   свободное состояние)
   DWORD dwTimerHighValue )    // старшая часть
```

Специфика:

• процедура завершения выполняется в том же потоке, который вызывал функцию SetWaitableTimer ();

• поток, вызывающий функцию SetWaitableTimer() должен быть в «тревожном» (alertable) состоянии, то есть ожидает вызова процедуры завершения посредством одной из функций: SleepEx(), WaitForSingleObjectEx(), WaitForMultipleObjectsEx(), MsgWaitForMultipleObjectsEx(), SignalObjectAndWait().

```
//Процедура асинхронного вызова
VOID CALLBACK TimerAPCProc(
   LPVOID lpArg,               // данные, передаваемые
                  процедуре посредством параметра функции
                  SetWaitableTimer()
   DWORD dwTimerLowValue,      // младшие 4 байта 64-
                  разрядного времени срабатывания таймера в
                  UTC
   DWORD dwTimerHighValue )    //  старшие 4 байта

{
    FILETIME ft;
    ft.dwLowDateTime = dwTimerLowValue;
    ft.dwHighDateTime = dwTimerHighValue;
//Переводим значение времени из формата UTC в системное время
    SYSTEMTIME st;
    FileTimeToSystemTime(&ft,&st);
//Выводим время
    std::cout<<"minutes="<< st.wMinute <<"
                  seconds="<<st.wSecond<<std::endl;
}
//Создание и запуск таймера. При инициализации таймера
                  указывается адрес процедуры, которую
                  система будет вызывать при срабатывании
                  таймера
int main ()
{
//Создание объекта
    HANDLE hSynchroTimer = CreateWaitableTimer(0,FALSE,0);
    if(hSynchroTimer)
    {
```

```
//Инициализация и запуск таймера
    __int64 interval = -3*10000000;//примерно 3 сек
    LARGE_INTEGER li;
    li.HighPart = (LONG)  ( interval >> 32 );
    li.LowPart = (DWORD) ( interval & 0xFFFFFFFF );
    BOOL  bSuccess = SetWaitableTimer(
            hSynchroTimer,           // дескриптор объекта
            &li,       // время первого срабатывания
            2000,              // период срабатывания - 2
                    секунды
            TimerAPCProc,      // процедура завершения
            0,           // данные, передаваемые процедуре
                    завершения
            FALSE );           // Do not restore a suspended
                    system
      if ( bSuccess )

            for ( int i=0; i < 10; i++ )
            {
               SleepEx(
                  INFINITE,     //усыпляем свой поток до того
                    момента, когда в очереди потока появится
                    процедура завершения
                  TRUE );       // переводим поток в тревожное
                    состояние
            }
        CloseHandle(hSynchroTimer);
    }
}
```

Пример задания абсолютного времени срабатывания таймера. Такой способ использования таймера хорош, когда нужно, чтобы компьютер «напомнил» хозяину о наступлении какого-то важного момента, например, о наступлении Нового Года. Напоминание будет выдано, правда, только в том случае, если пользователь не будет выключать компьютер, а только отправлять компьютер в режим сна.

```
int main()
{
    HANDLE hTimer = CreateWaitableTimer(0,TRUE,0);
    if(hTimer)
```

```
    {
        //Формируем абсолютное время срабатывания – за минуту
                    до наступления Нового Года
    SYSTEMTIME st = {0};
    st.wYear = 2010;
    st.wMonth = 12;
    st.wDayOfWeek = 0;//игнорируется
    st.wDay = 31;//день месяца
    st.wHour = 23;
    st.wMinute = 59;
    st.wSecond = 0;
    st.wMilliseconds = 0;
//Переводим системное время в формат времени UTC, требуемый
                    функцией SetWaitableTimer()
    FILETIME ftLocal, ftUTC;
    SystemTimeToFileTime(&st,&ftLocal);//системное в местное
    LocalFileTimeToFileTime(&ftLocal, &ftUTC);

    //Преобразую FILETIME в LARGE_INTEGER из-за различий в
                    выравнивании данных
    LARGE_INTEGER li;//или __int64
    li.LowPart = ftUTC.dwLowDateTime;//в сформированное в li
                    время
    li.HighPart = ftUTC.dwHighDateTime;
    //Устанавливаю таймер на 23.59 31 декабря 2010 года
    BOOL b=SetWaitableTimer(hTimer,&li,0,0,0,TRUE);
    WaitForSingleObject(hTimer, INFINITE);
    MessageBeep(-1);
    std::cout<<"Happy New Year!!!";
    }
    getch(); //ждем, пока пользователь полюбуется на выведенное
                    на экран поздравление
}
```

Пример задания относительного времени срабатывания таймера. Например, пользователь поставил варить яйца и занялся работой за компьютером. Чтобы вода не выкипела, нужно выдать напоминание через 10 минут.

```
int main()
{
    HANDLE hTimer = CreateWaitableTimer(0,TRUE,0);
    LARGE_INTEGER li;//или __int64
    // относительно момента запуска таймера
    li.QuadPart=-(LONGLONG)5000 * 10000;// 5 сек

    //Устанавливаю таймер на срабатывание через 5 сек после
                    вызова SetWaitableTimer()
    BOOL b=SetWaitableTimer(hTimer,&li,0,0,0,TRUE);
    WaitForSingleObject(hTimer, INFINITE);
    MessageBeep(-1);
    std::cout<<"Ready!!!";
    }
    getch();

}
```

12. КАНАЛЫ

Каналы используются для обмена данными между процессами, выполняющимися как на одном компьютере, так и на разных компьютерах в сети.

12.1. МЕХАНИЗМ УДАЛЕННОГО ОБМЕНА ДАННЫМИ

При удаленном взаимодействии проблема заключается в том, что на другом компьютере может быть:

- другая операционная система;
- другое «железо» (аппаратная платформа);
- другие протоколы обмена.

При взаимодействии по сети задачей операционной системы является обеспечение прикладному программисту виртуального интерфейса для использования устройств ввода/вывода как локально на одном компьютере, так и для удаленного взаимодействия. Для согласования клиента и сервера используется такое понятие, как «Взаимодействие в Открытых Системах» - OSI (Open Systems Interconnection). Согласно модели OSI весь процесс обмена данными разбит согласно на 7 уровней. Идея заключается в том, что каждый уровень (программно и/или аппаратно) предоставляет сервисы для вышележащего слоя и «транслирует» сервисы согласно возможностям нижележащего:

- добавляет к пользовательским данным некоторую дополнительную служебную информацию (или наоборот, удаляет);
- переводит данные в другое представление;
- и вызывает функции следующего уровня обмена.

Прикладному программисту совершенно необязательно знать подробности реализации низкоуровневых протоколов обмена данными, так как разработчики системы предоставляют в распоряжение программиста удобную надстройку - редиректор.

12.2. РЕДИРЕКТОР

Редиректор – это драйвер файловой системы, который функционирует следующим образом – рис. 12.1:

- при работе на стороне клиента (приложения) редиректор формирует и отправляет запрос на операцию ввода/вывода серверу и принимает ответ сервера;
- при работе на стороне сервера (файл, именованный канал, почтовая ячейка) редиректор принимает запрос и инициирует операцию ввода/вывода.

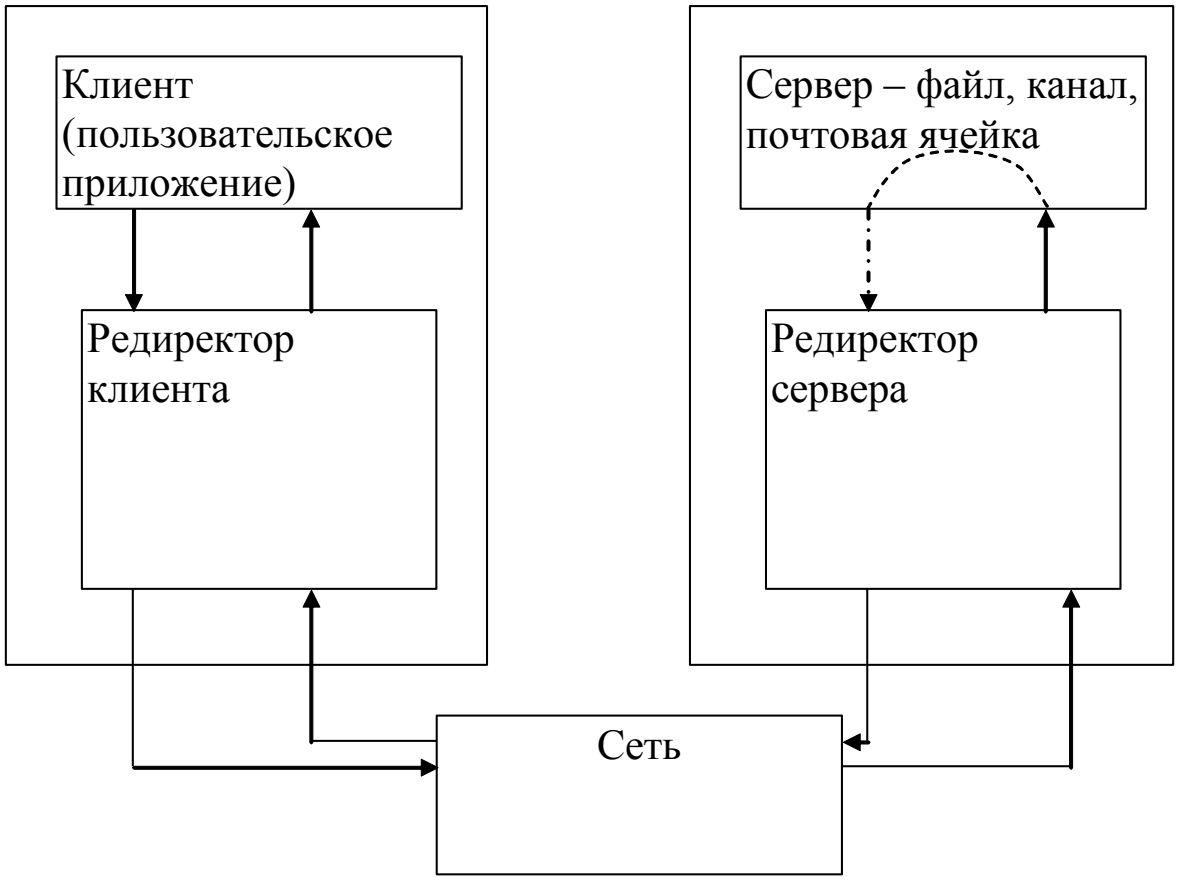

Рис. 12.1. Механизм удаленного обмена данными

Редиректор является фундаментом, на котором базируется не только работа с удаленными файлами, но также каналами (pipes) и

почтовыми ячейками (mailslots). Редиректор осуществляет низкоуровневое взаимодействие, в результате чего клиентское приложение может обмениваться данными с сервером точно так же, как если бы сервер находился на той же рабочей станции.

Замечание: для прикладного программиста это означает, что код программы не зависит от расположения ресурсов.

12.3. ОСОБЕННОСТИ КАНАЛОВ

Каналы - это простое средство межпроцессного взаимодействия, реализованное в разных семействах ОС Windows. Разработка приложений, использующих каналы, не требует знаний низкого уровня (транспортных протоколов типа TCP/IP или IPX), благодаря тому, что каналы для коммуникации по сети используют Microsoft Network Provider (MSNP) редиректор, который скрывает детали протокола.

Отметим особенности каналов:

1. С точки зрения программиста канал – это буфер, используемый двумя процессами и набор системных функций для манипулирования таким специфическим буфером. С точки зрения системы канал – это объект исполняющей системы.

2. Канал представляет собой псевдофайл, поэтому, открыв канал, клиент и сервер канала могут использовать для обмена данными стандартные системные функции ввода/вывода: ReadFile() и WriteFile().

3. Каналы предоставляют надежный способ одностороннего или двухстороннего обмена данными на одном компьютере или между разными компьютерами в сети. Одним из преимуществ использования каналов является защита, которую предоставляют ОС Windows NT и более поздние версии, что позволяет серверу контролировать подключение клиентов. Например, сервер может лимитировать доступ клиента посредством использования средств

безопасности Windows NT и выше.

Замечание: сервер именованного канала можно создать только в WindowsNT и более поздних версиях.

4. Функции ожидания (WaitForSingleObject() и т. д.) непосредственно с дескрипторами каналов не используются, так как у каналов нет сигнального состояния. Но есть возможность сопоставить именованному каналу объект синхронизации “event”, который уже можно использовать для ожидания.

5. Передача данных посредством почтовых ячеек (mailslots) имеет много общего с каналами (оба способа были изначально разработаны для OS/2 LAN Manager), но имеет особенности:

- преимуществом почтовых ячеек является явно реализованный широковещательный режим;
- недостатком – ненадежная однонаправленная передача данных.

6. Канал может быть анонимным (Anonymous Pipe) или именованным (Named Pipe). Именованному каналу программистом должна быть сопоставлена строка, содержащая имя согласно Universal Naming Convention (UNC).

12.4. АНОНИМНЫЕ КАНАЛЫ

Анонимные каналы обычно используются для организации передачи данных между родительскими и дочерними процессами, запущенными на одной рабочей станции или на “отдельно стоящем” компьютере. С анонимными каналами легче работать, но они обладают меньшими возможностями. Специфика:

- передают данные только в одном направлении (симплексные);
- и клиент, и сервер должны быть расположены на одном компьютере (только локально);
- передача осуществляется только как блок байтов (никакой дополнительной информации система при передаче не формирует);

• работают только синхронно (в режиме ожидания завершения операции обмена данными), не поддерживают асинхронный режим, поэтому функции ReadFileEx() и WriteFileEx() с анонимными каналами не работают.

Рассмотрим последовательность действий программиста для организации обмена данными с помощью анонимного канала – рис. 12.2.

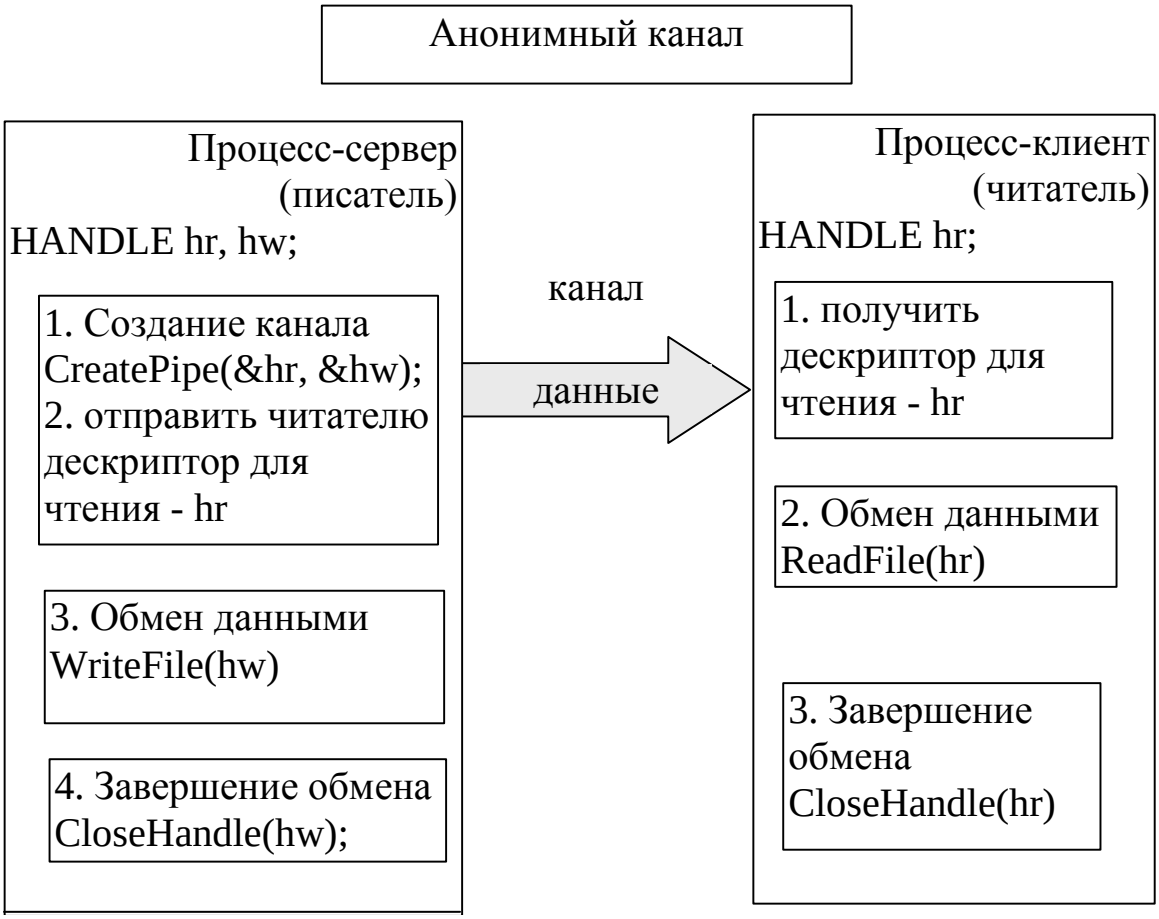

Рис. 12.2. Последовательность действий при организации обмена данными посредством анонимного канала

Создание анонимного канала

Анонимный канал создается системной функцией CreatePipe(). Этот функция формирует два дескриптора: для чтения и для записи в канал. Дескрипторы не взаимозаменяемые: с помощью дескриптора для чтения можно только читать, а с помощью дескриптора для

записи только писать.

Прототип функции:

```
BOOL CreatePipe(//возвращаемое значение TRUE, если канал
                    успешно создан
    PHANDLE phR,    //адрес переменной, в которую будет записан
                    дескриптор канала для чтения (это значение
                    может передаваться в качестве параметра
                    функциям ReadFile())
    PHANDLE phW,    //адрес переменной, в которую будет записан
                    дескриптор канала для записи (это значение
                    может передаваться в качестве параметра
                    функции WriteFile())
    LPSECURITY_ATTRIBUTES pPipeAttributes,  //защита и
                    возможность наследования
    DWORD nSize);   //количество байтов памяти,
                    зарезервированных для буфера канала. Это
                    только пожелание программиста, на самом
                    деле ОС может изменить указанный размер
                    буфера. Если 0, будет создан буфер с
                    размером по умолчанию
```

Соединение процессов посредство неименованного канала

Чтобы два процесса могли с помощью созданного канала обмениваться данными, их требуется соединить этим каналом, то есть каждый процесс должен получить дескриптор «своего конца» канала. Так как обычно анонимные каналы используются для процессов, связанных «родством» (один процесс порождает другой), то логично сделать соответствующий объект наследуемым и передать по наследству дочернему процессу. Последовательность действий программиста может выглядеть примерно так:

- создать и проинициализировать экземпляр структуры типа SECURITY_ATTRIBUTES, указав необходимость наследования;
- при создании канала указать в качестве соответствующего параметра функции CreatePipe() адрес проинициализированной структуры SECURITY_ATTRIBUTES;
- при создании дочернего процесса с помощью функции CreateProcess() указать, что все наследуемые объекты родительского процесса должны быть унаследованы дочерним процессом;

• так как дочерний процесс не знает об унаследованном канале, требуется передать ему соответствующий дескриптор:

- если канал «передан по наследству», то значение дескриптора этого объекта в дочернем процессе будет таким же, как у родителя, поэтому родитель может отправить дочернему «свой» дескриптор в параметрах командной строки или посредством простейших способов межпроцессного обмена данными;
- если канал «передан по наследству», можно перенаправить дочернему процессу стандартный ввод/вывод в канал с помощью полей структуры STARTUPINFO (при этом дочерний процесс может и не подозревать, что вместо стандартного ввода/вывода он читает/пишет в канал);
- если процессы не связаны родством, или создан ненаследуемый канал, то один процесс может продублировать для другого с помощью функции DuplicateHandle() дескрипторы канала.

Обмен данными

Обмен данными с помощью анонимного канала осуществляется только в одну сторону с помощью функций ReadFile() и WriteFile(). Операции ввода/вывода осуществляются только синхронно.

Для чтения из канала используется функция ReadFile(), которой в качестве параметра передается дескриптор «читающего» конца канала. Функция возвращает управление, если

• в канале появились данные для чтения (то есть процесс-«писатель» вызвал со своей стороны функцию WriteFile());

• возникла ошибка (например, пишущий дескриптор был закрыт).

Для записи в канал используется функция WriteFile(), которой в качестве параметра передается дескриптор «пишущего» конца канала. Функция возвращает управление, если:

• указанное количество байтов записано в канал. При этом если за один раз в буфер вся порция передаваемых данных не поместилась, функция будет ждать, пока читатель не прочитает содержимое, и

место в буфере не освободит (размер буфера указывается в качестве параметра функции CreatePipe());

- возникла ошибка (например, читающий дескриптор был закрыт).

Замечание: так как асинхронный и перекрывающийся ввод/вывод анонимными каналами не поддерживается:

- нельзя использовать функции ReadFileEx() и WriteFileEx();
- ненулевое значение параметра LPOVERLAPPED функциями ReadFile() и WriteFile() игнорируется.

Завершение

Анонимный канал существует, пока не закрыты все его открытые дескрипторы, как для записи, так и для чтения. Если только не используется перенаправление стандартного ввода/вывода, все открытые дескрипторы оба процесса должны закрыть с помощью функции CloseHandle().

12.5. ИМЕНОВАННЫЕ КАНАЛЫ

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

Замечание: первоначально именованные каналы разрабатывались Microsoft в качестве интерфейса высоко уровня к протоколам NetBIOS. Используя каналы, программист вместо того, чтобы заниматься на низком уровне маршрутизацией, пересылкой данных и т.д., может открывать каналы и читать/писать данные точно так же, как он делал бы это с файлом. На низком уровне (недоступном пользовательскому режиму) каналы используют для передачи данных один из транспортов нижнего уровня.

В отличие от анонимных каналов именованные обладают

следующей спецификой:

- могут быть дуплексными, то есть передавать данные в обоих направлениях;
- предназначены для обмена данными с процессами, запущенными на другом компьютере в сети, но в частности могут быть использованы и для локального взаимодействия;
- именованному каналу при создании присваивается имя, посредством которого клиент может подключиться к этому каналу даже с другой рабочей станции;
- используют интерфейс Named Pipe File System (NPFS), поэтому являются «псевдофайлами» и для обмена данными могут использовать функции ReadFile() и WriteFile(), а NPFS в свою очередь обращается к редиректору для посылки или приема данных, предоставляя программисту независимость от протоколов низкого уровня;
- у именованных каналов есть понятие «подчиненности»: процесс, создающий канал, называется pipe-сервером, а процесс, который подключается к каналу – pipe-клиентом. Основное отличие сервера и клиента именованного канала: только процесс сервера может создать канал, а процесс-клиент может только подключиться к существующему каналу. Но после того как соединение установлено, оба процесса могут писать и читать данные из канала с помощью функций, таких как ReadFile() и WriteFile();
- используют больше режимов работы по сравнению с анонимными каналами, например, ожидание подключения;
- каждый именованный канал может существовать в нескольких экземплярах (реализациях). Разные экземпляры одного и того же канала создаются, когда в одном и том же сервере (или даже в разных) в функции CreateNamedPipe() в качестве параметра указывается одно и то же имя. Это позволяет одному серверу подключаться к нескольким клиентам. Каждый экземпляр является на самом деле независимым каналом связи, так как у каждой реализации

(для каждого клиента) есть свой буфер и свой дескриптор (чтения\записи). Использование реализаций позволяет нескольким клиентам использовать один и тот же именованный канал одновременно.

Замечание: в ОС Windows NT4 и выше анонимные каналы создаются посредством именованных с уникальным (специфическим) именем – рис. 12.3, поэтому программист может в большинстве случаев передавать функциям именованных каналов дескриптор анонимного канала.

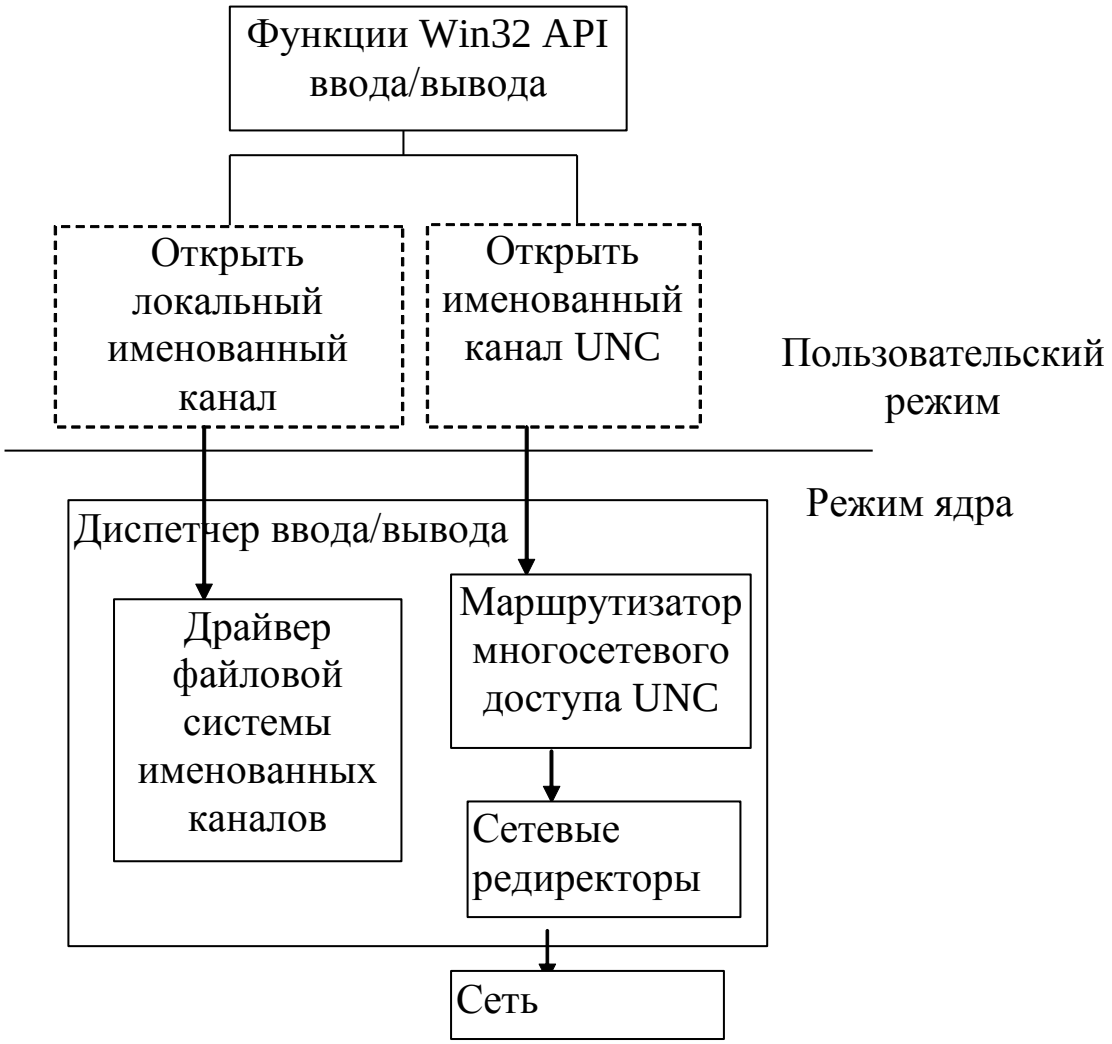

Рис. 12.3. Отличия в организации именованных и анонимных каналов в Windows NT4 и выше

Понятия, связанные с именованными каналами

Замечание: большинство из приведенных ниже характеристик канала задаются при создании канала посредством соответствующих параметров.

1. Односторонние (симплексные) и двухсторонние (дуплексные). Эти термины определяют направление, в котором данные «движутся» по каналу. Для односторонних каналов – один конец канала только пишет, другой – только читает. Соответствующие константы:

- PIPE_ACCESS_INBOUND – клиент посылает данные, сервер читает;
- PIPE_ACCESS_OUTBOUND – сервер пишет, клиент читает;
- PIPE_ACCESS_DUPLEX – обе стороны могут как писать, так и читать.

2. Байтовые каналы и каналы сообщений. Эти понятия относятся к тому, каким образом данные передаются и принимаются, а также к работе канала в режиме ожидания. Для задания режима записи используются константы PIPE_TYPE_BYTE (default) и PIPE_TYPE_MESSAGE. Для задания режима чтения PIPE_READMODE_BYTE (default) и PIPE_READMODE_MESSAGE.

В табл. 12.1 приведены различия поведения байтового канала и канала сообщений.

Таблица 12.1

Отличия байтовых каналов и каналов сообщений

Байтовый канал	Канал сообщений
Используется: • для передачи больших массивов (как только часть данных сформирована, можно передавать и начинать обрабатывать); • или размер просто заранее не известен	Используется, если передаваемые данные имеют смысл только как единое целое, например строки (а строки могут быть разной длины)

Окончание табл. 12.1

Байтовый канал	Канал сообщений
При записи ОС ничем не разделяет блоки данных, записанных посредством двух вызовов WriteFile(), поэтому несколько последовательных вызовов WriteFile() просто подряд заполняют буфер данными, которые потом могут быть считаны за одну операцию функцией ReadFile()	При записи ОС автоматически и прозрачно для программиста предваряет каждое сообщение невидимым заголовком, в котором в частности хранится размер сообщения. Эта информация недоступна (и прозрачна) для программиста, но к ней имеет доступ функция чтения ReadFile()
При чтении за один вызов функции ReadFile(): • извлекается все, что есть в канале; • извлекается заданное количество байтов	При чтении ОС анализирует заголовок и за один вызов функции ReadFile() считывает: • ровно одно сообщение; или если оно большое (то есть длина сообщения больше, чем размер буфера), то считывает то, что есть в буфере (при этом функция GetLastError() вернет ERROR_MORE_DATA), а остаток сообщения можно «дочитать» при следующем вызове функции ReadFile().

Режимы чтения/записи должны быть согласованы в сервере и клиенте. Существует несколько возможных сочетаний параметров:

• PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE – посылка и прием данных осуществляется в режиме сообщений;

• PIPE_TYPE_BYTE | PIPE_READMODE__BYTE– посылка и прием данных осуществляется в байтовом режиме;

• PIPE_TYPE_MESSAGE | PIPE_READMODE_BYTE - отправитель может писать в канал сообщения, а принимающий конец может читать в байтовом режиме. При этом «шапка» сообщения будет отфильтрована (проигнорирована);

• PIPE_TYPE_BYTE | PIPE_READMODE_MESSAGE - ошибка - ERROR_INVALID_PARAMETER, так как «шапку» сообщения взять неоткуда.

Замечания:

• для всех реализаций данного канала тип передачи данных должен быть одинаковым;

• клиент канала, дескриптор которого возвращается функцией CreateFile(), всегда по умолчанию создается в байтовом режиме. Для изменения режима клиента (впрочем, и сервера тоже) можно использовать функцию SetNamedPipeHandleState().

3. Режим ожидания - с блокировкой (синхронно) PIPE_WAIT и без блокировки (асинхронно) PIPE_NOWAIT. Это свойство касается выполнения функций чтения ReadFile(), записи WriteFile(), и соединения ConnectNamedPipe().

Разница представлена в табл. 12.2.

Таблица 12.2

Отличия использования синхронного и асинхронного режимов

Функция	Синхронный режим	Асинхронный режим
ConnectNamedPipe()	Ждет подключения клиента к другому концу (пока клиент не вызовет функцию CreateFile() или CallNamedPipe())	Сразу возвращает управление, а если клиент не подключился, функция GetLastError() возвращает значение ERROR_PIPE_ LISTENING
ReadFile()	Если канал пуст, функция ждет, пока в канале появятся данные	Если канал пуст, возвращает 0 немедленно, а функция GetLastError() - ERROR_NO_DATA
WriteFile()	Если канал практически полон, ждет, пока другой процесс прочитает со своего конца, то есть освободит место для записи.	Если канал практически полон, для байтового канала функция WriteFile() сначала записывает столько, сколько сможет, а для канала сообщений WriteFile() ничего не записывает.

Замечание: асинхронный режим поддерживается только для совместимости с Microsoft LAN Manager version 2.0. Не рекомендуется использовать асинхронный режим. Вместо этого режима для организации асинхронного обмена данными посредством каналов следует использовать перекрывающийся ввод/вывод, так как он позволяет продолжительные во времени операции выполнять в фоновом режиме.

4. Имена каналов формируются в соответствии со спецификацией Universal Naming Convention (UNC):

`\\server\Pipe\[path]name,`

где «server» – это строка с именем компьютера или «.». Так как сервер всегда создается локально, обычно указывается «.». Верхний/нижний регистр не имеет значения.

Поле «name» содержит строку с именем канала. У каждого именованного канала есть уникальное (в системном списке именованных объектов) имя. Сервер указывает это имя при вызове функции CreateNamedPipe(). Клиенты указывают это имя при вызове функций CreateFile() или CallNamedPipe() для установки соединения.

Последовательность действий программиста для организации обмена данными посредством именованного канала

Последовательность действий программиста представлена на рис. 12.3. Так как у именованных каналов есть ярко выраженное отношение подчиненности, то действия в серверном и клиентском процессах принципиально разные.

При создании серверного приложения именованного канала требуется создать один или более реализаций канала, к которым могут подключаться клиенты. С точки зрения сервера реализация канала – это не что иное, как дескриптор, с помощью которого можно соединяться с локальным или удаленным клиентским приложением.

1. Создание именованного канала. Сервер создает реализацию (экземпляр) именованного канала посредством функции CreateNamedPipe(). При этом:

• если соответствующий объект исполняющей системы еще не существует, он создается системой;

• если объект уже существует, формируется новый дескриптор и увеличивается счетчик его пользователей объекта.

Сервер	Клиент
1. Создание именованного канала – функция CreateNamedPipe() 2. Ожидание соединения с клиентом – функция ConnectNamedPipe() 3. Чтение или запись в канал – ReadFile()\WriteFile() 4. Ожидание, пока клиент не прочитает функция FlushFileBuffers() 5. Отключение от клиента – функция DisconnectNamedPipe() (после этого реализация канала может быть вновь использована для соединения с другим клиентским процессом) 6. Закрыть дескриптор данной реализации именованного канала – функция CloseHandle()	1. Попытка установления соединения с сервером – функция CreateFile(). Возможны два варианта: а) канал не существует б) существует, но все его реализации заняты 2. Во втором случае ожидание свободной реализации канала – функция WaitNamedPipe() и повтор установления связи - функция CreateFile() 3. Изменение режима обмена данными (по умолчанию – байтовый канал) - функция SetNamedPipeHandleState() 4. Чтение или запись в канал – ReadFile()\WriteFile() 5. Закрыть дескриптор именованного канала - функция CloseHandle()

Рис. 12.3. Последовательность действий при организации обмена данными посредством именованного канала

Прототип функции:

```
HANDLE CreateNamedPipe( // возвращаемое значение: если канал
                        создан, возвращается дескриптор «серверного
                        конца» данного экземпляра канала, иначе
                        INVALID_HANDLE_VALUE
  LPCTSTR lpName,   //имя канала в форме UNC
  DWORD dwOpenMode, //три группы флагов: направление, контроль
                        ввода/вывода и параметры безопасности.
                        Флаги комбинируются с помощью поразрядного
                        "|". Наиболее интересные значения приведены
                        в табл. 12.3
  DWORD dwPipeMode, //режим, характеризующий способы
                        чтения/записи. Флаги можно объединять
                        поразрядным "|" (по одному из каждой
                        категории). Значения флагов приведены в
                        табл. 12.4
  DWORD nMaxInstances,  // максимальное количество экземпляров,
                        которые можно создать для данного канала
                        (соответственно количество клиентов для
                        данного сервера). Это значение должно быть
                        указано при создании всех экземпляров!
                        Можно указать значение от 1 до
                        PIPE_UNLIMITED_INSTANCES (в этом случае
                        количество лимитируется только ресурсами
                        системы)
  DWORD nOutBufferSize, //размер буфера в байтах,
                        зарезервированного для записи (выходной
                        буфер). Замечание: размеры буферов для
                        чтения и для записи являются на самом деле
                        Вашим пожеланием системе (и должны быть
                        разумными).
  DWORD nInBufferSize,  // размер входного буфера в байтах (для
                        чтения из канала)
  DWORD nDefaultTimeOut,    // время ожидания в мс (как долго
                        клиент будет ожидать соединения с
                        именованным каналом). Это значение по
                        умолчанию используется только клиентами,
                        которые вызывают функцию WaitNamedPipe() и
                        в качестве соответствующего параметра
                        указывают NMPWAIT_USE_DEFAULT_WAIT. Это
                        значение должно быть указано при создании
                        всех экземпляров канала
  LPSECURITY_ATTRIBUTES lpSecurityAttributes //если NULL, то
                        дескриптор защиты по умолчанию и
                        возвращаемый дескриптор не может быть
                        унаследован дочерним процессом.
);
```

Таблица 12.3

Наиболее интересные значения параметра dwOpenMode

dwOpenMode	Флаги	Описание
Направление	PIPE_ACCESS_ DUPLEX	Двунаправленный канал: и клиент, и сервер могут и читать и писать.
	PIPE_ACCESS_ OUTBOUND	Однонаправленный канал: сервер пишет, клиент читает.
	PIPE_ACCESS_ INBOUND	Однонаправленный канал: клиент пишет, сервер читает.
Контроль ввода/вывода	FILE_FLAG_WRITE_ THROUGH	Только для байтовых каналов, когда клиент и сервер находятся на разных компьютерах в сети. Управление из пишущих в именованный канал функций не возвращается, пока данные не передадутся по каналу и не окажутся в буфере читающего конца (на удаленном компьютере). Если флаг не установлен, то система «копит» данные в буфере.
	FILE_FLAG_ OVERLAPPED	Позволяет функциям, использующим операции чтения, записи и соединения использовать перекрывающийся ввод/вывод (то есть возвращать управление немедленно, не дожидаясь завершения обычно длительных перечисленных выше операций).

Окончание табл. 12.3

dwOpenMode	Флаги	Описание
Безопасность	WRITE_DAC	Дает приложению права записи DACL именованного канала.
	ACCESS_SYSTEM_ SECURITY	Дает приложению права записи SACL именованного канала.
	WRITE_OWNER	Дает приложению права записи владельцу именованного канала и группе SID.

Таблица 12.4

Способы записи/чтения данных. Значения параметра dwPipeMode

dwPipeMode	Флаги	Описание
Write	PIPE_TYPE_BYTE	Данные записываются в канал как поток байтов.
	PIPE_TYPE_ MESSAGE	Данные записываются в канал как сообщения.
Read	PIPE_READMODE_ BYTE	Данные считываются из канала как поток байтов.
	PIPE_READMODE_ MESSAGE	Данные считываются из канала как сообщения.
Wait	PIPE_WAIT	Ждущий режим (операции ввода/вывода, такие как ReadFile() не возвращают управление, пока запрос ввода/вывода не будет закончен). Этот флаг устанавливается по умолчанию.
	PIPE_NOWAIT считается устаревшим и не рекомендуется для реализации асинхронного ввода/вывода	Режим без ожидания (управление из функций ввода/вывода возвращается немедленно, не дожидаясь завершения операций).

Обычно сервер взаимодействует с несколькими клиентами. Это можно реализовать несколькими способами:

- для обслуживания каждого клиента создать свой поток (многопоточный сервер). Это самое простое, но не очень эффективное решение, так как на создание каждого потока тратятся системные ресурсы;
- использовать перекрывающийся ввод/вывод посредством указания OVERLAPPED структуры в функциях ReadFile(), WriteFile(), и ConnectNamedPipe();
- использовать перекрывающиеся операции посредством ReadFileEx() и WriteFileEx(), которые позволяют в качестве параметра указать процедуру завершения.

Замечания:

- допускается создание нескольких экземпляров (реализаций) одного и того же канала, благодаря чему канал может поддерживать одновременно несколько клиентов;
- все создаваемые экземпляры данного именованного канала должны указывать одинаковые значения для dwPipeMode, dwOpenMode, nMaxInstances и nDefaultTimeOut.

Пример создания именованного канала:

```
HANDLE hNamedPipe;
hNamedPipe=CreateNamedPipe("\\\\.\\pipe\\MyNamedPipe",
            PIPE_ACCESS_DUPLEX, //двухсторонний обмен
            PIPE_TYPE_MESSAGE| PIPE_READMODE_MESSAGE|
                    PIPE_WAIT,//данные записываются
                    сообщениями,
                    читаются сообщениями, канал будет работать
                    в блокирующем режиме, то есть процесс будет
                    ожидать окончания операции чтения или
                    записи
            PIPE_UNLIMITED_INSTANCES,//число реализаций
                    ограничено только ресурсами системы
            512,512,    //выходной и входной буферы
            100, //тайм-аут
            0);\\дескриптор не может быть унаследован дочерним
                    процессом, защита по умолчанию
```

2. Ожидание подключения клиента. После того, как серверный процесс успешно создал реализацию канала, он может перейти в режим ожидания соединения с клиентским процессом именованного канала (если канал не использует перекрывающийся ввод/вывод, то есть при создании не указан FILE_FLAG_OVERLAPPED). Ожидание соединения можно осуществить посредством:

```
BOOL ConnectNamedPipe(//Возвращаемое значение: TRUE в случае
                      успеха (в синхронном режиме - если
                      подключился клиент).
                      Замечание: если клиент успел подключиться
                      после вызова CreateNamedPipe, но до вызова
                      ConnectNamedPipe(), то возвращается FALSE,
                      но GetLastError() вернет
                      ERROR_PIPE_CONNECTED. В такой ситуации
                      соединение все равно установлено!
            HANDLE hNamedPipe, //дескриптор, возвращенный
                      CreateNamedPipe()
            LPOVERLAPPED lpOverlapped);//Используется только в
                      случае перекрывающегося ввода/вывода обмена
                      данными (если канал был создан с флагом
                      FILE_FLAG_OVERLAPPED) Для синхронного – 0
                      (в этом случае сервис ConnectNamedPipe() не
                      возвращает управления пока клиент не
                      подключится к каналу).
```

Замечание: поведение функции ConnectNamedPipe() зависит от двух условий (задаются при создании канала или позже посредством параметров функции SetNamedPipeHandleState()):

- канал с блокировкой или без;
- режим обмена синхронный или перекрывающийся.

Пример:

```
ConnectNamedPipe(hNamedPipe,0);//в данном случае функция
                      перейдет в состояние ожидания, так как
                      канал был создан для работы в синхронном
                      блокирующем режиме.
```

3. Обмен данными посредством именованного канала

После того, как клиент успешно подключился к каналу, сервер может посылать данные в канал с помощью функции WriteFile() или получать данные из канала с помощью функции ReadFile().

Пример записи данных в канал:

```
HANDLE hNamedPipe;
DWORD cbWritten;
char szBuf[] = "QWERTY";
WriteFile(hNamedPipe,   //дескриптор реализации канала
        szBuf,  //адрес данных
        sizeof(szBuf),  //сколько – в байтах
        &cbWritten, // сколько действительно было записано
        0); //запись в синхронном режиме
```

Пример чтения данных из канала:

```
HANDLE hNamedPipe;
DWORD cbRead;
char szBuf[512];
ReadFile(hNamedPipe,    //дескриптор реализации канала
        szBuf,  //адрес данных
        512,    //сколько – в байтах
        &cbRead,    // сколько действительно было считано
        0); //чтение в синхронном режиме
```

4. Отключение сервера от канала. Установив канал с клиентским процессом при помощи функции ConnectNamedPipe(), серверный процесс может разорвать соединение с помощью функции:

```
BOOL DisconnectNamedPipe(HANDLE hNamedPipe); //дескриптор
                    реализации канала
```

После этого реализация канала может быть вновь использована для соединения с другим клиентским процессом.

5. Закрытие дескриптора канала. Если канал больше не нужен, после отключения от клиентского процесса серверный и клиентский процессы должны закрыть его дескрипторы (если при этом счетчик пользователей объекта исполняющей системы обнуляется, система уничтожает объект и освобождает память):

```
CloseHandle(hNamedPipe);
```

Пример создания простейшего сервера

```
// Server.cpp

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

void main(void)
{
```

```
    HANDLE PipeHandle;
    DWORD BytesRead;
    CHAR buffer[256];

    if ((PipeHandle = CreateNamedPipe("\\\\.\\Pipe\\MyPipe",
        PIPE_ACCESS_DUPLEX,
        PIPE_TYPE_BYTE | PIPE_READMODE_BYTE,
        1,
        0, 0, 1000, 0)) == INVALID_HANDLE_VALUE)
    {
        cout<<"CreateNamedPipe failed with error "<<
            GetLastError()<<endl;
        return;
    }

    cout<<"Server is now running "<<endl;

    if (ConnectNamedPipe(PipeHandle, 0) == 0)
    {
        cout<<"ConnectNamedPipe failed with error "<<
            GetLastError()<<endl;
        CloseHandle(PipeHandle);
        return;
    }

    if (ReadFile(PipeHandle, buffer, sizeof(buffer),
        &BytesRead,  0) <= 0)
    {
        cout<<"ReadFile failed with error"<<
GetLastError()<<endl;
        CloseHandle(PipeHandle);
        return;
    }
    buffer[BytesRead]=0;//на всякий случай
    cout<< buffer<<endl;
    if (DisconnectNamedPipe(PipeHandle) == 0)
    {
        cout<<"DisconnectNamedPipe failed with error "<<
            GetLastError()<<endl;
        return;
    }
    CloseHandle(PipeHandle);
}
```

Рассмотрим последовательность действий при разработке клиента именованного канала. Замечание: клиент именованного канала не может создать реализацию канала. Вместо этого он должен «подключиться» к каналу, созданному сервером (то есть получить

дескриптор «клиентского» конца канала).

1. Ожидание свободной реализации канала. Перед созданием соединения с каналом клиент может проверить: существует ли доступная реализация именованного канала посредством функции:

```
BOOL WaitNamedPipe(
    LPCTSTR lpNamedPipeName,    //имя объекта «именованный
                    канал»
    DWORD nTimeOut  //как долго клиент хочет ждать, пока сервер
                    завершит операцию ConnectNamedPipe(). Для
                    «бесконечного» ожидания -
                    NMPWAIT_WAIT_FOREVER
);
```

2. Соединение со стороны клиента. После того, как ожидание (функция WaitNamedPipe()) завершилось успешно, для подключения к каналу клиентский процесс может воспользоваться функцией CreateFile(). При успешном завершении функция возвращает клиенту дескриптор именованного канала, созданного сервером.

```
HANDLE CreateFile(
    LPCTSTR lpFileName,//имя канала
    DWORD dwDesiredAccess,//режим использования канала
                    клиентом (должен быть согласован с
                    параметром dwOpenMode CreateNamedPipe()
                    создания канала:
                    GENERIC_READ – чтение данных
                    GENERIC_WRITE – запись данных
                    GENERIC_READ | GENERIC_WRITE - запись и
                    чтение
    DWORD dwShareMode,//должен быть 0, так как в каждый момент
                    времени только один клиент может общаться с
                    одной из реализаций сервера
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, //защита и
                    возможность наследования
    DWORD dwCreationDisposition,//должен быть OPEN_EXISTING,
                    так как речь идет о подключении к
                    существующему каналу
    DWORD dwFlagsAndAttributes,//всегда должны быть
                    установлены в FILE_ATTRIBUTE_NORMAL
    HANDLE hTemplateFile    //должен быть 0, так как данный
                    параметр не имеет отношения к именованным
                    каналам
);
```

Пример:

```
char szPipeName[256];
HANDLE hNamedPipe=CreateFile("\\\\.\\pipe\\MyNamedPipe",
        GENERIC_READ | GENERIC_WRITE,//для записи и для чтения
        0, //
        0, //lpSecurityAttributes==0 – дескриптор не
                    наследуется
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        0);
```

3. Изменение режимов

Так как по умолчанию клиент создается в байтовом режиме обмена данными (PIPE_READMODE_BYTE) и блокирующем режиме ожидания (PIPE_WAIT), для изменения этих режимов следует вызвать функцию SetNamedPipeHandleState().

Пример:

```
DWORD dwMode = PIPE_READMODE_MESSAGE | PIPE_NOWAIT;T
   fSuccess = SetNamedPipeHandleState(
      hPipe,    //
      &dwMode,  // новый режим
      NULL,     //
      NULL);    //
```

4. Обмен данными со стороны клиента.

Если функция CreateFile() завершилась успешно, клиент может передавать и принимать данные из именованного канала, используя функции ReadFile() и WriteFile().

5. Завершение работы клиента. Соединение можно закрыть, используя CloseHandle().

Пример создания простейшего клиента. Когда клиентское приложение успешно подключается к серверу, оно посылает серверу строку "This is a test".

```
// Client.cpp

#include <windows.h>
#include <stdio.h>

#define PIPE_NAME "\\\\.\\Pipe\\MyPipe"
```

```
void main(void)
{
    HANDLE PipeHandle;
    DWORD BytesWritten;

    if (WaitNamedPipe(PIPE_NAME, NMPWAIT_WAIT_FOREVER) == 0)
    {
        printf("WaitNamedPipe failed with error %d\n",
            GetLastError());
        return;
    }

    // Create the named pipe file handle
    if ((PipeHandle = CreateFile(PIPE_NAME,
        GENERIC_READ | GENERIC_WRITE, 0,
        (LPSECURITY_ATTRIBUTES) NULL, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        (HANDLE) NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error %d\n",
GetLastError());
        return;
    }

    if (WriteFile(PipeHandle, "This is a test", 14,
&BytesWritten,
        NULL) == 0)
    {
        printf("WriteFile failed with error %d\n",
GetLastError());
        CloseHandle(PipeHandle);
        return;
    }

    printf("Wrote %d bytes", BytesWritten);

    CloseHandle(PipeHandle);
}
```

Другие системные функции для работы с именованными каналами

1. Функция TransactNamedPipe() объединяет передачу и прием данных (вызов функций WriteFile() и ReadFile()) в одну сетевую операцию. Работает только в режиме обмена сообщениями, поэтому если сервер создал канал в режиме чтения байтов, то функция завершится неудачно.

2. Функция CallNamedPipe() подключает клиента к каналу в режиме сообщений, ожидает свободной реализации канала, пишет, читает и закрывает соединение.

Обычно сценарий взаимодействия клиентского процесса с серверным заключается в следующей последовательности действий:

- подключение к каналу с помощью функции CreateFile();
- выполнение операций записи/чтения с помощью функции TransactNamedPipe();
- отключение от канала с помощью функции CloseHandle().

Вызов функции CallNamedPipe() позволяет выполнить эти операции за один прием при условии:

- канал открыт в режиме передачи сообщений;
- клиент посылает одно сообщение серверу и в ответ также получает от сервера одно сообщение.

3. Функция PeekNamedPipe() похожа на функцию ReadFile(), но данные из канала не извлекает, а просто копирует их в буфер.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows: пер. с англ. /Дж. Рихтер. - 4-е изд. - СПб., : Питер; М. : Издательско-торговій дом "Русская редакция", 2001. - 752 с., ил.

2. Вильямс А. Системное программирование в Windows 2000 для профессионалов : Пер. с англ. : СПб.: Изд-во Питер, 2001. – 624 с., ил.

3. Харт Дж. М. Системное программирование в среде Win32, 2 - е изд. : Пер. с англ. : - М. : Издательский дом “Вильямс”, 2001. – 464 с., ил.

4. Руссинович М., Соломон Д. Внутреннее устройство Windows: Windows Server 2003, Windows XP и Windows 2000. 4-е изд. - М.: Издательско-торговый дом "Русская Редакция"; СПб.: Питер; 2005. – 992 стр.: ил.

5. Таненбаум Э. Современные операционные системы, 3-е изд. : Пер. с англ. : СПб. : Изд-во Питер, 2010. – 1116 с., ил.

Полубенцева Марина Игоревна

МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ

Лицензия ЛР № 020593 от 07.08.97

Налоговая льгота – Общероссийский классификатор продукции

ОК 005-93, т. 2; 95 3005 – учебная литература

Подписано в печать 20.05.2008. Формат 60×84/16 Печать цифровая

Усл. печ. л. Уч.-изд. л. . Тираж . Заказ

Отпечатано с готового оригинал-макета, предоставленного автором

в цифровом типографском центре Издательства Политехнического

университета:

195251, Санкт-Петербург, Политехническая ул., 29.

Тел. (812) 540-40-14

Тел./факс: (812) 927-57-76