Министерство образования и науки Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

В.М. СТАСЫШИН

УПРАВЛЕНИЕ РЕСУРСАМИ В ОС WINDOWS

Утверждено Редакционно-издательским советом университета
в качестве конспекта лекций

НОВОСИБИРСК
2013

УДК 004.451.9(075.8)
С 779

Выполнено в рамках Программы стратегического развития НГТУ

Рецензенты

канд. техн. наук, доцент *Ю.В. Тракимус*
канд. техн. наук, доцент *Н.Л. Долозов*

Стасышин В.М.

С 779 Управление ресурсами в ОС Windows : конспект лекций / В.М. Стасышин. – Новосибирск : Изд-во НГТУ, 2013. – 104 с.

ISBN 978-5-7782-2303-5

Излагаются вопросы управления ресурсами в ОС Windows, входящие в программу курса «Управление ресурсами в вычислительных системах», читаемого студентам III курса специальности факультета прикладной математики и информатики Новосибирского государственного технического университета. Рассмотрены создание и управление потоками и процессами, описание структуры консольных и оконных приложений и используемых динамических библиотек; управление классами приоритетов потоков и процессов; синхронизация потоков и процессов посредством критических секций, мьютексов, событий и семафоров; обмен данными между параллельными процессами посредством каналов и почтовых ящиков.

Конспект лекций иллюстрирован примерами, способствующими успешному усвоению материала. Он может быть полезен и для специалистов, связанных с информационными технологиями, и самостоятельно осваивающих вопросы управления ресурсами в различных операционных системах.

Работа подготовлена на кафедре программных систем и баз данных для студентов III курса ФПМИ дневного отделения (направления 01040062, 01050062)

УДК 004.451.9(075.8)

ISBN 978-5-7782-2303-5

1. УПРАВЛЕНИЕ РЕСУРСАМИ В ОС WINDOWS

1.1. АРХИТЕКТУРА СИСТЕМЫ И ОБЪЕКТЫ ЯДРА

Аналогично ОС Unix программа может выполняться в одном из двух режимов:

- *Kernel* {*режим ядра*) или;
- *User* (*режим пользователя*).

Соответственно архитектура ОС Windows состоит из двух основных частей:

- привилегированной подсистемы режима ядра (*privileged kernel mode part*);
- непривилегированной подсистемы пользовательского режима (*nonprivileged user mode part*).

Подсистема ядра содержит следующие компоненты:

- HAL (Hardware Abstraction Layer) – уровень, абстрагирующий другие компоненты от аппаратных различий, зависимых от платформы;
- микроядро (MicroKernel), которое отвечает за планирование потоков, переключение задач, обработку прерываний и исключительных ситуаций, многопроцессорную синхронизацию (аналог подсистемы управления процессами в ОС Unix);
- драйверы устройств – драйверы оборудования, файловой системы и сетевой поддержки, реализующие пользовательские функции ввода-вывода;
- управление окнами и графическую п/с, реализующую функции графического интерфейса;
- исполнительную часть, отвечающую за управление памятью, управление процессами и потоками, безопасность, ввод-вывод данных и взаимодействие процессов.

Для управления системой и приложениями ресурсами (процессами, потоками, файлами, семафорами, событиями и пр.) используются так называемые **объекты ядра**.

Каждый объект ядра – это блок памяти, выделенный ядром и доступный только ему. Как и в ОС Unix, в блоке размещается структура

данных, поля которой содержат информацию об объекте. Некоторые поля (например, дескриптор объекта, дескриптор защиты и счетчик количества пользователей) присутствуют во всех объектах, но большая их часть специфична для объектов конкретного типа. Приложение не имеет прямого доступа к структурам данных, представляющим собой объекты ядра (аналогично ОС Unix). Оперировать объектами ядра приложение может только через специальные функции Windows (аналог системных вызовов в ОС Unix).

При вызове функции, создающей объект ядра (например, CreateFile), возвращается дескриптор созданного объекта (как и в ОС Unix). Этот дескриптор может быть использован любым потоком процесса. Значения дескрипторов объектов ядра действительны только в адресном пространстве процесса, их создавшего. Поэтому все попытки передать такой дескриптор другому процессу и использовать его в другом процессе приводят к ошибкам (аналогично ОС Unix).

Вместе с тем иногда возникает необходимость в совместном использовании объектов ядра *несколькими* процессами. Существуют три механизма, позволяющих процессам использовать одни и те же объекты ядра:

- наследование дескриптора объекта в дочернем процессе;
- именованные объекты;
- дублирование дескрипторов объектов.

Объекты принадлежат ядру, а не процессу (аналогично ОС Unix). Ядро отслеживает, сколько пользователей используют объект (*счетчик пользователей*). Когда к этому объекту обращается другой процесс, значение счетчика увеличивается на единицу, а когда какой-то процесс завершается, счетчики всех используемых им объектов ядра уменьшаются на единицу. Как только счетчик пользователей объекта обнуляется, ядро уничтожает этот объект.

Объекты ядра можно защитить с помощью дескриптора защиты (*security descriptor*). Он содержит информацию о том, кто создал объект и кто имеет права на доступ к нему.

Для задания дескриптора защиты функции, создающие объекты ядра, в качестве аргумента используют указатель на структуру **SECURITY_ATTRIBUTES** или **NULL**, и тогда объект создается с защитой по умолчанию.

Вне зависимости от того, как был создан объект ядра, после окончания работы с ним его нужно закрыть вызовом функции **CloseHandle**:

BOOL CloseHandle(*HANDLE hObject*);

Функция проверяет таблицу дескрипторов данного процесса, чтобы убедиться, что процесс имеет доступ к объекту **hObject**. Если доступ разрешен, то система получает адрес структуры данных объекта **hObject** и уменьшает в ней счетчик количества пользователей. Как только счетчик обнулится, ядро удаляет объект из памяти.

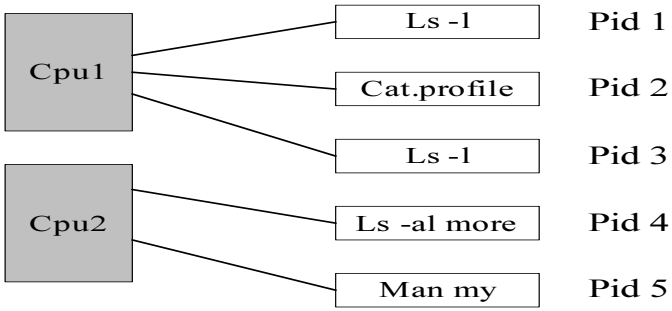
1.2. ПРОЦЕССЫ И ПОТОКИ

Понятия **потока** в классической операционной системе Unix нет. Вместе с тем понятие **потока** введено не разработчиками Microsoft. Появилось оно впервые у разработчиков СУБД более 20 лет назад.

Что представляет собой СУБД? СУБД – это фактически своя, в определенном смысле, операционная система со своими дисциплинами выделения памяти, буферизации, синхронизации, защиты данных, своими специфическими методами организации и хранения данных.

Операционная система – функционально универсальная система, используется для решения разных задач (вычислительные, работа с данными, графика, звук), алгоритмы управления ОС – тоже универсальные, а любые универсальные алгоритмы не могут быть одинаково оптимальны для всех задач, поскольку они всегда оптимальны в среднем.

Стандартная схема обработки процессов, связанная с переключением с одного контекста на другой и порождением и уничтожением процессов по мере их завершения, представлена ниже.



Для конкретной же области применения, каковой является СУБД, требуются более оптимальные алгоритмы для решения ограниченного спектра задач: СУБД лучше, чем операционная система, знает, как управлять буферами ОП (упреждающее чтение, насильственное удержание), как более эффективно организовать внешнюю память. Поэтому

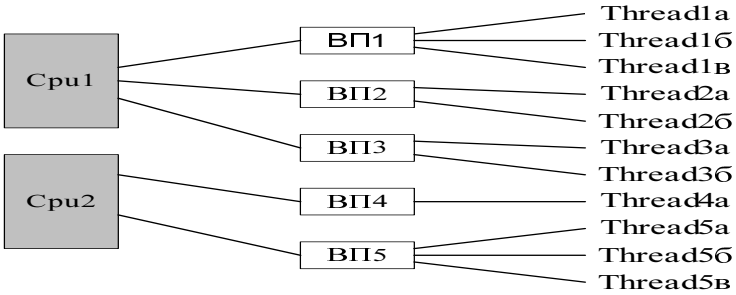
для разработчиков СУБД встала задача повышения эффективности управления ресурсами.

В предыдущих лекциях отмечалось, что переключение одного процессора с процесса на другой связано со сменой контекста процесса. Контекст же процесса очень тяжеловесен (пользовательский контекст, регистровый контекст, контекст системного уровня).

Поэтому появилась идея многопотоковой архитектуры.

Многопотоковая архитектура (DSA-Dynamic Scalable Architecture) в СУБД – способ организовать одновременную обработку множества задач внутри процесса сервера БД при минимальных накладных расходах на переключение контекста и максимальном разделении ресурсов, не доверяя средствам ОС (с целью сократить накладные расходы)

Поток (thread) – это фрагмент контекста одного процесса, включающий только те данные, которые необходимы для реализации выполняемых потоком функций. В рамках СУБД существует механизм виртуальных процессоров (ВП), который переключает потоки для обработки точно так же, как процессор переключает процессы.



Поскольку затраты ресурсов на запуск, останов, создание и уничтожение потока силами управляющего процесса гораздо ниже по сравнению с действиями операционной системы над обычными процессами, сокращается общее число процессов ОС, загружающих вычислительную систему. Для обеспечения параллелизма поток может быть клонирован на потоки более низкого уровня и т. д.

Такова общая идея понятия потока. Разработчики Microsoft подхватили идею потока, и потоки появились в операционной системе Windows.

При запуске приложения операционная система Windows создает процесс.

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

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

Все ресурсы, необходимые для исполнения процесса, называются **контекстом процесса**.

Кроме дескриптора, каждый процесс в ОС Windows имеет свой идентификатор, который уникален для процессов, выполняющихся в системе.

Смысл виртуального адресного пространства аналогичен ОС Unix, адресные процессы разных процессов (аналогично ОС Unix) не пересекаются, и процессы не имеют непосредственного доступа в адресное пространство других процессов (полная аналогия с ОС Unix).

Как уже было отмечено, сам по себе процесс не выполняется. Вместо этого он запускает единственный поток, который часто называют ***первичным потоком*** {**primary thread**). В консольном приложении – это поток, который выполняет функцию **main.** В приложениях с графическим интерфейсом – это поток, который исполняет функцию **WinMain**. Если процесс имеет только один первичный поток, то фактически понятия «процесс» и «поток» совпадают. Первичный поток может создавать другие потоки, те, в свою очередь, новые потоки и т. д.

Основной причиной введения понятия потока, как говорилось выше, служит то, что взаимодействие параллельных процессов требует **больших затрат на пересылку** данных, а это заметно замедляет работу приложений. Потоки же выполняются в адресном пространстве одного процесса, **могут обращаться к общим адресам памяти**, что упрощает их взаимодействие.

Потоком *в ОС Windows называется объект ядра, которому ОС выделяет* **процессорное время** *для выполнения приложения* (основная выполняемая единица).

Каждому потоку принадлежат следующие ресурсы:

- код исполняемой программы (функции);
- набор регистров процессора;
- стек для работы с приложением;
- стек для работы операционной системы;
- маркер доступа, который содержит информацию для системы безопасности.

Все эти ресурсы образуют **контекст потока** в ОС Windows (с точки зрения ядра – структура).

Кроме дескриптора каждый поток в Windows также имеет свой идентификатор, который уникален для потоков, выполняющихся в системе. Идентификатор позволяет пользователю отслеживать работу потоков.

В ОС Windows различают потоки двух типов:

- системные потоки;
- пользовательские потоки.

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

Пользовательские потоки служат для решения задач пользователя и запускаются приложением.

В работающем приложении различают потоки двух типов:

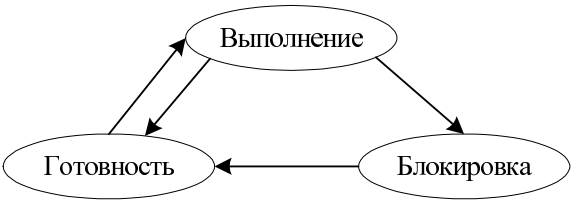
- рабочие потоки (working threads);
- потоки интерфейса пользователя (user interface threads).

Рабочие потоки выполняют различные фоновые задачи в приложении.

Потоки интерфейса пользователя связаны с окнами и выполняют обработку сообщений.

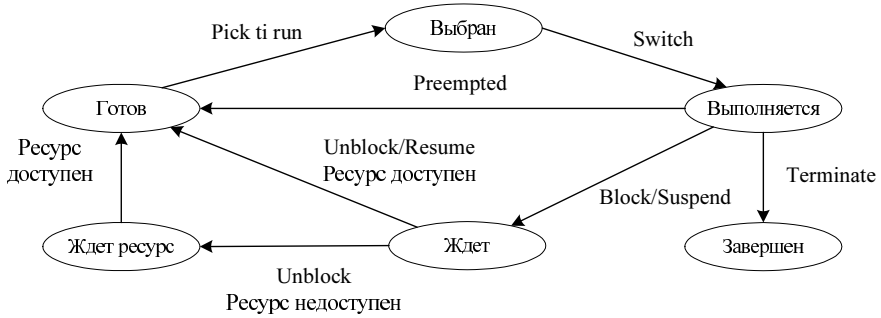
1.3. ПЛАНИРОВАНИЕ И ДИСПЕТЧЕРИЗАЦИЯ ПОТОКОВ

В простейшем варианте диаграмма состояний потока в ОС Windows аналогична диаграмме состояний процесса в ОС Unix.



В однопроцессорной системе в любой момент времени только один поток может находиться в состоянии выполнения. Все остальные потоки находятся либо в состоянии готовности, либо в состоянии блокировки.

Более подробно диаграмма состояний имеет вид:



Выделение центрального процессора потоку осуществляется в режиме разделения времени. Распределение квантов времени между потоками осуществляет **менеджер потоков**. Величина кванта времени, выдаваемого потоку, зависит от типа ОС Windows, типа процессора и приблизительно равна 20 мс.

Когда менеджер потоков переключает процессор на исполнение другого потока, выполняются следующие действия:

- сохранение контекста прерываемого потока;
- восстановление контекста запускаемого потока на момент его прерывания;
- передача управления запускаемому потоку (точно так же, как для процессов в ОС Unix).

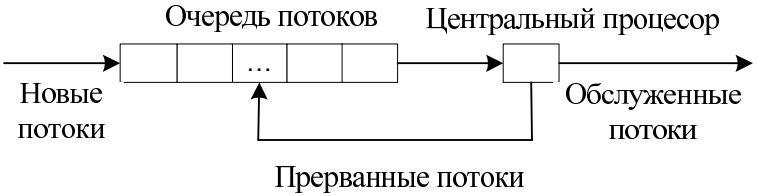
Алгоритм управления потоками должен оптимизировать следующие параметры системы:

- время загрузки микропроцессора работой должно быть минимальным;
- пропускная способность системы должна быть минимальной;
- время нахождения потока в системе должно быть минимальным;
- время ожидания потока в очереди должно быть минимальным;
- время реакции системы на обслуживание записи должно быть минимальным.

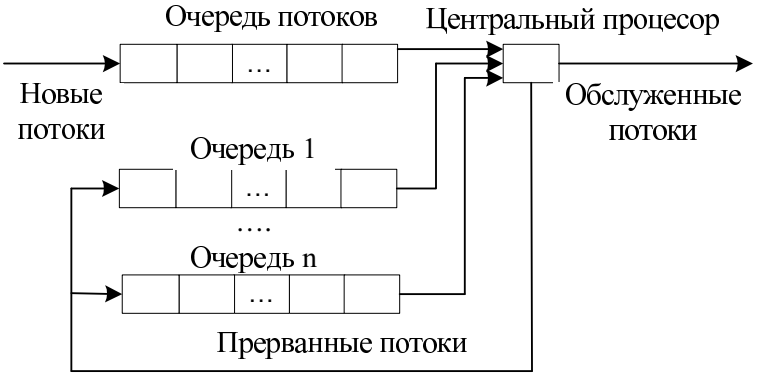
Существует несколько дисциплин управления потоками.

В предположении, что все потоки имеют одинаковый приоритет, поток попадает в единственную очередь готовых к выполнению потоков. Через какое-то время ОС выделяет в его распоряжение центральный процессор и поток переходит в состояние выполнения.

Идеальной дисциплиной обслуживания потоков процессором является дисциплина FIFO (First in – First out). Но поскольку незавершившиеся потоки блокируются до следующего обслуживания, а не уходят необслуженными, работает дисциплина FCFS (First come – First served).



Если потоки имеют разные приоритеты, то для управления ими используются дисциплины обслуживания с несколькими очередями. Каждая очередь включает потоки, которые имеют одинаковый приоритет. В системе Windows реализована система вытесняющего планирования на основе приоритетов. Это означает, что освободившийся процессор продолжает обслуживать тот поток из очереди, который обладает наибольшим приоритетом.



Когда после очередного прерывания квант потока становится равным нулю, ОС Windows помещает поток в состояние готовности в соответствующую очередь и ищет очередь с самым высоким приорите-

том, содержащую потоки, находящиеся в состоянии готовности. Если в состоянии готовности находятся несколько потоков с приоритетом не ниже предыдущего выполняемого потока, то первый поток из этой очереди будет переведен в состояние выполнения и начнет функционировать. Таким образом, потоки с одинаковым уровнем приоритета обслуживаются в циклическом порядке. При отсутствии других претендентов предыдущий поток может получить еще один квант.

Однако выполняемый поток не всегда полностью использует свой квант. Его выполнение может быть прервано при ненулевом кванте в двух ситуациях:

- когда появился в состоянии готовности другой поток с более высоким приоритетом; при этом текущий поток вытесняется и переводится в состояние готовности;
- текущему потоку потребовался какой-либо системный ресурс (или объект ядра), который в настоящий момент времени занят; в этом случае поток переводится в состояние блокировки (ожидания события).

Выбрав новый поток, операционная система переключает *контекст*. Эта операция заключается в сохранении содержимого регистров процессора для вытесненного потока и загрузке контекста для выбранного потока.

2. УПРАВЛЕНИЕ ПРОЦЕССАМИ И ПОТОКАМИ

2.1. СОЗДАНИЕ ПОТОКОВ

Любое приложение в ОС Windows после запуска реализуется как процесс. При инициализации программы пользователя ОС создает первичный (основной) поток, который исполняет код программы. В первичном потоке используется одна из функций: WinMain, wWinMain, main или wmain. Из основного потока при необходимости может быть запущен один или несколько вторичных потоков, которые выполняются одновременно с основным потоком.

Замечание. Поскольку задачей курса является изучение средств управления ресурсами в ОС Windows, а не вообще программирование под Windows, в большинстве случаев в качестве примеров будут приводиться main-программы.

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

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

При успешном завершении функция **CreateThread** возвращает дескриптор созданного потока и его идентификатор, который уникален для всей системы. В противном случае функция возвращает значение NULL.

*Параметры функции **CreateThread***

Параметр **lpThreadAttributes** устанавливает атрибуты защиты создаваемого потока. Установка этого параметра в NULL означает, что ОС сама установит атрибуты защиты потока, используя настройки по умолчанию (чаще всего так и делают).

Параметр **dwStackSize** определяет размер стека, который выделяется потоку при запуске. Если этот параметр равен нулю, то потоку выделяется стек, размер которого по умолчанию равен 1 Мбайт. Это наименьший размер стека, который может быть выделен потоку. Если величина параметра dwStackSize меньше значения, заданного по умолчанию, то все равно потоку выделяется стек размером в 1 Мбайт.

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

DWORD WINAPI имя_функции_потока (LPVOID lpParameter);

Видно, что функции потока может быть передан единственный параметр **lpParameter**, который является указателем на пустой тип. Это ограничение следует из того, что функция потока вызывается операционной системой, а не прикладной программой. Программы ОС являются исполняемыми модулями, и поэтому они должны вызывать только функции, сигнатура которых заранее определена.

Замечание. Функции потоков, вызываемые ОС, носят название **функций обратного вызова** (см. далее).

В отличие от входной функции первичного потока, имеющей одно из предопределенных имен: WinMain, wWinMain, main или wmain, – функции других потоков могут иметь произвольные имена.

Параметр **dwCreationFlags** определяет, в каком состоянии будет создан поток. Если значение этого параметра равно нулю (самая распространенная ситуация), то функция потока начинает выполняться сразу после создания потока (но вызывает функцию все равно ОС). Если же значение этого параметра равно CREATE SUSPEND, то поток создается в подвешенном состоянии. В дальнейшем этот поток можно запустить функцией **ResumeThread.**

Параметр **lpThreadid** является выходным, его значение устанавливает Windows. Этот параметр должен указывать на переменную, в которую Windows поместит идентификатор потока. Этот идентификатор уникален для всей системы и может в дальнейшем использоваться для ссылок на поток. Идентификатор потока используется системными функциями и редко функциями приложения.

В случае завершения потока сначала уничтожаются все User-объекты, принадлежащие потоку. После этого объект ядра «поток» переходит в свободное состояние, а счетчик пользователей объекта ядра «поток» уменьшается на единицу.

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

DWORD GetLastError(VOID);

Эта функция возвращает код последней ошибки, установленной в потоке.

Пример создания потока

```
#include <windows.h>
#include <iostream.h>
volatile int n;              // WINAPI – спецификатор соглашения о вызове,
DWORD WINAPI Add(LPVOID iNum);    // определяющий порядок передачи
                                                    // параметров
```

```
{                       // тип DWORD – 32-битное целое без знака
    cout << "Thread is started." << endl;
            n += (int)iNum;
            cout << "Thread is finished." << endl;
    return 0;
}
int main()
{
    int  inc = 10;
            HANDLE  hThread;  // дескриптор потока
            DWORD  IDThread; // идентификатор потока
            cout << "n = " << n << endl;
            // запуск потока Add
            hThread = CreateThread(NULL, 0, Add, (void*)inc, 0, &IDThread);
            if (hThread == NULL)
                    return GetLastError(); //   возвратиться с кодом последней ошибки
            // ожидание завершение потока Add (аналог системного вызова Wait в Unix)
            WaitForSingleObject(hThread, INFINITE);
            CloseHandle(hThread);        // закрытие дескриптора потока Add
            cout << "n = " << n << endl;
            return 0;
}
```

Пояснения к программе

1. **WINAPI** – спецификатор соглашения о вызове, определяющий порядок передачи параметров.

2. В функции main создается поток, в рамках потока запускается функция с именем **Add** (запускается сразу же, но запускает ее операционная система), ей передается адрес целочисленной передаваемой переменной **inc** (если нужно было бы передать несколько – или массив, или структура) и выводится **n**, увеличенная на величину переданного параметра.

3. И основной, и порожденный потоки могут иметь доступ к одним и тем же данным (**n**).

В этой программе следует обратить внимание на квалификатор типа **volatile**, который указывает компилятору, что значение переменной count должно храниться в памяти, поскольку к этой переменной имеют доступ параллельные потоки. Дело в том, что сам компилятор языка

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

4. Для ожидания завершения потока **Add** использована функция ожидания **WaitForSingleObject**, параметром которой является дескриптор объекта синхронизации (дескриптор потока) – аналог системного вызова **Wait()** в Unix. Значение второго параметра – **INFINITE** – определяет, что функция ждет перехода объекта синхронизации в сигнальное состояние бесконечно долго (пока не завершится порожденный поток).

5. Основной поток main отображает старое и новое значения переменной **n** и закрывает дескриптор.

2.2. ЗАВЕРШЕНИЕ ПОТОКА

Завершение потока можно организовать четырьмя способами:

- функция потока возвращает управление (этот способ использован выше);
- поток самоуничтожается вызовом функции **ExitThread**;
- один из потоков этого или другого процесса вызывает функцию **TerminateThread**;
- завершается процесс, содержащий данный поток.

Поток завершается вызовом функции **ExitThread**, которая имеет следующий прототип:

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

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

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

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

В случае успешного завершения функция **TerminateThread** возвращает ненулевое значение, в противном случае – **FALSE**. Функция **TerminateThread** завершает поток, но не освобождает все ресурсы, принадлежащие этому потоку. Поэтому эта функция должна вызываться только в крайних ситуациях при зависании потока.

Пример программы, которая демонстрирует работу функций **TerminateThread, Sleep**.

```
#include <windows.h>
#include <iostream.h>
volatile UINT count;
void WINAPI thread(void*)
{   for (;;)
            {  ++count;
            Sleep(100);   // функция приостанавливает выполнение потока на 100 мсек
            }
}
int main()
{
        HANDLE   hThread;
        DWORD  IDThread;
        char c;
        hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread,
                NULL, 0, &IDThread);      // создание потока
        if (hThread == NULL)
                return GetLastError();
   for (;;)
     {     cout << "Input 'y' to display the count or any char to finish: ";
                cin >> c;
                if (c == 'y')
                        cout << "count = " << count << endl;
                else
            break;
     }
```

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

Пояснения к программе:

1. В программе использована функция **Sleep**, которая имеет следующий прототип:

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

Единственный параметр функции – количество миллисекунд, на которые поток, вызвавший эту функцию, приостанавливает свое исполнение.

2. В рамках потока вызывается бесконечно работающая функция: увеличение переменной – пауза, и все сначала.

2.3. ПРИОСТАНОВКА И ВОЗОБНОВЛЕНИЕ ПОТОКА

Исполнение потока может быть приостановлено вызовом функции **SuspendThread**, которая имеет следующий прототип:

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

Каждый созданный поток имеет счетчик приостановок, максимальное значение которого равно **MAXIMUM_SUSPEND_COUNT**. Счетчик приостановок показывает, сколько раз исполнение потока было приостановлено. Поток может исполняться только при условии, что значение счетчика приостановок равно нулю. В противном случае поток не исполняется или, как говорят, находится в подвешенном состоянии.

Функция **SuspendThread** увеличивает значение счетчика приостановок на 1, и при успешном завершении возвращает текущее значение этого счетчика. В случае неудачи функция suspendThread возвращает значение, равное –1.

Поток может приостановить также и сам себя. Для этого он должен передать функции **SuspendThread** свой псевдодескриптор, который можно получить при помощи функции **GetcurrentThread**, имеющей прототип:

HANDLE GetCurrentThread(VOID);

и возвращающей псевдодескриптор текущего потока. **Псевдодескриптор** текущего потока отличается от настоящего дескриптора потока тем, что он может использоваться только самим текущим потоком и, следовательно, может наследоваться другими процессами. Псевдодескриптор потока не нужно закрывать после его использования.

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

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

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

Пример программы, использующей функции **SuspendThread, ResumeThread** и **Sleep**.

```
include <windows.h>
#include <iostream.h>
volatile UINT  nCount;
volatile DWORD dwCount;
void WINAPI thread(void*){
    for (;;)
    {   nCount++;
        Sleep(100);    // приостанавливаем поток на 100 миллисекунд
    }
}
int main()
{
            HANDLE   hThread;
            DWORD  IDThread;
            char c;
```

```
        hThread = CreateThread (NULL, 0, (LPTHREAD_START_ROUTINE)thread, NULL,
                                0, &IDThread);
        if (hThread == NULL)   return GetLastError();
    for (;;)
    {   cout << "Input :" << endl;
        cout << "\t'n' to exit" << endl;
        cout << "\t'y' to display the count" << endl;
        cout << "\t's' to suspend thread" << endl;
        cout << "\t'r' to resume thread" << endl;
        cin >> c;
        if (c = = 'n')     break;
        switch (c)
        {  case 'y':     cout << "count = " << nCount << endl;      break;
           case 's':     dwCount = SuspendThread(hThread);  // приостанавливаем
                                                            // поток thread
                         cout << "Thread suspend count = " << dwCount << endl;
                         break;
                         case 'r':    dwCount = ResumeThread(hThread);    // возобновляем поток
                                                                          // thread
                        cout << "Thread suspend count = " << dwCount << endl;
                        break;
        }
    }
    TeminateThread (hThread, 0)   // прерываем выполнение потока thread
    CloseHandle(hThread);         // закрываем дескриптор потока
    return 0;
}
```

Пояснения к программе

1. В порожденном потоке – бесконечный цикл.

2. В потоке **main** после создания дочернего потока в цикле запрашивается параметр, и в зависимости от введенного значения дочерний поток или приостанавливается, или возобновляется (или отображается параметр, или выход из цикла).

3. При выходе из цикла прерываем выполнение потока **thread**.

В заключение приведем пример оконного многопоточного приложения под Windows.

2.4. ОКОННОЕ ПРИЛОЖЕНИЕ ПОД WINDOWS

Ниже приводятся некоторые базовые понятия написания оконных Windows-приложений.

1. Структура Windows-приложений

Полноценная программа для Win32 должна содержать как минимум две функции:

- WinMain – главную функцию, в которой создается основное окно программы и запускается цикл обработки сообщений;
- оконную процедуру (имя – любое), обеспечивающую обработку сообщений для основного окна программы.

На некотором псевдоязыке каркас Windows-программы можно записать следующим образом:

```
WinMain (список аргументов)
{
Подготовить и зарегистрировать класс окна с требуемыми характеристиками;
Создать экземпляр окна зарегистрированного класса и отобразить его;
Пока не произошло необходимое для выхода событие
    {       Извлечь очередное сообщение из очереди сообщений;
            Передать его через ОС Windows оконной функции;
        }
Возврат из программы;
}
WndProc (список аргументов) – предполагаемое имя процедуры-
                                                    обработчика
{       Обработать полученное сообщение;
    Возврат;
    }
```

2. Концепция сообщений

В основе взаимодействия Windows-программы с внешним миром и ОС лежит концепция сообщений (вспомните понятие сообщения в ОС Unix). С точки зрения приложения сообщение – это уведомление о том, что произошло некоторое событие, которое может требовать, а может и не требовать выполнения определенных действий. Это событие может быть следствием действий пользователя, например, пере-

мещения курсора или щелчка кнопкой мыши, изменения размеров окна, выбора пункта меню. Кроме того, событие может генерироваться приложением, а также ОС.

С точки зрения ОС сообщение – структура данных, содержащая следующие элементы:

- дескриптор окна, которому адресовано сообщение;
- код (номер) сообщения;
- дополнительную информацию, зависящую от кода сообщения.

Сообщения, получаемые приложением, могут поступать асинхронно из разных источников. Например, приложение может работать с системным таймером, посылающим ему сообщения с заданным интервалом, и одновременно оно должно быть готовым в любой момент получить любое сообщение от ОС. Чтобы не допустить потери сообщений, Windows одновременно с запуском приложения создает глобальный объект, называемый **очередью сообщений приложения**. После вывода окна на экран (см. WinMain) любое действие пользователя ОС Windows должна преобразовать в сообщение, которое помещается **в очередь данного окна**.

Таким образом, путь следования сообщений, вызванных событиями (аппаратными и программными), можно представить так:

Аппаратное сообщение
Сообщение от ОС → Системная очередь сообщений → Очередь сообщений приложения
Сообщение от приложения

3. Функция WinMain

а) Заголовок функции выглядит следующим образом:

int WINAPI WinMain (HINSTANCE hlnstance,
HINSTANCE hPrevInstance, LPSTR IpCmdLine, int nCmdShow)

Функция возвращает значение типа int и принимает следующие параметры:

- *HINSTANCE hlnstance* – дескриптор, который присваивается запущенному приложению;
- *HINSTANCE hPrevInstance* – дескриптор предыдущего оконного приложения (обычно 0);
- *LPSTR IpCmdLine* – указатель на строку, в которую копируются аргументы приложения, если оно запущено в режиме командной строки.

Чаще всего программа запускается при помощи щелчка мышью на имени файла или на пиктограмме, а этот способ не предполагает передачи каких-либо аргументов в приложение. Но приложение можно запустить в режиме командной строки с помощью команды стартового меню **Пуск ►Выполнить** или в оболочке типа **Norton Commander**. При этом в командной строке набирается имя приложения, а после пробела указывается список аргументов, разделенных символом пробела;

- *int nCmdShow* – целое значение, которое может быть передано функции **ShowWindow**.

Оконное Windows-приложения включает следующие этапы:

- создание и регистрация класса окна;
- создание и отображение главного окна приложения;
- цикл обработки сообщений.

б) Создание и регистрация класса окна

Каждое сообщение, о которых говорилось выше, связано с конкретным окном, с каждым из которых связана собственная оконная процедура (процедура обработки сообщений).

Любое окно Windows принадлежит одному из существующих в данный момент в системе классов, который должен быть определен до отображения окна на экране. Класс окна задает основные свойства окон, например форму курсора, меню и пр. Другими словами, класс окна – это шаблон, в котором определяются выбранные стили, шрифты, заголовки, пиктограммы, размер.

```
WNDCLASS wcl;          // элемент структуры класса окна
HWND hWnd;             // дескриптор окна
                       // элементы структуры класса окна
wcl.hInstance = hThisInst;   // дескриптор экземпляра приложения,
                             // в котором находится процедура
                             // обработки
wcl.lpszClassName = szClassName; //указатель на строку,
                                 // содержащую имя класса
wcl.lpfnWndProc = WndProc;   // указатель на процедуру
                             // обработки сообщений
wcl.style = CS_HREDRAW | CS_VREDRAW;        // стиль класса окна
wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);   // дескриптор
                                               // пиктограммы
wcl.hCursor = LoadCursor(NULL ,IDC_ARROW);// дескриптор курсора
```

```
wcl.lpszMenuName = NULL;    // указатель на строку, содержащую
                            // имя меню
wcl.cbClsExtra = 0;         // дополнительные параметры
wcl.cbWndExtra = 0;         // дополнительные параметры
wcl.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
                                                  // цвет фона окна
RegisterClass(&wcl);        // регистрация класса окна
```

Комментарии к представленному коду

В строках 1 и 2 описаны классы окна и дескриптора окна.

Далее описываются поля структуры класса окна.

wcl.hInstance = hThisInst; дескриптор приложения задается как дескриптор текущего приложения:

wcl.lpszClassName = szClassName; настраивается указатель на строку – название класса окна приложения. Строка с названием может быть описана как LPCSTR szClassName = "Name";

wcl.lpfnWndProc = WndProc; – указывается функция-обработчик сообщений (упомянутая оконная процедура WndProc, функция обработки сообщений). В эту функцию будут поступать сообщения от системы. Подробнее о ней чуть позже.

wcl.style = CS_HREDRAW | CS_VREDRAW; – стиль окна. В данном случае окно будет перерисовываться, как только изменится один из его размеров.

wcl.hIcon = LoadIcon(NULL,IDI_APPLICATION); – указывается дескриптор пиктограммы (иконка, отображаемая в левой части заголовка окна: загружается некоторый ресурс пиктограммы – в данном случае ресурс по умолчанию).

wcl.hCursor = LoadCursor(NULL,IDC_ARROW); – указывается дескриптор курсора окна. Загружается некоторый системный курсор – стандартная стрелка).

wcl.lpszMenuName = NULL; – указывается, что меню в окне не предусмотрено.

wcl.cbClsExtra = 0; wcl.cbWndExtra = 0; дополнительные параметры не используются.

wcl.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);

указывается дескриптор кисти, используемой для закраски цвета фона окна (белая кисть).

в) Создание и отображение главного окна приложения

После регистрации класса окна нам необходимо создать окно, для этого используется функция CreateWindow. Прототип функции

```
HWND CreateWindow(
    LPCTSTR lpClassName,     // имя зарегистрированного класса
    LPCTSTR lpwindowName,    // имя окна
    DWORD dwStyle,           // стиль окна
    int x,                   // горизонтальная позиция
    int y,                   // вертикальная позиция
    int nWidth,              // ширина окна
    int nHeight,             // высота окна
    HWND hWndParent,         // дескриптор родительского окна
    HMEnu hMenu,             // дескриптор меню
    HINSTANCE hinstance,     // дескриптор экземпляра приложения
    LPVoid lParam            // указатель на данные, передаваемые
    );                       // в сообщение WM_CREATE
```

Пример вызова функции ***CreateWindow***

```
hWnd = CreateWindow(szClassName, szTitle,
   WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN |
   WS_CLIPSIBLINGS, 100, 50, 700, 120,
   HWND_DESKTOP, NULL, hThisInst, NULL);
```

Комментарии:

szTitle – адрес строки с названием окна, которое отображается в заголовке окна.

WS_OVERLAPPEDWINDOW – стандартный стиль.

HWND_DESKTOP – дескриптор окна-родителя (в данном случае Рабочего стола).

Меню окна нет (NULL).

hThisInst – дескриптор данного приложения.

Подробные параметры можно найти в справочной системе MSDN.

Отображение окна:

```
ShowWindow(hWnd, nWinMode);
UpdateWindow(hWnd);
```

В первый параметр – дескриптор окна, второй параметр определяет, в каком виде будет показано окно (четвертый параметр из ***WinMain***).

Замечание. В справочной системе MSDN рекомендуется после вызова ***ShowWindow*** вызвать ***UpdateWindow*****,** хотя это не обязательно.

г) Стандартный цикл обработки сообщений

После создания окна программа WinMain извлекает сообщения из очереди, выполняя блок команд, известный как «цикл обработки сообщений» (*message loop*):

```
While (GetMessage(&msg, NULL, 0, 0))
      { TranslateMessage(&msg);
  DispatchMessage(&msg); }
return msg.wParam;
```

В этом фрагменте кода переменная **msg** – это структура типа **MSG**, которая определена в заголовочных файлах Windows.

Извлечение очередного сообщения осуществляется с помощью функции **GetMessage**, имеющей прототип:

BOOL GetMessage (LPMSG IpMsg, HWND hWnd.,
UINT wMsgFilterMin, UINT wMsgFilterMax)

- параметр **IpMsg** задает адрес структуры типа **MSG**, в которую помещается выбранное сообщение;
- параметр **hWnd** содержит дескриптор окна, принимающего сообщение; обычно значение этого параметра равно NULL, что позволяет выбрать сообщение для любого окна приложения;
- параметры **wMsgFilterMin** и **wMsgFilterMax** указывают минимальный и максимальный номер принимаемого сообщения; обычно – 0, и функция выбирает из очереди *любое* очередное сообщение.

Функция **GetMessage** возвращает значение **TRUE** при извлечении любого сообщения, кроме одного – **WM_QUIT.** Получив сообщение **WM_QUIT**, функция возвращает значение **FALSE**. В результате этого происходит немедленный выход из цикла, и приложение завершает работу, возвращая операционной системе код возврата msg.wParam.

В теле цикла обработки сообщений – вызов двух функций: **TranslateMessage** и **DispatchMessage**.

Строго говоря, вызов **TranslateMessage** нужен только в тех приложениях, которые должны обрабатывать данные с клавиатуры: система генерирует сообщения **WM_KEYDOWN**, когда клавиша нажимается, и **WM_KEYUP**, когда клавиша отпускается. Назначение функции **TranslateMessage** – преобразовать пару этих сообщений в сообщение **WM_CHAR**, помещаемое в очередь.

Функция **DispatchMessage** передает структуру **msg** обратно в Windows. И Windows отправляет сообщение для его обработки соответствующей оконной процедуре, вызывая ее как функцию обратного вызова.

4. Оконная процедура WndProc

Реальная работа приложения осуществляется в оконной процедуре (windows-процедуре). **Оконная процедура WndProc** предназначена для обработки сообщений, адресованных любому окну того «оконного класса», в котором содержится ссылка на данную процедуру. Оконная процедура определяет то, что выводится в клиентскую область окна, и то, как окну реагировать на пользовательский ввод. Оконная процедура является так называемый «функцией обратного вызова»,

Понятие «функция обратного вызова». Так называют функции, которые вызывает сама ОС. Поэтому в коде приложения вы не найдете прямого вызова такой функции. Компилятор узнает функцию обратного вызова по спецификатору **CALLBACK**. Обычно оконная процедура имеет заголовок со стандартным синтаксисом:

LRESULT CALLBACK Имя_функции
(HWND HWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

LRESULT – тип возвращаемого значения,
hWnd – дескриптор окна, которому адресовано сообщение,
uMsg – код сообщения,
wParam и IParam – параметры сообщения. Имя функции может быть произвольным, но для главного окна приложения обычно используется имя **WndProc**.

В теле функции после объявления необходимых локальных переменных обычно размещается оператор **Switch**, внутри которого и происходит обработка нужных сообщений.

Каждому коду сообщения в Windows поставлен в соответствии уникальный символический идентификатор. Все системные идентификаторы определены в заголовочном файле winuser.h.

Некоторые сообщения Windows

Сообщение	Назначение
WM_CREATE	Посылается, когда приложение создает окно вызовом функции **CreateWindow** или **CreateWindowEx**. Оконная процедура получает это сообщение, когда окно уже создано, но еще не показано на экране. Поэтому, обрабатывая это сообщение, можно изменить характеристики окна, выполнить некоторые инициализирующие действия, например, открыть необходимые файлы, запустить новые потоки. После обработки сообщения приложение должно вернуть нулевое значение для продолжения процесса создания окна. Если приложение вернет значение –1, то окно не будет создано, а функция **CreateWindow** вернет значение NULL

Окончание таблицы

Сообщение	Назначение
WM_CLOSE	Уведомляет окно о том, что оно должно быть закрыто. Сообщение может быть использовано для вывода запроса пользователю с предложением подтвердить завершение работы
WM_DESTROY	Посылается оконной процедуре уничтожаемого окна, после того как окно удалено с экрана
WM_COMMAND	Сообщение посылается, когда пользователь выбирает команду меню или посылает команду из элемента управления
WM_PAINT	Посылается окну, содержимое которого требует перерисовки
WM_MOVE	Посылается окну, которое переместилось на экране
WM_SIZE	Посылается окну, размеры которого изменились

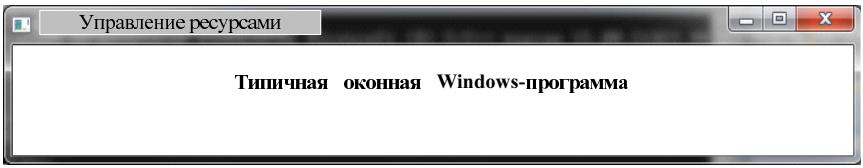
Кроме того, **сообщение с любым типом** может быть послано окну любой процедурой посредством функции **SendMessage**.

Пример оконной типичной Windows-программы.

```
////////////////////////////////////////////////////////////////////// filel.cpp
include <windows.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(HINSTANCE hlnstance, HINSTANCE hPrevInstance,
LPSTR IpCmdLine, int nCmdShow)
{
    char szClassName[] = "MyClass";
    MSG msg;
    WNDCLASS wcl;              // элемент структуры класса окна
    HWND hWnd;                 // дескриптор окна
    wcl.hInstance = hThisInst; // заполнение структуры класса окна
    wcl.lpszClassName = szClassName;
    wcl.lpfnWndProc = WndProc;
    wcl.style = CS_HREDRAW | CS_VREDRAW;
    wcl.hIcon = LoadIcon(NULL,IDI_APPLICATION);
    wcl.hCursor = LoadCursor(NULL,IDC_ARROW);
    wcl.lpszMenuName = NULL;
    wcl.cbClsExtra = 0;
    wcl.cbWndExtra = 0;
    wcl.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    RegisterClass(&wcl);              // регистрация класса окна
    hWnd = CreateWindow(szClassName, “Управление ресурсами”,
```

```
            WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN |
            WS_CLIPSIBLINGS, 100, 50, 700, 120,
            HWND_DESKTOP, NULL, hThisInst, NULL); //создание окна
    ShowWindow(hWnd, nWinMode);                    // отображение окна
    UpdateWindow(hWnd);
    While (GetMessage(&msg, NULL, 0, 0))  // цикл обработки сообщений
                { TranslateMessage(&msg);
                        DispatchMessage(&msg); }
        return msg.wParam;
}
LRESULT CALLBACK WndProc (HWND HWnd, UINT uMsg,
        WPARAM wParam, LPARAM lParam)
{               //   оконная процедура обработки сообщений WndProc
    HDC hDC;
    PAINTSTRUCT ps;
    static char text[100]= "Типичная оконная Windows-программа";
    switch (uMsg)
{           //      обработка сообщений
case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    TextOut (hDC, 10, 10, text, strlen(text));  // вывод строки в окно
    EndPaint(hWnd, &ps);
    break;
case WM_DESTROY:                    // при уничтожении окна
    PostQuitMessage(0);
    break;
default:
        return DefWindowProc(hWnd, Umsg, wParam, lParam);
}
return 0;
}
```

Результат:



Теперь объединим знания по написанию оконной Windows-программы со знаниями по созданию многопоточных программ.

Пример оконной Windows-программы для работы с потоками (текст программы приведен в сокращении).

```
# include <windows.h>
                                        ……..
# include "KWnd.h"          //   заголовочный файл используемого класса
enum UserMsg { UM_THREAD_D0NE = WM_USER+1 };
struct ThreadManager
                                        ……..   (элементы структуры опущены)
//   Структура, объекты которой tm_A, tm_B и tm_C
//   используются для взаимосвязи дочерних потоков с первичным потоком.
//   Адреса этих объектов передаются в качестве четвертого параметра при
//   вызовах функции CreateThread.
//   Описание прототипов используемых функций (опущены)
int WINAPI WinMain (HINSTANCE hinstance, HINSTANCE hPrevInstance,
                        LPSTR lpCmdLine, int nCmdShow)    //   главная функция
{
    MSG msg;
// определение объекта mainWnd класса KWnd, в котором создается и отображает окно
          KWnd mainWnd("CreateThreads", hinstance, nCmdShow,
                                    WndProc, NULL, 100, 100, 400, 160);
//        цикл обработки сообщений
          while (GetMessage(&msg, NULL, 0, 0))   //   извлечение очередного сообщения
            {  TranslateMessage(&msg);        //   преобразование сообщения
               DispatchMessage(&msg);  }          // передача структуры msg в Windows
    return (msg.wParam);
}
//   Оконная процедура WindProc
LRESULT CALLBACK WndProc (HWND HWnd, UINT uMsg,
                        WPARAM wParam, LPARAM lParam)   // оконная процедура WindProc
{
                                        ……...
    switch (uMsg)
//        обработка сообщений
    {
    case WM_CREATE:                //   создание трех дочерних потоков
```

```
        tm_A.hwndParent = hWnd;
        hThreadA = CreateThread(NULL, 0, ThreadFuncA, &tm_A, 0, NULL);
        if (!hThreadA)
            MessageBox (hWnd, "Error of create hThreadA", NULL, MB_OK);
//  потоки с идентификаторами hThreadB, hThreadC создаются аналогично
        break;
    case UM_THREAD_DONE:
//  обработка сообщения посланного функцией SendMessage дочерним потоком
            pTm = (ThreadManager*)wParam;
            sprintf(text, "%s: count = %d", pTm->name.c_str(), pTm->nValue);
//  Подготовка строки о завершившемся потоке
            y += 30;    //  координата строки вывода
            InvalidateRect (hWnd, NULL, FALSE);     //  генерация сообщения
                                                    //  WM_PAINT
            break;
    case WM_PAINT:
//  перерисовка окна – вывод строки
          hDC = BeginPaint(hWnd, &ps);
            TextOut (hDC, 20, y, text, strlen(text)); //  вывод строки в окно
            EndPaint(hWnd, &ps);
            break;
    case WM_DESTROY:
//  уничтожение дескрипторов потоков при закрытии окна
                    CloseHandle (hTreadA);
                    CloseHandle (hTreadB);
                    CloseHandle (hTreadC);
                    PostQuitMessage(0);
                    break;
    default:
            return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
    return 0;
}
//  Функции потоков ThreadFuncA, ThreadFuncB, ThreadFuncC
DWORD WINAPI ThreadFuncA(LPVOID lpv)
{       ThreadManager* pTm = (ThreadManager*)lpv;
        int count = 0;
```

```
        for (int i = 0;  i < 100000000;  ++i) count++;
        pTm->nValue = count:
        SendMessage (pTm->hwndParent, UM_THREAD_DONE, (WPARAM)pTm, 0);
// Посылка окну первичного потока пользовательского сообщения UM_THREAD_D0NE
    return 0:
}
// Функции потоков ThreadFuncB, ThreadFuncC  аналогичны, но  с другими
// параметрами цикла
```

Пояснения к программе

1. В программе используется созданный класс **KWnd**. Интерфейс класса **KWnd** описан в файле **KWnd.h**. Код, отвечающий за подготовку и создание окна, размещен в теле конструктора классса **KWnd** (файл **KWnd.cpp**). В нем описаны заголовок конструктора класса, тело конструктора класса, включающее код, отвечающий за подготовку, создание и отображение окна.

Оба этих вспомогательных файла, равно и как текст полной программы, можно найти в приложении 1.

2. В теле функции **WinMain** путем обращения к конструктору класса **KWnd** создается, регистрируется и отображается окно **mainWnd** класса, после чего остается только записать цикл обработки сообщений.

3. В оконной процедуре **WndProc** при обработке сообщения WM CREATE (при создании окна, но до его отображения) функцией **CreateThread** создаются три потока с идентификаторами **hThreadA**, **hThreadB**, **hThreadC**. Функции потоков **ThreadFuncA**, **ThreadFuncB**, **ThreadFuncC** работают по одному и тому же сценарию. В каждой из них есть локальный счетчик count, инкрементируемый в цикле for. Разница только в числе повторений цикла: 100 000 000, 50 000 000 и 20 000 раз соответственно. После завершения цикла каждая функция посылает окну первичного потока пользовательское сообщение **UM_THREAD_D0NE**.

4. Объекты **tm_A**, **tm_B** и **tm_C** структура **ThreadManager** используются для взаимосвязи дочерних потоков с первичным потоком. Адреса этих объектов передаются в качестве четвертого параметра при вызовах функции **CreateThread**.

5. Получив сообщение **UM_THREAD_DONE**, посланное очередным дочерним потоком, функция **WndProc** формирует строку

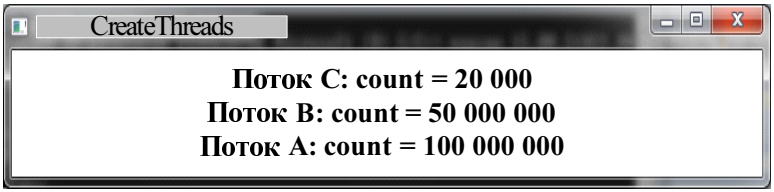
о завершившемся потоке вида: **Поток XX: count = XXXXXX** (значения **20 000, 50 000 000, 100 000 000** соответственно**).** После этого функция **InvalidateRect** генерирует сообщение **WM_PAINT**. Предварительно определяется координата вывода строки.

6. При обработке сообщения **WM_PAINT** выполняется вывод строки в окно.

7. При обработке сообщения **WM_DESTROY**, связанного с закрытием окна, уничтожаются дескрипторы потоков.

Хотя потоки запущены почти одновременно (порядок запуска: поток A, поток B, поток C), быстрее всех завершается поток C, затем поток B и последним – поток A. В этой последовательности информация о них и выводится в главное окно приложения.

Результат программы:



До сих пор мы говорили о потоках как «легковесных процессах», которые

- имеют доступ ко всей памяти процесса;
- общие данные не дублируются;
- переключение контекста быстрое.

2.5. Создание процесса

Отметим еще раз, что в Windows под **процессом** понимается объект ядра, которому принадлежат системные ресурсы, используемые исполняемым приложением. Поэтому в ОС Windows процесс и исполняемое приложение – синонимы. Выполнение каждого процесса начинается с первичного потока. Во время своего исполнения процесс может создавать другие потоки. Исполнение процесса заканчивается при завершении работы всех его потоков.

Кроме дескриптора, каждый процесс в ОС Windows (как и поток) имеет свой идентификатор, который уникален для процессов, выполняющихся в системе.

Новый процесс в ОС Windows создается вызовом функции **Create-Process** с прототипом:

```
BOOL CreateProcess (LPCTSTR lpApplicationName, // имя исполняемого
                                                // модуля
LPTSTR lpCommandLine,              // командная строка
LPSECDRITY_ATTRIBUTES lpProcessAttributes, // защита процесса
LPSECURITY_ATTRIBUTES lpThreadAttributes,  // защита потока
BOOL MnheritHandles,               // признак наследования дескриптора
DWORD dwCreationFlags,             // флаги создания процесса
LPVOID lpEnvironment,              // блок новой среды окружения
LPCTSTR lpCurrentDirectory,        // текущий каталог
LPSTARTUPINFO IpStartUpInfo,       // вид главного окна
LPPROCESS_INFORMATION lpProcessInformation); // информация
                                                // о процессе
```

Указать программу, запускаемую в качестве порождаемого процесса, можно двумя способами: заданием имени исполняемого модуля параметром **lpApplicationName;**

```
char lpszAppName[] = "C:\\ConsoleProcess.exe";

…… C:ConsoleProcess.exe

CreateProcess (lpszAppName, ……..     // создание нового процесса
```

или с использованием второго параметра **lpCommandLine**, задающего командную строку:

```
char lpszCommandLine[] = "C:WConsoleProcess.exe pi p2 p3";

……..

CreateProcess(NULL, IpszContmandLine, ……   // создание нового процесса
```

Отличие этого варианта от предыдущего состоит в том, что мы передаем системе имя нового процесса и его параметры через командную строку. В этом случае имя нового процесса может и не содержать полный путь к exe-файлу, а только имя самого exe-файла. В этом случае система для запуска нового процесса осуществляет поиск требуемого exe-файла в такой последовательности:

- каталог, из которого запущено приложение;
- системный каталог Windows;
- каталог Windows;

• каталоги, которые перечислены в переменной PATH среды окружения.

Пример запуска приложения Notepad.exe (Блокнот), с использованием командной строки.

```
#include <windows.h>
#include <iostream.h>
int main()
  { STARTUPINFO si;
    PROCESS_INFORMATION pi;
// заполнение нулями структуры STARTUPINFO по умолчанию
        ZeroMemory(&si, sizeof(STARTUPINFO));
        si.cb = sizeof(STARTUPINFO);
        if (!CreateProcess(                  // запуск процесса Notepad
            NULL,       // имя не задаем
            "Notepad.exe", // имя программы
            NULL,   // атрибуты защиты процесса устанавливаем по умолчанию
            NULL,       // атрибуты защиты первичного потока по умолчанию
            FALSE,      // дескрипторы текущего процесса не наследуются
            0,          // по умолчанию NORMAL_PRIORITY_CLASS
            NULL,       // используем среду окружения вызывающего процесса
            NULL,       // текущий диск и каталог, как и в вызывающем процессе
            &si,      // вид главного окна – по умолчанию
            &pi))       // информация о новом процессе
                  {     cout << "The mew process is not created." << endl
                    << "Check a name of the process." << endl;
                    return 0;
                    }
        Sleep(1000);              //     ожидание
        CloseHandle(pi.hThread);  //  закрытие дескрипторов запущенного процесса
        CloseHandle(pi.hProcess); //и потока в текущем процессе
        return 0;
}
```

Пояснения к программе

1. Перед запуском процесса Notepad все поля структуры **si** типа **STARTUPINFO** должны заполняться нулями. Это делается при помощи вызова функции **ZeroMemory**.

В этом случае вид главного окна запускаемого приложения определяется по умолчанию самой ОС Windows.

2. В параметре **dwCreationFiags** (шестой параметр) нулевое значение определяет по умолчанию класс порождаемого процесса **NORMAL_PRIORITY_CLASS** (см. далее).

3. Структура **pi** типа **PROCESS_INFORMATION** содержит идентификаторы и дескрипторы нового создаваемого процесса и его главного потока. В конце программы они закрываются.

4. Значение FALSE параметра **MnheritHandles** (пятый параметр) говорит о том, что эти дескрипторы не являются наследуемыми.

2.6. ЗАВЕРШЕНИЕ ПРОЦЕССА

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

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

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

Процесс может быть завершен другим при помощи вызова функции **TerminateProcess**, которая имеет следующий прототип:

BOOL TerminateProcess (*HANDLE hProcess,* *// дескриптор процесса*

UINT uExitCode *// код возврата*);

Нормальное завершение функции **TerminateProcess –** 0, при ошибке – FALSE. Функция **Terminate Process** завершает работу процесса, но не освобождает ресурсы, принадлежащие этому процессу. Поэтому эта функция должна вызываться только в аварийных ситуациях при зависании процесса.

Пример программы с функцией **TerminateProcess**. Считаем, что имеется бесконечный процесс-счетчик **ConsoleProcess.exe**, расположенный на диске С:

```
#include <windows.h>
#include <iostream.h>
int count;
```

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

Ниже приведена программа, которая создает этот бесконечный процесс-счетчик, а потом завершает его по требованию пользователя, используя для этого функцию **TerminateProcess**.

```
#include <windows.h>
#include <conio.h>
int main()
{       char lpszAppName[] = "C:\\ConsoleProcess.exe";
          STARTUPINFO si;
          PROCESS_INFORMATION pi;
          ZeroMemory(&si, sizeof(STARTUPINFO));
          si.cb=sizeof(STARTUPINFO);
        // создание нового консольного процесса
          if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
                                CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi))
                    {    _cputs("The new process is not created.\n");
                         _cputs("Press any key to finish.\n");  _getch();   return 0;   }
          _cputs("The new process is created.\n");
           while (true)
          { char c;
                    _cputs("Input 't' to terminate the new console process: ");
                    c = _getch();
                    if (c == 't')
         { _cputs("t\n  TerminateProcess(pi.hProcess,1);// завершение процесса
                                                                    break;   }
           }
          CloseHandle(pi.hThread);    //   закрытие дескрипторов нового процесса
          CloseHandle(pi.hProcess);   //   в текущем процессе
          return 0;
}
```

Пояснения к программе

1. В отличие от предыдущего примера имя исполняемого модуля задано первым параметром функции **CreateProcess** – параметром **lpApplicationName**.

2. В параметре **dwCreationFiags** (шестой параметр) установлен флаг **CREATE_NEW_CONSOLE**. Это говорит о том, что для запускаемого процесса должна быть создана новая консоль.

3. При вводе в основном процессе значения 't' порожденный дочерний процесс завершается функцией **TerminateProcess.**

2.7. КЛАССЫ ПРИОРИТЕТОВ ПРОЦЕССОВ И ПРИОРИТЕТЫ ПОТОКОВ

Операционная система Windows поддерживает 32 приоритета (от 0 до 31) – чем больше номер, тем выше приоритет.

Классы приоритетов процессов:

Класс	Флаг в функциях CreateProcess или SetPrioriryClass	Базовый уровень
Idle	IDLE_PRIORITY_CLASS	4
Below normal	BELOE_NORMAL_PRIORITY_CLASS	6
Normal	NORMAL_PRIORITY_CLASS	8
Above normal	ABOVE_NORMAL_PRIORITY_CLASS	10
High	HIGH_PRIORITY_CLASS	13
Realtime	REALTIME PRIORITY CLASS	24

Класс фоновых процессов Idle назначается процессу, который должен простаивать в случае активности других процессов, например, для приложения – хранителя экрана.

Процессам, запускаемым пользователем, по умолчанию присваивается класс **Normal**. Это самые многочисленные процессы в системе.

Классы **Below normal** и **Above normal**, устанавливаемые флагами **BELOE_NORMAL_ PRIORITY_CLASS** и **ABOVE_NORMAL_ PRIORITY_CLASS**, соответственно немного повышают или понижают приоритет пользовательского процесса.

Процессы с высоким приоритетом, относящиеся к классу **High**, – это такие пользовательские процессы, от которых требуется более быстрая реакция на некоторые события, чем от обычных пользовательских процессов. Обычно с классом **High** работают некоторые системные процессы, которые большую часть времени ожидают какого-либо события, например, **winlogon.exe.** Эти процессы должны содержать небольшой программный код и выполняться очень быстро, чтобы не замедлять работу системы. Если в приложении какая-то подзадача требует быстрой реакции на некоторое событие, то можно повышать

класс приоритета процесса до значения **High** именно на тот период, когда решается эта подзадача, а затем возвращать его к значению **Normal.**

Практически никогда не следует использовать класс приоритета **Realtime**, поскольку в этом случае ваше приложение будет прерывать системные потоки, управляющие мышью, клавиатурой и дисковыми операциями. Система будет фактически парализована. Только в особых случаях, когда программа взаимодействует непосредственно с аппаратурой или решаются короткие подзадачи, для которых нужно гарантировать отсутствие прерываний, может быть кратковременно использован класс приоритета **Realtime**. Работа таких процессов обычно происходит в масштабе реального времени и связана с реакцией на внешние события.

Для изменения класса приоритета процесса во время работы приложения может применяться функция **SetPrioriryClass**, которая имеет следующий прототип:

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

При успешном завершении функция **SetPriorityClass** возвращает – 0, в противном случае значение – FALSE. Параметр **dwPriorityClass** этой функции должен быть равен одному из флагов, которые приведены выше.

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

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

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

Пример манипуляции приоритетами процесса:

```
#include <windows.h>
#include <conio.h>
int main()
{   HANDLE  hProcess;
    DWORD  dwPriority;
    hProcess = GetCurrentProcess();  //   получение псевдодескриптора
                                     //   текущего процесса
```

```
    dwPriority = GetPriorityClass(hProcess);  //  получение приоритета
                                              //  текущего процесса
    _cprintf("The priority of the process = %d.\n", dwPriority);
// устанавливается фоновый приоритет текущего процесса
    if (!SetPriorityClass(hProcess, IDLE_PRIORITY_CLASS))
    {   _cputs("Set priority class failed.\n");
        _cputs("Press any key to exit.\n");  _getch();  return GetLastError();  }
    dwPriority = GetPriorityClass(hProcess);  //  получение приоритета
                                              //  текущего процесса
    _cprintf("The priority of the process = %d.\n", dwPriority);
// устанавливается высокий приоритет текущего процесса
    if (!SetPriorityClass(hProcess, HIGH_PRIORITY_CLASS))
    {   _cputs("Set priority class failed.\n");
        _cputs("Press any key to exit.\n");    _getch();  return GetLastError();  }
    dwPriority = GetPriorityClass(hProcess);
   _cprintf("The priority of the process = %d.\n", dwPriority);
    _cputs("Press any key to exit.\n");    _getch();
    return 0;
}
```

Пояснения.

1. В примере меняется приоритет текущего процесса. Для этого предварительно функцией **GetCurrentProcess** с прототипом:

HANLDE GetCurrentProcess(VOID);

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

2. В программе последовательно устанавливается фоновый приоритет (IDLE_PRIORITY_CLASS), затем высокий приоритет процесса (HIGH_PRIORITY_CLASS).

Приоритет потока определяется на основе приоритета процесса. Для каждого приоритета потока существует очередь потоков. Приоритет потока складывается из двух составляющих:

- класса приоритета процесса, его создавшего;
- относительного приоритета потока внутри этого класса.

По умолчанию создаваемый поток получает *базовый приоритет* в соответствии с классом своего процесса. После создания потока его

приоритет может изменяться как операционной системой, так и приложением с помощью функции **SetThreadPriority** с прототипом

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

Относительное значение приоритета определяется вторым параметром функции **SetThreadPriority**, и их можно разбить на две группы.

Первая группа:

- THREAD_PRIORITY_LOWEST – низший приоритет;
- THREAD_PRIORITY_BELOW_NORMAL – приоритет ниже нормального;
- THREAD_PRIORITY_NORMAL – нормальный приоритет;
- THREAD_PRIORITY_ABOVE_NORMAL – приоритет выше нормального;
- THREAD_PRIORITY_HIGHRST – высший приоритет.

Вторая группа:

- THREAD_PRIORITY_IDLE – приоритет фонового потока;
- THREAD_PRIORITY_TIME_CRITICAL – приоритет потока реального времени.

Относительный приоритет	**Флаг в функции SetThreadPriority**	**Описание действия флага**
Idle	THREAD_PRIORIRY_IDLE	Для процессов класса Realtime приоритет потока равен 16, для процессов остальных классов – 1
Lowest	THREAD_PRIORIRY_LOWEST	Приоритет потока меньше базового приоритета на 2
Below normal	THREAD_PRIORIRY_BELOW_ NORMAL	Приоритет потока меньше базового приоритета на 1
Normal	THREAD_PRIORIRY_NORMAL	Приоритет потока равен базовому приоритету
Above normal	THREAD_PRIORIRY_ABOVE_ NORMAL	Приоритет потока больше базового приоритета на 1
Highest	THREAD_PRIORIRY_HIGHEST	Приоритет потока больше базового приоритета на 2
Time critical	THREAD_PRIORIRY_TIME_ CRITICAL	Для процессов класса Realtime приоритет потока равен 31, для процессов остальных классов – 15

В первом столбце – названия приоритетов потоков. Правила их назначения следующие:

Значение флага из первой группы в сумме с приоритетом процесса, в контексте которого этот поток выполняется, уменьшает, оставляет неизменным или увеличивает значение базового приоритета потока соответственно на величину –2, – 1, 0, 1, 2.

Значение флага **THREAD_PRIORITY_IDLE** устанавливает приоритет потока равным 16, если приоритет процесса, в контексте которого выполняется поток, равен Realtime, и 1 – в остальных случаях. Значение флага **THREAD_PRIORITY_TIME_CRITICAL** устанавливает приоритет потока равным 31, если приоритет процесса, в контексте которого выполняется поток, равен Realtime, и 15 – в остальных случаях.

Ниже приведены базовые приоритеты потоков в зависимости от приоритета процесса и относительного значения приоритета потока (параметра функции **SetThreadPriority**).

		Классы процессов					
		Real time	High	Above normal	Normal	Below normal	Idle
Относительное значение приоритета потока (параметр функции **SetThread-Priority**)	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

При удачном завершении функция **SetThreadPriorit**y возвращает ненулевое значение, в противном случае – FALSE. Параметр **nPriority** этой функции должен быть равен одному из перечисленных уровней приоритетов.

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

DWORD GetThreadPriority (HANDLE hThread); // дескриптор потока

При успешном завершении функция возвращает одно из значений уровня приоритета, в противном случае функция **GetThreadPriority** возвращает значение **THREAD_PRIORITY _ERROR_RETURN**.

Пример манипуляции уровнем приоритета потоков

```
#include <windows.h>
#include <conio.h>
int main()
{  HANDLE  hThread;
   DWORD  dwPriority;
   hThread = GetCurrentThread();       //   получение псевдодескриптора
                                       //   текущего потока
   dwPriority = GetThreadPriority(hThread);  // получение уровня приоритета
                                             //   текущего процесса
   _cprintf("The priority level of the thread = %d.\n", dwPriority);
// понижение приоритета текущего потока
    if (!SetThreadPriority(hThread, THREAD_PRIORITY_LOWEST))
   {      _cputs("Set thread priority failed.\n");
     _cputs("Press any key to exit.\n");
     _getch();
     return GetLastError();  }
   dwPriority = GetThreadPriority(hThread);  // получение уровня приоритета
                                             // текущего процесса
   _cprintf("The priority level of the thread = %d.\n", dwPriority);
// повышение приоритета текущего потока
    if (!SetThreadPriority(hThread, THREAD_PRIORITY_HIGHEST))
   {      _cputs("Set thread priority failed.\n");
     _cputs("Press any key to exit.\n");
     _getch();
     return GetLastError();  }
   dwPriority = GetThreadPriority(hThread);   // получение уровня приоритета
                                             // текущего процесса
  _cprintf("The priority level of the thread = %d.\n", dwPriority);
  _cputs("Press any key to exit.\n");
  _getch();
  return 0;
}
```

Пояснения к программе

1. Функция **GetCurrentThread** использована для получения псевдодескриптора текущего потока.

2. В программе последовательно устанавливается приоритет потока меньше базового приоритета процесса на 2 (**THREAD_ PRIORITY_ LOWEST**) и больше базового приоритета процесса на 2 (**THREAD_ PRIORITY_HIGHEST**).

Базовый приоритет потока может изменяться системой динамически, если этот приоритет находится в пределах между уровнями от 0 до 15. При получении потоком сообщения или при его переходе в состояние готовности система повышает базовый приоритет этого потока на 2. В процессе выполнения с каждым отработанным квантом времени базовый приоритет такого потока понижается на 1, но никогда не опускается ниже исходного базового приоритета.

Программные средства ОС Windows позволяют программно управлять режимом динамического изменения базовых приоритетов потоков.

Функции **GetProcessPriorityBoost**, и **GetThreadPriorityBoost** позволяют определить, разрешен ли режим динамического повышения базовых приоритетов всех потоков указанного процесса и отдельного потока соответственно, а функции **SetProcessPriorityBoos**t и **SetThreadPriorityBoost** служат для установки или отмены режима динамического повышения базовых приоритетов всех потоков указанного процесса и отдельного потока соответственно. Первый параметр функций – дескриптор процесса или потока, второй – **TRUE** для запрета повышения динамического приоритета и **FALSE** – для разрешения повышения динамического приоритета.

Пример управления динамическим изменением приоритетов потоков

```
#include <windows.h>
#include <conio.h>
int main()
{
        HANDLE  hProcess, hThread;
        BOOL    bPriorityBoost;
   hProcess = GetCurrentProcess();  //   получение псевдодескриптора
                                    //   текущего процесса
// получение режима динамического повышения приоритетов для процесса
```

```
    if (!GetProcessPriorityBoost(hProcess, &bPriorityBoost))
                    {   < обработка ошибок>    return GetLastError();  }
          _cprintf("The process priority boost = %d.\n", bPriorityBoost);
// выключение режима динамического повышения приоритетов для процесса
    if (!SetProcessPriorityBoost(hProcess, TRUE))
                    {   < обработка ошибок>    return GetLastError();  }
    hThread = GetCurrentThread();   //   получение псевдодескриптора
                                    //   текущего потока
 // получение режима динамического повышения приоритетов для потока
    if (!GetThreadPriorityBoost(hThread, &bPriorityBoost))
                    {   < обработка ошибок>    return GetLastError();  }
    _cprintf("The thread priority boost = %d.\n", bPriorityBoost);
// включаем режим динамического повышения приоритетов для потока
          if (!SetThreadPriorityBoost(hThread, FALSE))
                    {   < обработка ошибок>    return GetLastError();  }
          _cputs("Press any key to exit.\n");_getch();
          return 0;
}
```

2.8. ДИНАМИЧЕСКИ ПОДКЛЮЧАЕМЫЕ БИБЛИОТЕКИ (DLL)

Преимущества DLL

1. Расширение функциональности приложения.

DLL можно загружать в адресное пространство процесса динамически, что позволяет приложению, определив, какие действия от него требуются, подгружать нужный код.

2. Возможность использования разных языков программирования.

Благодаря DLL можно написать, например, интерфейс на Visual Basic, математическое ядро на Fortran-е, логику на C++ и все это «связать между собой» с помощью DLL.

3. Простота управления проектом.

Если отдельные модули программного продукта создаются отдельными группами, то при использовании DLL таким проектом управлять гораздо проще.

4. Экономия памяти.

Если одну и ту же DLL использует несколько экземпляров приложений, в оперативной памяти может храниться только один ее экземпляр, доступный этим приложениям. Например: DLL версия библиотеки C++.

5. Разделение ресурсов.

DLL могут содержать такие ресурсы, как шаблоны диалоговых окон, строки, значки и битовые карты (растровые изображения).

6. Реализация специфических возможностей.

Определенная функциональность в Windows доступна только при использовании DLL. Например, отдельные виды ловушек.

Динамически подключаемые библиотеки DLL представляют собой набор модулей исходного кода, в каждом из которых содержится определенное число функций (одна или несколько), вызываемых приложением (исполняемым файлом) или другими DLL.

Последовательность действий при создании DLL

1. Подготовить заголовочный файл с прототипами, структурами и идентификаторами, экспортируемыми из DLL. Этот файл включается в исходный текст всех модулей вашей DLL.

2. Написание на C/C++ модулей исходного кода с телами функций и определениями переменных, которые должны находиться в DLL.

3. Компилятор преобразует исходный код модулей DLL в OBJ-файлы.

4. Компоновщик собирает все OBJ-модули в единый загрузочный DLL-модуль (библиотеку DLL), в котором будет размещен двоичный код и переменные, относящиеся к данной DLL.

5. Если компоновщик обнаружит, что DLL экспортирует хотя бы одну переменную или функцию, он создаст LIB-файл, в котором содержится список символьных имен и переменных, экспортируемых из DLL.

Пример создания программы динамической библиотеки, включающей одну функцию:

```
#include <stdio.h>
extern "C" __declspec(dllexport) int Information(char *InfoString)
{
        sprintf(InfoString," Управление ресурсами");
        return 0;
}
```

Предполагается, что в единственную функцию DLL с именем **Information** передается параметр – строка **InfoString**.

Важным моментом является описание функции. Функция должна быть описана как внешняя, реализованная на языке Си, и специфицирована как экспортируемая функция динамической библиотеки.

Наличие модификатора **__declspec(dllexport)** перед переменной, прототипом функций или классом C++ заставляет компилятор Microsoft C/C++ встраивать в конечный OBJ-файл дополнительную информацию, что необходимо компоновщику при сборке DLL из OBJ-файлов.

Файлы с исходным кодом компилируются и компонуются так же, как и при создании EXE-файла. Но, создавая DLL, необходимо указывать компоновщику ключи /DLL или LD + функции библиотеки Windows API.

Пример компиляции библиотеки DLL:

cl /LD info.cpp kernel32.lib user32.lib gdi32.lib advapi32.lib

Результатом компиляции **info.cpp** является динамическая библиотека **info.dll**.

Существует достаточно много способов взаимодействия основного приложения и динамической библиотеки (динамическое связывание во время загрузки, динамическое связывание во время выполнения, отложенная загрузка…). Наиболее простым является явный способ подключения такой библиотеки.

```
typedef int (*ImportFunction)(char *);      // определен тип указателя на
                                            // функцию
char String[256];       // параметр (символьный массив, передаваемый в
                        // функцию DLL)
ImportFunction DLLInfo;             // указатель на функцию
HINSTANCE hinstLib = LoadLibrary(TEXT("info.dll"));      // загрузка
                                    // динамической библиотеки info.dll
DLLInfo = (ImportFunction)GetProcAddress(hinstLib, "Information");
                // загрузка функции Information
DLLInfo(String);        // вызов функции из DLL с передачей
                        // ей параметра
FreeLibrary(hinstLib);    // освобождения дескриптора DLL
```

В первой строке определен тип указателя на функцию. Во второй строке листинга объявлен символьный массив, который будет передан в функцию динамической библиотеки. Затем в третьей строке объявлена переменная – указатель на функцию. В строке четыре происходит загрузка динамической библиотеки **info.dll**, текст которой был приведен выше. Затем полученный в строке четыре дескриптор динамиче-

ской библиотеки используется для загрузки функции **Information**, определенной в динамической библиотеке (строка пятая). После этого в строке шестой выполняется вызов функции из динамической библиотеки. При вызове функции передается параметр **String** – символьный массив. В седьмой строке динамическая библиотека закрывается посредством освобождения дескриптора.

Схема построения консольного приложения в расчетно-графической работе с использованием DLL:

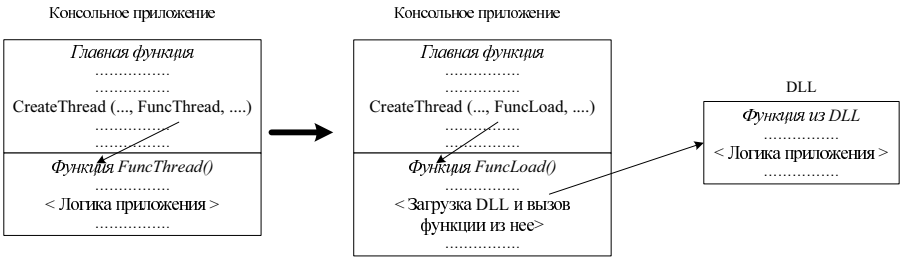
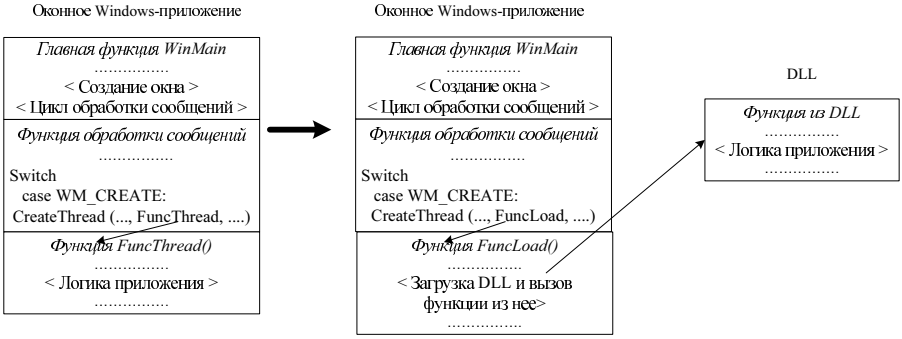


Схема построения Windows-приложения в расчетно-графической работе с использованием DLL:



3. СИНХРОНИЗАЦИЯ ПОТОКОВ И ПРОЦЕССОВ

При разбиении приложения на отдельные потоки возникает задача синхронизации потоков. Простейшие примеры необходимости синхронизации:

- использование потоками общих глобальных переменных;
- параллельное выполнение ввода-вывода.

Пример простой операции считывания и записи в общую глобальную переменную из нескольких потоков без должной синхронизации, которая может быть источником ошибок в работе программы. Программа оформлена как оконное приложение. Программа приводится в сокращении, полный ее текст – в Приложении 2.

```
#include ……...          //   необходимые заголовочные файлы
#include "KWnd.h"
#define N 50000000
long g_counter = 0;
//   Описание прототипов используемых функций (опущены)
int WINAPI WinMain (HINSTANCE hinstance, HINSTANCE hPrevInstance,
                                LPSTR IpCmdLine, int nCmdShow)   //   главная функция
{
    MSG msg;
//   определение  объекта  mainWnd класса KWnd, в котором
//   создается и отображается окно
   KWnd mainWnd("BadCount", hinstance, nCmdShow, WndProc, NULL, 100, 100, 400, 100);
         while (GetMessage(&msg, NULL, 0, 0))    // извлечение очередного сообщения
           {  TranslateMessage(&msg);         //   преобразование сообщения
              DispatchMessage(&msg);  }          // передача структуры msg в Windows
    return (msg.wParam);
}
//   Оконная процедура WndProc
LRESULT CALLBACK WndProc (HWND HWnd, UINT uMsg,
                  WPARAM wParam, LPARAM lParam)  // оконная процедура WindProc
{   ……...                                                  //   необходимые описания
        char text[100];
    switch (uMsg)                                 //   обработка сообщений
    {   case WM_CREATE:                           //   создание нового потока
                hThread = CreateThread(NULL, 0, ThreadFunc, &tm_A, 0, NULL);
                if (!hThread)
                    MessageBox (hWnd, "Error of create hThread", NULL, MB_OK);
                DecCounter();
                WaitForSingleObject (ChThread, INFINITE);
```

```
//   приостановка выполнения потока на неограниченное время (IN FINITE)
                        InvalideRect (hWnd, NULL, TRUE);
                        break;
            case WM_PAINT:          //   перерисовка окна
                        hDC = BeginPaint(hWnd, &ps);
                        sprintf(text, "g_counter = %d", g_counter);
                        TextOut(hDC, 20, 20, text, strlen(text));
                        EndPaint(hWnd, &ps);
                        break;
            case WM_DESTROY:        //   закрытие окна
              ……...
            default:
                        return DefWindowProc(hWnd, uMsg, wParam, lParam);
            }
            return 0;
}
//////////////////////////////////////////////////////////////
DWORD WINAPI ThreadFunc (LPVOID lpv)        //  функция дочернего потока
{   IncCount();
    Return 0;   }
void IncCount()                 //   функция, вызываемая дочерним потоком
{   for (int i = 0; I < N; ++)  ++g_counter;    }
void DecCount()                 //     функция, вызываемая основным потоком
{   for (int i = 0; I < N; ++)  --g_counter; }
```

Пояснения к программе

Первичный поток создает с помощью **CreateThread** дочерний поток, имеющий входную функцию **ThreadFunc**. В теле функции **ThreadFunc** вызывается функция **IncCount**, которая выполняет в цикле N раз операцию увеличения для глобального счетчика **g_counter**.

Запустив дочерний поток и не дожидаясь окончания его выполнения, первичный поток вызывает функцию **DecCount**. Функция **DecCount** выполняет в цикле N раз операцию уменьшения для глобального счетчика **g_counter**.

После возврата из **DecCount** первичный поток ждет окончания работы дочернего потока с помощью функции **WaitForSingleObject**. Вызов этой функции обеспечивает приостановку выполнения потока на неограниченное время (INFINITE), до тех пор, пока не освободится объект ядра hThread.

После этого, вызывая функцию **InvalidateRect**, первичный поток заставляет Windows сформировать сообщение **WM_PAINT**. Обрабатывая это сообщение, функция **WndProc** выводит значение переменной **g_counter** в главное окно приложения. Нетрудно видеть, что правильным результатом выполнения этого приложения должно быть значение **g_counter = 0**.

При маленьких значениях N так и происходит. Но вот для значения 50 000 000 эта программа дает самые разные результаты на разных компьютерах. Можно посчитать, что при тактовой частоте 2 ГГц функция **IncCount**, так же как и функция **DecCount**, требует для своего выполнения примерно 150 мс. При такой продолжительности выполнения каждый поток будет прерываться не менее пяти раз, отдавая процессор другому потоку.

И результат был иногда ненулевой, иногда нулевой.

Чем же вызвано искажение результата работы программы?

Допустим, что функция **DecCount** первичного потока приступила к очередному вычитанию единицы из счетчика. Процессор, реализуя эту операцию, скопировал значение глобальной переменной **g_counter** в свой регистр. Предположим, что это значение равно 100 000. Далее процессор успел вычесть единицу, **но не успел переписать новое значение 99 999 обратно в глобальную переменную**. В этот момент система отбирает процессор у первичного потока, сохранив контекст потока, и передает его на использование дочернему потоку. Предположим, что функция **IncCount** дочернего потока успела 50 000 раз добавить в счетчик единицу, так что переменная **g_counter** стала равна 150 000. В этот момент квант дочернего потока истек и система возвращает процессор первичному потоку. При этом система восстанавливает контекст первичного потока с **незавершенной операцией**. Завершая эту операцию, процессор переписывает значение 99 999 из своего регистра в глобальную переменную **g_counter**. В результате квант работы дочернего потока пошел насмарку – его результаты утеряны!

Таким образом, одновременное использование несколькими потоками общей глобальной переменной без должной синхронизации может быть источником ошибок.

Другой пример работы несинхронизированных потоков. Каждый из потоков main и thread выводит строки **одинаковых** чисел. Но из-за параллельной работы потоков каждая выведенная строка может содержать **неодинаковые** между собой элементы, поскольку при выводе очередной строки процесс может быть прерван и процессор передан другому процессу.

```
#include <windows.h>
#include <iostream.h>
DWORD WINAPI thread(LPVOID)
{  int i, j;
   for (j = 0; j < 10; ++j)
   {      // вывод строки чисел j
    for (i = 0; i < 10; ++i)
    {    cout << j << ' ' << flush;      Sleep(17);    }
    cout << endl;
   }
return 0;
}
int main()
{   int i, j;
    HANDLE  hThread;
    DWORD  IDThread;
    hThread=CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
    if (hThread == NULL)    return GetLastError();
    for (j = 10; j < 20; ++j)
     {   // вывод строки чисел j
                    for (i = 0; i < 10; ++i)
                    {     cout << j << ' ' << flush;      Sleep(17);    }
                    cout << endl;
    }
          WaitForSingleObject (hThread, INFINITE);       //  ожидание завершения
                                                         //  работы потока thread
    return 0;
}
```

Кроме перечисленных возможны и другие проблемы доступа к общим системным ресурсам. Потоки должны взаимодействовать друг с другом в двух основных случаях:

- совместно используя разделяемый ресурс (чтобы не разрушить его);

- когда нужно уведомить другие потоки о завершении каких-либо операций.

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

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

- мьютексы (mutexs);
- события (events);
- семафоры (semaphores).

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

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

- потоки;
- процессы.

Для блокирования потоков используются функции ожидания, параметрами которых являются перечисленные объекты синхронизации. Две основные функции ожидания: **WaitForSingleObject** и **WaitForMultipleObjects** (функцию **WaitForSingleObject** уже использовали в примерах ожидания завершения выполнения потока).

Сама блокировка потока выполняется следующим образом. Если дескриптор объекта синхронизации является параметром функции ожидания, а сам объект синхронизации находится в **несигнальном состоянии**, то поток, вызвавший эту функцию ожидания, блокируется до перехода этого объекта синхронизации в **сигнальное состояние**.

По завершении потока выполняется переход в сигнальное состояние и происходит выход из состояния ожидания.

Функция **WaitForSingleObject**, которая имеет следующий прототип:

DWORD WaitForSingleObject (HANDLE hHandle, // дескриптор объекта

DWORD dwMiliiseconds) ; // интервал ожидания в миллисекундах

используется для ожидания перехода в сигнальное состояние **одного** объекта синхронизации.

Первый аргумент функции **WaitForSingleObject** – дескриптор объекта синхронизации (в ранее использованных примерах – дескриптор потока).

Второй аргумент функции **WaitForSingleObject**:

- интервал времени, в течение которого функция ожидает перехода объекта синхронизации в сигнальное состояние;
- нулевое значение для проверки состояния объекта синхронизации;
- значение **INFINITE**, при котором функция ждет перехода объекта синхронизации в сигнальное состояние бесконечно долго.

Основные случаи успешного завершения функции **WaitForSingleObject**:

- WAIT_OBJECT_O – объект перешел в сигнальное состояние;
- WAIT_TIMEOUT – время ожидания истекло.

Ошибочное завершение функции **WaitForSingleObjects** – WAIT_FAILED.

Функция **WaitForMultipleObject** используется для ожидания перехода в сигнальное состояние нескольких объектов синхронизации или одного из нескольких объектов синхронизации (в зависимости от значений параметров).

Для решения проблемы взаимного исключения для параллельных потоков, выполняемых в **рамках одного процесса**, используется примитив синхронизации **критические секции** (**critical sections**), который не является объектами ядра.

Начнем с этого примитива синхронизации.

3.1. КРИТИЧЕСКИЕ СЕКЦИИ

В ОС Windows проблема взаимного исключения для параллельных потоков, выполняемых ***в контексте одного процесса***, наиболее просто решается при помощи объекта типа **CRITICAL_SECTION**. Для работы с объектами типа **CRITICAL_SECTION** используются следующие функции:

```
VOID InitializeCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
// инициализация критической секции
VOID EnterCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
// вход в критическую секцию
BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
// попытка войти в критическую секцию
```

BOOL LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);

// выход из критической секции

BOOL DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);

// разрушение критической секции

Каждая из этих функций имеет единственный параметр, указатель на объект типа **CRITICAL_SECTION**. Все эти функции, за исключением **TryEnterCriticalSection**, не возвращают значения. Функция **TryEnterCriticalSection** возвращает нулевое значение, если поток вошел в критическую секцию или уже находится в ней, в противном случае функция возвращает значение **FALSE**.

Порядок работы с этими функциями

Перед тем как начать работу с объектом типа **CRITICAL_ SECTION**, его необходимо инициализировать **(InitializeCriticalSection)**. Перед входом в критическую секцию вызывается функция **EnterCriticalSection**, которая исключает одновременный вход в критические секции, выполняющиеся в параллельных потоках и связанные с разделяемым ресурсом. После завершения работы с разделяемым ресурсом поток должен покинуть свою критическую секцию, что выполняется вызовом функции **LeaveCriticalSectio**n. После окончания работы с объектом типа **CRITICAL_SECTION** необходимо освободить все ресурсы, которые использовались этим объектом, вызвав функцию **DeleteCriticalSection**.

Теперь рассмотрим использование этих функций в задаче взаимного исключения в предыдущем примере. Задача будет заключаться в следующем: нужно так синхронизировать потоки main и thread, чтобы в каждой строке выводились только равные между собой числа (показаны вставки).

```
#include <windows.h>
#include <iostream.h>
CRITICAL_SETION cs;
DWORD WINAPI thread(LPVOID)
{  int i,j;
   for (j = 0; j < 10; ++j)
   {    // входим в критическую секцию
     EnterCriticalSection (&cs);
     for (i = 0; i < 10; ++i)
```

```
            {     cout << j << ' '<< flush;      Sleep(7);    }
     cout << endl;
     LeaveCriticalSection(&cs);    // выходим из критической секции
    }
return 0;
}
int main()
{  int i,j;
   HANDLE  hThread;
   DWORD  IDThread;
    InitializeCriticalSection(&cs);   // инициализация критической секции
    hThread=CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
    if (hThread == NULL)    return GetLastError();
    for (j = 10; j < 20; ++j)
    {     // входим в критическую секцию
     EnterCriticalSection(&cs);
         for (i = 0; i < 10; ++i)
         {     cout << j << ' ' << flush;      Sleep(7);    }
           cout << endl;
          LeaveCriticalSection(&cs);     // выходим из критической секции
    }
    WaitForSingleObject (hThread, INFINITE);   // ожидание завершения работы
                                                // потока thread
    DeleteCriticalSection(&cs);      // закрываем критическую секцию
    return 0;
}
```

Другой способ синхронизации – использование функции **TryEnterCriticalSection** вместо функции **EnterCriticalSection**. При использовании этих функций в приведенной выше программе необходимо заменить фрагмент из двух строк в основном и порожденном потоке на фрагмент из трех строк:

```
    {     // входим в критическую секцию
     EnterCriticalSection (&cs);
```

```
     // попытка войти в критическую секцию
     if (TryEnterCriticalSection (&cs))
```

Замечание. Поскольку объекты типа **CRITICAL_SECTION** не являются объектами ядра ОС, работа с ними происходит несколько быстрее, чем с объектами синхронизации, которые являются объектами ядра ОС (те, что будем рассматривать далее): для них не требуется дополнительной работы на переключение контекстов потоков из режима пользователя в режим ядра ОС. Поэтому при разработке многопоточных приложений для решения задач взаимного исключения, как правило, используют объекты типа **CRITICAL_SECTION**.

3.2. МЬЮТЕКСЫ

Для решения проблемы взаимного исключения между параллельными потоками, выполняющимися **в контекстах разных процессов**, в ОС Windows используется объект ядра мьютекс. Происхождение – от английского слова mutex (сокращение от mutual exclusion – «взаимное исключение»). **Мьютекс находится в сигнальном состоянии, если он не принадлежит ни одному потоку**. В противном случае мьютекс находится в несигнальном состоянии. Одновременно мьютекс может принадлежать только одному потоку.

Потоки, ждущие сигнального состояния мьютекса, обслуживаются в порядке FIFO.

Используемые функции при работе с мьютексами

1. Создание мьютекса – функция **CreateMutex**, которая имеет следующий прототип:

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

При значении параметра **lpMutexAttributes**, равного NULL, атрибуты защиты заданы по умолчанию (дескриптор мьютекса не является наследуемым, и доступ к мьютексу открыт для всех пользователей).

Если значение параметра **bInitialOwner** равно TRUE, то мьютекс сразу переходит во владение потоку, которым он был создан. В противном случае созданный мьютекс свободен и может быть захвачен любым потоком.

Значение параметра **lpName** – уникальное имя мьютекса для всех процессов, выполняющихся под управлением ОС. Это имя позволяет

обращаться к мьютексу из других процессов, запущенных под управлением этой же ОС. Значением параметра **lpName** может быть пустой указатель NULL (так называемый безымянный мьютекс).

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

2. Доступ к уже созданному мьютексу из другого потока – функция **OpenMutex**, которая имеет следующий прототип:

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

Параметр **dwDesiredAccess**:

- MUTEX_ALL_ACCESS – полный доступ к мьютексу;
- SYNCHRONIZE – поток может использовать мьютекс только в функциях ожидания, чтобы захватить мьютекс, или в функции **ReleaseMutex** для его освобождения.

Параметр **blnheritHandle** определяет свойство наследования мьютекса:

- TRUE – дескриптор открываемого мьютекса является наследуемым;
- NULL – дескриптор не наследуется.

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

3. Захват мьютекса – любая функция ожидания.

4. Освобождение мьютекса – функция **ReleaseMutex**, имеющая прототип:

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

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

Пример несинхронизированных потоков, выполняющихся в разных процессах:

```
#include <windows.h>
#include <iostream.h>
int main()
{  char  lpszAppName[] = "C:\\ConsoleProcess.exe";
   STARTUPINFO  si;
   PROCESS_INFORMATION  pi;
   ZeroMemory(&si, sizeof(STARTUPINFO));
   si.cb = sizeof(STARTUPINFO);
   if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
       NULL, NULL, NULL, &si, &pi))// создание нового консольного процесса
            {   < Обработка ошибки>   return GetLastError();  }
    for (int j = 0; j < 10; ++j)              // вывод на экран строки
    {   for (int i = 0; i < 10; ++i)
     {     cout << j << ' ' << flush;     Sleep(10);   }
                          cout << endl;
     }
    WaitForSingleObject(pi.hProcess, INFINITE);       //   ожидание завершения
                                                      //   дочернего процесса
    CloseHandle(pi.hThread);        // закрываем дескрипторы дочернего
    CloseHandle(pi.hProcess);       // процесса в текущем процессе
    return 0;
}
//////////////////////////////////////
#include <windows.h>
#include <iostream.h>
int main()        // дочерний процесс
{  int  i,j;
  for (j = 10; j < 20; ++j)
  {       for (i = 0; i < 10; ++i)
          {     cout << j << ' ' << flush;     Sleep(5);   }
          cout << endl;
  }
   return 0;
}
```

Пояснения к программе

Это тот же самый пример, когда основной и дочерний потоки выводят строки одинаковых чисел с той разницей, что дочерний поток принадлежит другому процессу, созданному исходным процессом. Точно так же из-за параллельной работы потоков каждая выведенная строка может содержать **неодинаковые** между собой элементы, поскольку при выводе очередной строки процесс может быть прерван и процессор передан другому процессу.

Пример синхронизированных потоков, выполняющихся в разных процессах с использованием мьютекса:

```
#include <windows.h>
#include <iostream.h>
int main()
{ char lpszAppName[] = "C:\\ConsoleProcess.exe";
  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  HANDLE hMutex;
  hMutex = CreateMutex(NULL, FALSE, "DemoMutex");      // создание мьютекса
  if (hMutex == NULL)
          {   < Обработка ошибки>   return GetLastError();  }
  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb = sizeof(STARTUPINFO);
  if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
     NULL, NULL, NULL, &si, &pi))     // создаем новый консольный процесс
          {   < Обработка ошибки>   return GetLastError();  }
  for (int j = 0; j < 10; ++j)              // вывод на экран строки
  {   WaitForSingleObject(hMutex, INFINITE);  // захват мьютекса
      for (int i = 0; i < 10; ++i)
              {    cout << j << ' ' << flush;     Sleep(10);   }
          cout << endl;
      ReleaseMutex(hMutex);     // освобождение мьютекса
  }
  CloseHandle(hMutex);   // закрытие дескриптора мьютекса
  WaitForSingleObject(pi.hProcess, INFINITE);  //   ожидание завершения
                                  //   дочернего процесса
```

```
    CloseHandle(pi.hThread);   // закрываем дескрипторы дочернего
    CloseHandle(pi.hProcess);  // процесса в текущем процессе
    return 0;
}
////////////////////////////////////
#include <windows.h>
#include <iostream.h>
int main()        // дочерний процесс
{   HANDLE  hMutex;
  int  i,j;
  hMutex = OpenMutex(SYNCHRONIZE, FALSE, "DemoMutex"); // открытие мьютекса
  if (hMutex == NULL)
            {   < Обработка ошибки>    return GetLastError();  }
  for (j = 10; j < 20; ++j)
  {   WaitForSingleObject(hMutex, INFINITE);     // захват мьютекса
      for (i = 0; i < 10; ++i)
         {     cout << j << ' ' << flush;      Sleep(5);    }
      cout << endl;
      ReleaseMutex(hMutex);                    // освобождение мьютекса
  }
   CloseHandle(hMutex);              // закрытие дескриптора мьютекса
   return 0;
}
```

Комментарии

1. При создании и открытии мьютекса (**CreateMutex, OpenMutex**) второй параметр – FALSE, что говорит о том, что мьютекс не принадлежит никакому потоку (состояние Signal).

2. При захвате мьютекса (**WaitForSingleObject(hMutex, INFINITE);**) первый успевший поток (пока мьютекс в сигнальном состоянии) не блокируется и начинает вывод, при этом мьютекс переводится в несигнальное состояние и до тех пор, пока поток не освободит мьютекс (вернет его в сигнальное состояние), любой другой поток будет переводиться функцией **WaitForSingleObject** в состояние ожидания.

3. Когда один поток выдаст **ReleaseMutex**, другой поток, заблокированный функцией **WaitForSingleObject**, выйдет из состояния ожидания.

3.3. СОБЫТИЯ

Задача оповещения одного потока о некотором действии, которое совершил другой поток, в Windows носит название **задачи условной синхронизации** или **задачи оповещения**. В Windows оповещение одного потока о том, что другой поток произвел некоторое действие, выполняется посредством объекта ядра, называемого **событием** (Events). Различают два типа событий:

- события с ручным сбросом – в **несигнальное состояние** событие с ручным сбросом можно перевести только посредством вызова функции **ResetEvent**;
- события с автоматическим сбросом – в **несигнальное состояние** событие с автоматическим сбросом переходит как при помощи функции **ResetEvent**, так и при помощи **функции ожидания**.

Если события с автоматическим сбросом ждут несколько потоков, используя функцию **WaitForSingieobject**, то из состояния ожидания освобождается только один из этих потоков.

Используемые функции при работе с событиями

1. Создание события – функция **CreateEvent**, которая имеет следующий прототип:

```
HANDLE CreateEvent ( LPSECDRITY_ATTRIBUTES
IpSecurityAttributes,
                        // атрибуты защиты
BOOL bManualReset,      // тип события
BOOL binitialstate,         // начальное состояние события
LPCTSTR lpName          // имя события
```

При значении первого параметра, равного NULL, атрибуты защиты заданы по умолчанию. Основную смысловую нагрузку в этой функции несут второй и третий параметры.

При значении второго параметра **bManualReset**, равного TRUE, создается событие с ручным сбросом, в противном случае – с автоматическим сбросом.

Если значение третьего параметра **binitialstate** равно TRUE, то начальное состояние события является сигнальным, в противном слу-

чае – несигнальным. Параметр **lpName** задает имя события, которое позволяет общаться к нему из потоков, выполняющихся в разных процессах. Этот параметр может быть равен NULL; создается безымянное событие.

В случае удачного завершения функция **CreateEvent** возвращает дескриптор созданного события. В случае неудачи эта функция возвращает значение NULL. Если событие с заданным именем уже существует, то функция **CreateEvent** возвращает дескриптор этого события, а функция **GetLastError**, вызванная после функции **CreateEvent**, возвращает значение ERROR_ALREADY_EXISTS (как и с мьютексами).

2. Перевод события в сигнальное состояние – функция **SetEvent**, имеющая прототип:

BOOL SetEvent (HANDLE hEvent); // дескриптор события

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

3. Перевод события в несигнальное состояние – функция **ResetEvent** с прототипом:

BOOL ResetEvent (HANDLE hEvent); // дескриптор события

Аналогично при успешном завершении эта функция возвращает ненулевое значение, в случае неудачи – FALSE.

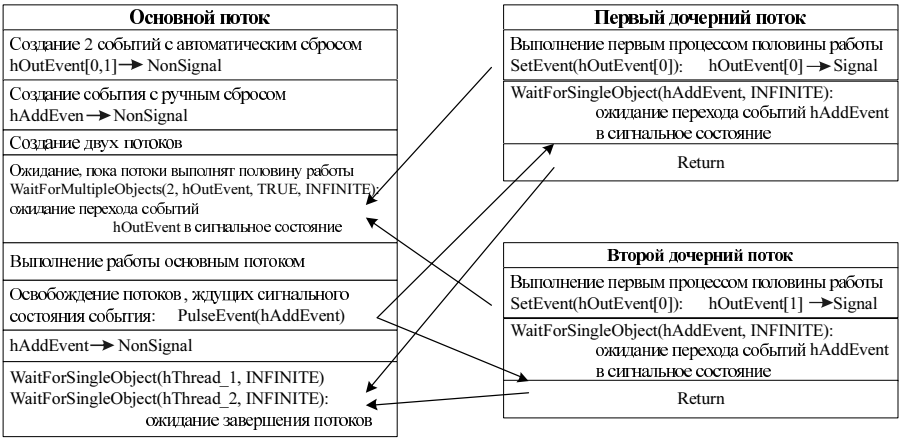
4. Освобождение потоков, ждущих сигнального состояния события с ручным сбросом, – функция **PulseEvent**, которая имеет следующий прототип:

BOOL PulseEvent (HANDLE hEvent); // дескриптор события

При вызове этой функции все потоки, ждущие события с дескриптором **hEvent**, выводятся из состояния ожидания, а само событие сразу переходит в несигнальное состояние.

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

Пример. Схема синхронизации потоков при помощи событий с ручным сбросом в рамках одного процесса.



Текст программы приведен в приложении 3.

Пояснения к программе

1. В основном потоке создаются два события с автоматическим сбросом **hOutEvent** и событие с ручным сбросом **hAddEven**. После создания все события переходят в несигнальное состояние.

2. Создаются два дочерних потока.

3. Основной поток переходит в состояние ожидания (функция **WaitForMultipleObjects**), дожидаясь, пока дочерние потоки выполнят половину работы и функцией **SetEvent** переведут события **hOutEvent** в сигнальное состояние.

4. После перевода событий **hOutEvent** в сигнальное состояние основной поток выполняет свою часть работы. Дочерние процессы в это время находятся в состоянии ожидания (функция **WaitForSingleObject**), ожидая перевода основным потоком события **hAddEven** в сигнальное состояние. В результате размораживаются дочерние процессы, а событие **hAddEven** опять переходит в несигнальное состояние.

5. Дочерние процессы доделывают свою работу и завершаются. Основной процесс дожидается завершения дочерних потоков.

Доступ к существующему событию из потока другого родственного процесса обеспечивается функцией **OpenEvent** с прототипом:

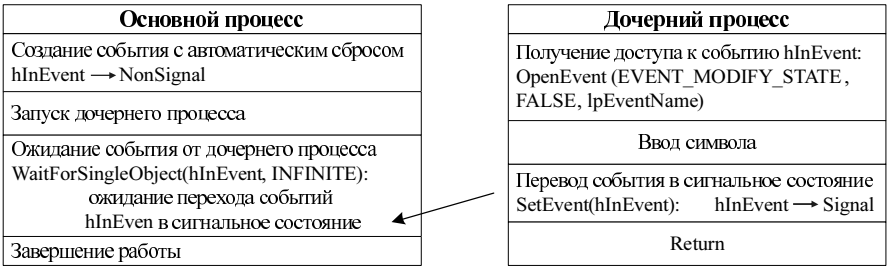
```
HANDLE OpenEvent( DWORD dwDesiredAccess,    // флаги доступа
                  BOOL blnheritHandle,      // режим наследования
                  LPCTSTR lpName);          // имя события
```

Флаги доступа (параметр dwDesiredAccess):

- EVENT_ALL_ACCESS – полный доступ: поток может выполнять над событием любые действия;
- EVENT_MODIFY_STATE – модификация состояния: поток может использовать функции SetEvent и ResetEvent для изменения состояния события;
- SYNCHRONIZE – поток может использовать событие в функциях ожидания.

Пример. Схема синхронизации потоков при помощи событий с автоматическим сбросом, выполняющихся в разных процессах.

Текст программы приведен в приложении 3.



Пояснения к программе

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

Выводы: события в ОС Windows – некоторый усеченный аналог сигналов в ОС Unix.

3.4. СЕМАФОРЫ

Семафоры в ОС Windows описываются объектами ядра **Semaphores**. **Семафор находится в сигнальном состоянии, если его значение больше нуля**. В противном случае семафор находится в несигнальном состоянии. Потоки, ждущие сигнального состояния семафора,

обслуживаются в порядке FIFO (первый пришел, первый вышел). Однако если поток ждет наступления асинхронного события, то функции ядра могут исключить поток из очереди к семафору для обслуживания наступления этого события. После этого поток становится в конец очереди семафора.

Используемые функции при работе с семафорами

1. Создание семафора – функция **CreateSemaphore** с прототипом:

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

При значении первого параметра, равного NULL, атрибуты защиты заданы по умолчанию. Основную смысловую нагрузку в этой функции несут второй и третий параметры.

Значение второго параметра **UnitialCount** устанавливает начальное значение семафора, которое должно быть не меньше 0 и не больше его максимального значения, которое устанавливается третьим параметром **IMaximumCount**.

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

В случае успешного завершения функция **CreateSemaphore** возвращает дескриптор семафора, в случае неудачи – значение NULL. Если семафор с заданным именем уже существует, то функция **CreateSemaphor** возвращает дескриптор этого семафора, а функция **GetLastError**, вызванная после функции **CreateSemaphor**, вернет значение ERROR_ALREADY_EXISTS (как и с другими объектами ядра).

2. Уменьшение значения семафора: значение семафора уменьшается на 1 при его использовании в функции ожидания.

3. Увеличение значения семафора – функция **ReleaseSemaphore**, которая имеет следующий прототип:

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

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

Значение третьего параметра **lpPreviousCount** этой функции может быть равно NULL.

4. Доступ к существующему семафору из потока другого процесса (если известно, что семафор с заданным именем уже существует) – функция **OpenSemaphore**, которая имеет следующий прототип:

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

Флаги доступа аналогично соответствующей функции для событий (параметр dwDesiredAccess):

- SEMAPHORE_ALL_ACCESS – полный доступ к семафору: поток может выполнять над семафором любые действия;
- SEMAPHORE_MODIFY_STATE – модификация состояния: поток может выполнять только функцию **ReleaseSemaphore** для изменения значения семафора;
- SYNCHRONIZE – поток может использовать семафор только в функциях ожидания.

Пример программы с несинхронизированными потоками:

```
#include <windows.h>
#include <iostream.h>
volatile int a[10];
DWORD WINAPI thread(LPVOID)
{  for (int i = 0; i < 10; i++)          //  подготовка элементов массива
    {    a[i] = i + 1;
         Sleep(7);
     }
   return 0;
}
int main()
```

```
{   int i;
    HANDLE  hThread;
    DWORD  IDThread;
    cout << "An initial state of the array: ";
    for (i = 0; i < 10; i++)
                cout << a[i] <<' ';
    cout << endl;
//  создание потока, подготавливающего элементы массива
    hThread = CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
    if (hThread == NULL)
                    return GetLastError();
    cout << "A modified state of the array: ";
    for (i = 0; i < 10; i++)        // вывод элементов массива
    {    cout << a[i] << ' ' << flush;
                Sleep(11);
    }
    cout << endl;
    CloseHandle(hThread);
    return 0;
}
```

Пояснения к программе

1. В основном потоке **main** осуществляет вывод элементов массива **a**.

2. Поток **thread** последовательно присваивает элементам массива **a** значения, которые на единицу больше, чем их индекс.

3. Поток **main** последовательно выводит элементы массива **a** на консоль.

Так как потоки **thread** и **main** не синхронизированы, то измененное состояние массива, которое выведет на консоль поток **main**, не гарантировано.

Задача состоит в обеспечении того, чтобы поток **main** выводил на консоль элементы массива **a** сразу после их подготовки потоком **thread**.

Пример синхронизации потоков с использованием считающего семафора:

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

```
HANDLE hSemaphore; //  Дескриптор семафора
volatile int a[10];
DWORD WINAPI thread(LPVOID)
{  for (int i = 0; i < 10; i++)            //   подготовка элементов массива
     {     a[i] = i + 1;
      ReleaseSemaphore(hSemaphore,1,NULL);   //   отметить, что очередной
                                                 //   элемент подготовлен
      Sleep(100);
      }
    return 0;
}
int main()
{   int i;
     HANDLE  hThread;
     DWORD  IDThread;
     cout << "An initial state of the array: ";
     for (i = 0; i < 10; i++)
                    cout << a[i] <<' ';
     cout << endl;
     hSemaphore=CreateSemaphore(NULL, 0, 10, NULL);   // создание семафора
     if (hSemaphore == NULL)        return GetLastError();
//   создание потока, подготавливающего элементы массива
     hThread = CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
     if (hThread == NULL)
             return GetLastError();
     cout << "A modified state of the array: ";
     for (i = 0; i < 10; i++)        // вывод элементов массива
     {    WaitForSingleObject(hSemaphore, INFINITE);// поток main выводит элементы
                                       // массива только после их подготовки потоком thread
             cout << a[i] << ' ' << flush;
     }
     cout << endl;
     CloseHandle(hSemaphore);
     CloseHandle(hThread);
     return 0;
}
```

Пояснения к программе

1. В основном потоке создан семафор. Начальное значение семафора – 0.

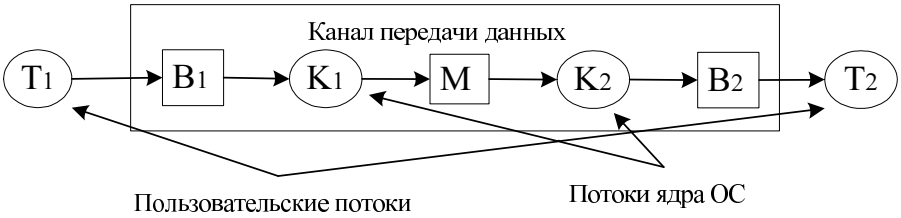
2. При формировании очередного элемента массива **a** функцией **ReleaseSemaphore** увеличивается на 1 значение семафора.

3. В основном потоке очередной элемент массива **a** не выводится на консоль до тех пор, пока семафор не перейдет в сигнальное состояние (приостановку процесса **main** обеспечивает функция **WaitForSingleObject**). Когда семафор переходит в сигнальное состояние (функцией **ReleaseSemaphore** в дочернем процессе), функция **WaitForSingleObject** уменьшает значение семафора и на консоль выводится элемент массива **a**.

4. ПЕРЕДАЧА ДАННЫХ МЕЖДУ ПРОЦЕССАМИ

Под **обменом данными между параллельными процессами** понимается пересылка данных от одного потока к другому потоку, предполагая, что эти потоки исполняются в контекстах разных процессов.

Если потоки выполняются в одном процессе, то для обмена данными между ними можно использовать глобальные переменные и средства синхронизации потоков. Дело обстоит сложнее в том случае, если потоки выполняются в разных процессах – потоки не могут обращаться к общим переменным и для обмена данными между ними необходимы специальные средства ОС. Если говорить концептуально, то для обмена данными между процессами создается **канал передачи данных**, организация которого схематически имеет вид:



Канал данных включает входной и выходной буферы памяти, потоки ядра операционной системы и общую память, доступ к которой

имеют оба потока ядра. Работает канал передачи данных следующим образом:

- первый поток ядра операционной системы читает данные из входного буфера B_1 и записывает их в общую память М;
- второй поток ядра читает данные из общей памяти М и записывает их в буфер B_2.

Пользовательские потоки T_1 и T_2 посредством вызова функций ядра операциями системы имеют доступ к буферам B_1 и B_2 соответственно. Поэтому пересылка данных из потока T_1 в поток T_2 происходит следующим образом:

- пользовательский поток T_1 записывает данные в буфер B_1, используя некоторую специальную функцию ядра ОС;
- поток K_1 ядра операционной системы читает данные из буфера B_1 и записывает их в общую память М;
- поток K_2 ядра ОС читает данные из общей памяти М и записывает их в буфер B_2;
- пользовательский поток T_2 читает данные из буфера B_2.

Прежде чем передавать данные между процессами, нужно установить между этими процессами связь. С точки зрения направления передачи данных различают следующие виды связей:

- *полудуплексная связь*, когда данные могут передаваться только в одном направлении;
- *дуплексная связь*, когда данные могут передаваться в обоих направлениях.

При обмене данными между параллельными процессами различают два способа передачи данных:

- потоком;
- сообщением.

Если данные передаются непрерывной последовательностью байт, то такая пересылка данных называется **передача данных потоком**. В этом случае общая память М, доступная потокам ядра ОС, может и отсутствовать, а пересылка данных выполняется одним потоком ядра непосредственно из буфера B_1 в буфер B_2.

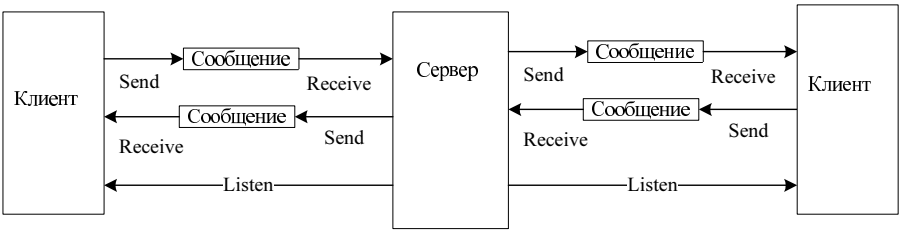
Если данные пересылаются группами байт, то такая группа байт называется **сообщением**, а сама пересылка данных называется **передачей данных сообщений**.

Сообщение состоит из двух частей: заголовка и тела сообщения. В заголовке сообщения находится такая служебная информация, как:

- тип сообщения;
- имя адресата сообщения;
- имя отправителя сообщения;
- длина сообщения.

Тело сообщения содержит само сообщение.

Обмен сообщениями между процессами выполняется по технологии «Клиент-сервер»: сервер «слушает» канал связи и принимает сообщения от всех клиентов.



При передаче данных различают синхронный и асинхронный обмен данными.

Если поток-отправитель, отправив сообщение, блокируется до получения этого сообщения потоком-адресатом, то такое отправление сообщения называется **синхронным**. В противном случае отправление сообщения называется **асинхронным**. Если поток-адресат блокируется до тех пор, пока не получит сообщение, то такое получение сообщения называется **синхронным**. В противном случае получение сообщения называется **асинхронным**.

Обмен сообщениями может быть и **синхронным**, и **асинхронным**.

Обмен данными потоком всегда происходит **синхронным образом**, так как в этом случае между отправителем и адресатом устанавливается непосредственная связь.

4.1. РАБОТА С АНОНИМНЫМИ КАНАЛАМИ В ОС WINDOWS

Терминология

Анонимным каналом называется объект ядра ОС, который обеспечивает передачу данных в виде непрерывной последовательности байт между процессами, выполняющимися на одном компьютере.

Процесс, который создает анонимный канал, называется **сервером анонимного канала**. Процессы, которые связываются с анонимным

каналом, называются **клиентами анонимного канала**. Поскольку анонимный канал не имеет имени, доступ к такому каналу имеют только родительский процесс-сервер и дочерние процессы-клиенты этого канала.

Характеристики анонимных каналов:

- не имеют имени;
- полудуплексные;
- передача данных потоком;
- синхронный обмен данными;
- возможность моделирования любой топологии связей (1-1, 1-N, N-1, N-M).

Порядок работы с анонимными каналами:

- создание анонимного канала сервером;
- соединение клиентов с каналом;
- обмен данными по каналу;
- закрытие канала.

Используемые функции при работе с анонимными каналами

1. Создание анонимного канала процессом-сервером – функция **CreatePipe** с прототипом:

```
BOOL CreatePipe (PHANDLE hReadHandle,
                                    // дескриптор для чтения из канала
PHANDLE hWriteHandle,               // дескриптор для записи в канал
LPSECURITY_ATTRIBUTES lpPipeAttributes, // атрибуты защиты
 DWORD dwSize);                     // размер буфера в байтах
```

При удачном завершении функция **CreatePipe** возвращает ненулевое значение, а в случае неудачи – FALSE.

В случае успешного завершения функция **CreatePipe** создает два дескриптора анонимного канала: один для чтения данных из канала, а второй для записи данных в канал (**hReadHandle, hWriteHandle**). Первый дескриптор используется в функциях чтения данных из канала, второй – в функциях записи данных в канал.

Если значение параметра **dwsize** установлено равным 0, ОС выбирает размер буфера по умолчанию.

2. Соединение клиентов с каналом.

Для соединения процесса-клиента с анонимным каналом ему необходимо передать один из дескрипторов анонимного канала. При этом

передаваемый дескриптор должен быть наследуемым, а сам процесс-клиент должен быть дочерним процессом процесса сервера анонимного канала и наследовать наследуемые дескрипторы процесса-сервера.

Наследование дескрипторов анонимного канала определяется значением поля **bInheritHandle** в структуре типа SECURITY_ATTRIBUTES, на которую указывает третий параметр **lpPipeAttributes** функции **CreatePipe**. Если значение этого поля равно TRUE, то дескрипторы анонимного канала создаются наследуемыми, иначе дескриптор ненаследуемый.

Возможны две альтернативы:

а) определить в функции **CreatePipe** дескрипторы анонимного канала, передаваемого процессу-клиенту, как наследуемые;

б) определить в функции **CreatePipe** дескрипторы анонимного как ненаследуемые, а затем с помощью функции **DuplicateHandle** создать дубликат исходного дескриптора, определив новый дескриптор как наследуемый. Прототип функции **DuplicateHandle**:

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

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

Первые четыре параметра очевидны.

Основной шестой параметр **bInheritHandle** функции **DuplicateHandle** устанавливает свойство наследования нового дескриптора. Если значение этого параметра равно TRUE, то создаваемый дубликат исходного дескриптора является наследуемым, в случае FALSE – ненаследуемым.

Для того чтобы процесс-клиент наследовал дескрипторы анонимного канала, он должен быть создан функцией **CreateProcess** в процессе-сервере анонимного канала и параметр **bInheritHandle** этой функции должен быть установлен в TRUE.

Параметр **dwOptions** – комбинация флагов DUBLICATE_ CLOSE_SOURCE и DUBLICATE_SAME_SOURCE.

Если установлен флаг DUBLICATE_CLOSE_SOURCE, то при любом своем завершении функция **DuplicateHandle** закрывает исходный дескриптор. Если установлен флаг DUBLICATE_SAME_SOURCE, то режимы доступа к объекту через дублированный дескриптор совпадают с режимами доступа к объекту через исходный дескриптор.

Пятый параметр **dwDesiredAccess** определяет возможные режимы доступа к объекту через дубликат исходного дескриптора, используя определенную комбинацию флагов. Если доступ к объекту не изменяется, что определяется значением последнего параметра **dwOptions**, то система игнорирует значение данного параметра.

Наиболее распространенный способ передачи наследуемого дескриптора процессу-клиенту – через командную строку.

3. Обмен данными по анонимному каналу.

Для обмена данными по анонимному каналу в ОС Windows используются те же функции, что для записи/чтения данных в файл: **WriteFile** и **ReadFile**.

Прототип функция WriteFiie:

```
BOOL WriteFile (HANDLE hAnonymousPipe,
                                    // дескриптор анонимного канала
LPCVOID IpBuffer,                   // буфер данных
DWORD dwNumberOfBytesToWrite,       // количество байт для записи
LPDWORD lpNumberOfBytesWritten,     // количество записанных байт
LPOVERLAPPED lpOverlapped);         // асинхронный вывод
```

Функция **WriteFile** записывает в анонимный канал (в файл) количество байт, заданных третьим параметром **dwNumberOfBytesToWrite**, из буфера данных, на который указывает второй параметр **ipBuffer**. Дескриптор вывода этого анонимного канала должен быть задан первым параметром функции **WriteFile**. При успешном завершении функция **WriteFile** возвращает ненулевое значение, а в случае неудачи – FALSE. Количество байт, записанных функцией **WriteFile** в анонимный канал, возвращается в переменной, на которую указывает четвертый параметр **lpNumberOfBytesWritten**. Параметр **lpOverlapped** предназначен для выполнения асинхронной операции вывода (для каналов не используется – NULL).

Функция **ReadFile** имеет схожий прототип:

```
BOOL ReadFile ( HANDLE hAnonymousPipe,
                                  // дескриптор анонимного канала
LPCVOID lpBuffer,                 // буфер данных
DWORD dwNumberOfBytesToRead,      // количество байт для чтения
LPDWORD lpNumberOfBytesRead,      // количество прочитанных байт
LPOVERLAPPED lpOverlapped);       // асинхронный ввод
```

Функция **ReadFile** читает из анонимного канала (из файла) количество байт, заданных третьим параметром **dwNumberOfBytesToRead**, в буфер данных, на который указывает второй параметр **ipBuffe**r. Дескриптор ввода этого анонимного канала должен быть задан первым параметром функции **ReadFile**. При успешном завершении функция **ReadFile** возвращает ненулевое значение, а в случае неудачи – FALSE. Количество байт, прочитанных функцией **ReadFile** из анонимного канала, возвращается в переменной, на которую указывает четвертый параметр **lpNumberOfBytesRead**.

Пример. Схема взаимодействия процессов, в котором процесс-сервер создает анонимный канал и дочерний процесс, которому передает один из дескрипторов этого анонимного канала. Для определенности клиенту передается дескриптор для записи в анонимный канал, а серверу остается дескриптор для чтения. Дочерний и основной процессы обмениваются данными через анонимный канал.

Схема взаимодействия приведена ниже, текст программы – в приложении 4.

Пояснения.

1. Программа представлена в двух вариантах: канал с наследуемыми дескрипторами (I вариант) и канал с ненаследуемыми дескрипторами (II вариант).

2. В случае канала с ненаследуемыми дескрипторами дополнительно функцией DuplicateHandle наследуется дескриптор для записи в анонимный канал.

3. Создается дочерний процесс, который становится клиентом анонимного канала.

4. Для передачи дескриптора используется командная строка, в которую в качестве параметра запускаемой функции (C:\\Client.exe) записывается дескриптор для записи в анонимный канал (I вариант), либо дубликат дескриптора для записи в анонимный канал (II вариант).

5. Полученный как параметр дескриптор (или его дубликат) для записи в канал преобразуется в клиентском процессе в число, и с его

использованием осуществляется запись в анонимный канал. В серверном процессе осуществляется чтение из канала.

6. Аккуратным образом закрываются все дескрипторы (канала, процесса, потока).

Может быть организован и двусторонний обмен данными по анонимному каналу между клиентом и сервером. Для этого и дескрипторы чтения, и дескрипторы записи анонимного канала используются как сервером, так и клиентом этого анонимного канала.

Однако для организации такого двустороннего обмена данными по анонимному каналу сервер и клиенты канала должны синхронизировать доступ к этому каналу. Для организации передачи данных необходимо разработать протокол передачи данных или использовать объ-

екты синхронизации, которые исключают одновременный неконтролируемый доступ параллельных потоков к анонимному каналу. При отсутствии такой синхронизации возможно одновременное чтение данных сервером и клиентом, так как они работают параллельно и это вызовет неправильную работу программы и ее зависание (см. фрагмент лекции по Unix).

4.2. ПЕРЕНАПРАВЛЕНИЕ ВВОДА-ВЫВОДА

Анонимные каналы можно использовать для перенаправления стандартного ввода-вывода.

Компилятор языка программирования C++ фирмы Microsoft содержит стандартную библиотеку, которая поддерживает три варианта функций стандартного ввода-вывода. Эти функции описываются в заголовочных файлах **stdio.h**, **iostream.h** и **conio.h**.

Функции, которые описаны в файле **stdio.h**, обеспечивают ввод-вывод в следующие стандартные файлы:

- **stdin** – стандартный файл ввода;
- **stdout** – стандартный файл вывода;
- **stderr** – файл вывода сообщений об ошибках.

Эти функции составляют стандартную библиотеку ввода-вывода языка программирования C.

Функции и операторы, которые описаны в заголовке **iostream.h**, обеспечивают ввод-вывод в стандартные потоки. Эти функции составляют стандартную библиотеку ввода-вывода в языке программирования C++.

При создании консольного процесса стандартные файлы и стандартные потоки ввода-вывода связываются с дескрипторами, которые заданы в полях **hStdinput**, **hStdoutput** и **hStdError** структуры типа STARTUPINFO.

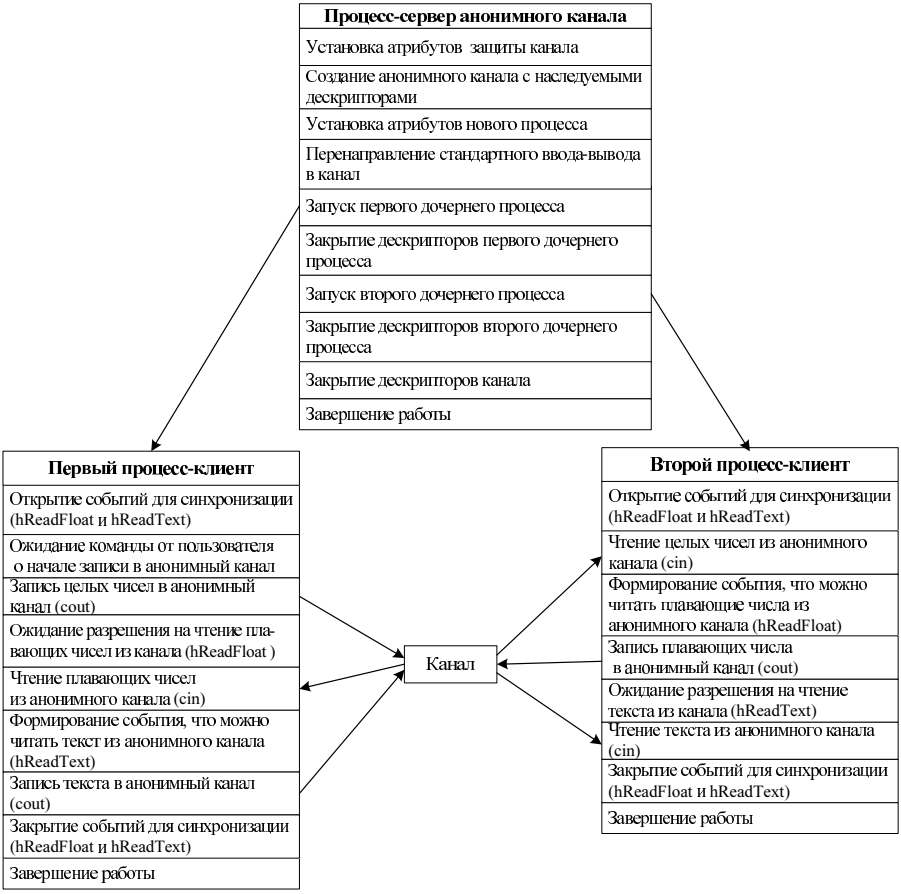
Поэтому если в эти поля будут записаны соответствующие дескрипторы анонимного канала, то для передачи данных по анонимному каналу можно использовать функции стандартного ввода-вывода. Такая процедура называется **перенаправлением стандартного ввода-вывода**.

```
    STARTUPINFO si;
    ……….
// установка стандартных дескрипторов (перенаправление ввода-вывода)
    si.hStdInput = hReadPipe;
    si.hStdOutput = hWritePipe;
    si.hStdError = hWritePipe;
```

Функции ввода-вывода из заголовочного файла **conio.h** отличаются от функций ввода-вывода из заголовочного файла **stdio.h** стандартной библиотеки языка программирования С тем, что они всегда работают с консолью. Поэтому эти функции можно использовать для ввода-вывода на консоль даже в случае перенаправления стандартного ввода-вывода.

Пример. Ниже приведена схема программы, в которой стандартный ввод-вывод перенаправляется в анонимный канал, а для обмена данными по анонимному каналу используются операторы ввода-вывода языка программирования С++.

Текст программы – в приложении 5.



Пояснения.

1. Процесс-сервер создает анонимный канал, два клиентских процесса и перенаправляет ввод-вывод, передавая клиентским процессам дескрипторы анонимного канала через поля структуры STARTUPINFO (там, где ранее были стандартные дескрипторы ввода-вывода).

2. Клиентские процессы осуществляют встречный обмен данными через анонимный канал, пользуясь стандартными дескрипторами ввода-вывода. Синхронизация встречной передачи данных осуществляется посредством события.

Для обмена используются операторы ввода-вывода (cin, cout).

4.3. ДРУГИЕ СРЕДСТВА ОБМЕНА ДАННЫМИ МЕЖДУ ПАРАЛЛЕЛЬНЫМИ ПРОЦЕССАМИ В ОС WINDOWS (обзор)

I. Именованные каналы

Именованный канал, в отличие от неименованного, имеет имя. Процесс, который создает именованный канал, называется **сервером именованного канала**. Процессы, которые связываются с именованным каналом, называются **клиентами именованного канала**.

Характеристики именованных каналов:

- имеют имя, которое используется клиентами для связи с именованным каналом;
- могут быть как полудуплексные, так и дуплексные;
- передача данных может осуществляться как потоком, так и сообщениями;
- обмен данными может быть как синхронным, так и асинхронным;
- возможность моделирования любой топологии связей.

Порядок работы с именованными каналами:

– создание именованного канала сервером;
– соединение сервера с экземпляром именованного канала;
– соединение клиента с экземпляром именованного канала;
– обмен данными по именованному каналу;
– отсоединение сервера от экземпляра именованного канала;
– закрытие именованного канала клиентом и сервером.

II. Почтовые ящики

Почтовым ящиком называется объект ядра ОС, который обеспечивает передачу сообщений от процессов-клиентов к процессам-серверам, выполняющимся на компьютерах в пределах локальной сети.

Процесс, который создает почтовый ящик, называется **сервером почтового ящика**. Процессы, которые связываются с именованным почтовым ящиком, называются **клиентами почтового ящика**.

Характеристики почтовых ящиков:

- имеют имя, которое используется клиентами для связи с почтовыми ящиками;
- направление передачи данных от клиента к серверу;
- передача данных осуществляется сообщениями;
- обмен данными может быть как синхронным, так и асинхронным.

Порядок работы с почтовым ящиком:

- создание почтового ящика сервером;
- соединение клиента с почтовым ящиком;
- обмен данными через почтовый ящик;
- закрытие почтового ящика клиентом и сервером.

В целом схема работы полностью аналогична схеме клиент-серверного взаимодействия Unix.

ПРИЛОЖЕНИЕ 1

Пример оконной Windows-программы для работы с потоками

```
//   CreateThreads.cpp
# include <windows.h>
# include <string>
using namespace std;
# include "KWnd.h"
enum UserMsg { UM_THREAD_D0NE = WM_USER+1 };
struct ThreadManager {
      ThreadManager (string _name) : name(_name) { nValue = 0;  }
        HWND hwndParent;
       string name;
        int nValue;
};
//   Структура, объекты которой tm_A, tm_B и tm_C используются
//   для взаимосвязи дочерних потоков с первичным потоком.
//   Адреса этих объектов передаются в качестве четвертого параметра
//   при вызовах функции CreateThread.
ThreadManager tm_A(string("Поток A"));
ThreadManager tm_B(string("Поток B"));
ThreadManager tm_C(string("Поток C"));
//   Прототипы используемых функций
DWORD WINAPI ThreadFuncA(LPVOID);
DWORD WINAPI ThreadFuncB(LPVOID);
DWORD WINAPI ThreadFuncC(LPVOID);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
////////////////////////////////////////////////////////////////////////////////
//   главная функция, в которой создается основное окно программы
//   и запускается цикл обработки сообщений;
Int WINAPI WinMain (HINSTANCE hinstance, HINSTANCE hPrevInstance,
                                  LPSTR IpCmdLine, int nCmdShow)
{
    MSG msg;
//   определение объекта mainWnd класса KWnd, в котором создается
//   и отображается окно
          KWnd mainWnd("CreateThreads", hinstance, nCmdShow,
                                  WndProc, NULL, 100, 100, 400, 160);
//        цикл обработки сообщений
        while (GetMessage(&msg, NULL, 0, 0))  // извлечение очередного сообщения
```

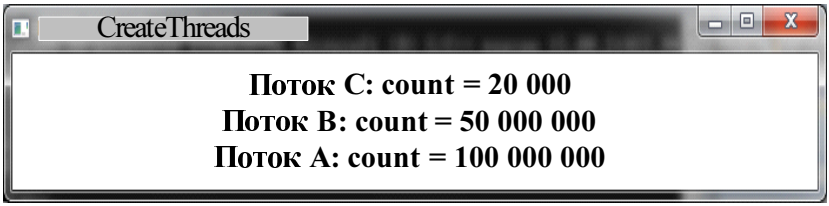
```
            { TranslateMessage(&msg);        // преобразование сообщения
              DispatchMessage(&msg); }   //передача структуры msg в Windows
    Return (msg.wParam);
}
////////////////////////////////////////////////////////////////////////////////////////////
//  Оконная процедура WndProc
LRESULT CALLBACK WndProc (HWND HWnd, UINT uMsg,
                                    WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    static HANDLE hThreadA, hThreadB, hThreadC;
    static char text[100];
    ThreadManager* pTm;
    static int y=0;
    switch (uMsg)
//      обработка сообщений
    {
    case WM_CREATE:
// создание трех дочерних потоков с дескрипторами hThreadA, hThreadB
// и hThreadC в блоке обработки сообщения WM_CREATE
        tm_A.hwndParent = hWnd;
        hThreadA = CreateThread(NULL, 0, ThreadFuncA, &tm_A, 0, NULL);
        if (!hThreadA)
            MessageBox (hWnd, "Error of create hThreadA", NULL, MB_OK);
        tm_B.hwndParent = hWnd
        hThreadB = CreateThread(NULL l, 0, ThreadFuncB, &tm_B, 0, NULL);
        if (!hThreadB)
            MessageBox (hWnd, "Error of create hThreadB", NULL, MB_OK);
        tm_C.hwndParent = hWnd
        hThreadC = CreateThread(NULL, 0, ThreadFuncC, &tm_C, 0, NULL);
        if (!hThreadC)
            MessageBox (hWnd, "Error of create hThreadC", NULL, MB_OK);
        break;
    case UM_THREAD_DONE:
//  обработка сообщения, посланного функцией SendMessage дочерним потоком
            pTm = (ThreadManager*)wParam;
            sprintf(text, "%s: count = %d", pTm->name.c_str(), pTm->nValue);
//  Подготовка строки о завершившемся потоке
             y += 30;    //  координата строки вывода
            InvalidateRect (hWnd, NULL, FALSE);  // генерация сообщения
// WM_PAINT
```

```
            break;
    case WM_PAINT:
//  перерисовка окна – вывод строки
            hDC = BeginPaint(hWnd, &ps);
            TextOut (hDC, 20, y, text, strlen(text));     // вывод строки в окно
            EndPaint(hWnd, &ps);
            break;
    case WM_DESTROY:
//  уничтожение дескриптором потоков при закрытии окна
            CloseHandle (hTreadA);
            CloseHandle (hTreadB);
            CloseHandle (hTreadC);
            PostQuitMessage(0);
            break;
    default:
            return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
    return 0;
}
//////////////////////////////////////////////////////////////
//  Функции потоков ThreadFuncA, ThreadFuncB, ThreadFuncC
DWORD WINAPI ThreadFuncA(LPVOID lpv)
{
        ThreadManager* pTm = (ThreadManager*)lpv;
        int count = 0;
        for (int i = 0;  i < 100000000;  ++i) count++;
        pTm->nValue = count:
        SendMessage (pTm->hwndParent, UM_THREAD_DONE, (WPARAM)pTm, 0);
//      Посылка окну первичного потока пользовательского сообщения
// UM_THREAD_D0NE
          return 0:
}
//////////////////////////////////////////////////////////////
DWORD WINAPI ThreadFuncB(LPVOID lpv)
{
        ThreadManager* pTm = (ThreadManager*)lpv;
        int count = 0;
        for (int i = 0;  i < 50000000;  ++i) count++;
        pTm->nValue = count:
        SendMessage (pTm->hwndParent, UM_THREAD_DONE, (WPARAM)pTm, 0);
//  Посылка окну первичного потока пользовательского сообщения
//  UM_THREAD_D0NE
          return 0:
```

```
}
////////////////////////////////////////////////////////////
DWORD WINAPI ThreadFuncC(LPVOID lpv)
{
   ThreadManager* pTm = (ThreadManager*)lpv;
   int count = 0;
   for (int i = 0;  i < 20000;  ++i) count++;
   pTm->nValue = count:
   SendMessage (pTm->hwndParent, UM_THREAD_DONE, (WPARAM)pTm, 0);
//   Посылка окну первичного потока пользовательского сообщения
//   UM_THREAD_D0NE
         return 0:
}
////////////////////////////////////////////////////////////////////
-------------------------------------------------------------------------------------------------------------
//   Определение класса KWnd, используемого в процедуре WinMain
// KWnd.h
// Заголовочный файл
#include <windows.h>
// Интерфейс класса
Class KWnd {
public:
         KWnd(LPCTSTR windowName, HINSTANCE hInst, int cmdShow,
         LRESULT (WINAPI *pWndProc)(HWND, UINT, WPARAM, LPARAM),
         LPCTSTR menuName = NULL,
         int x = CW_USEDEFAULT, int y = 0,
         int width = CW_USEDEFAULT, int height = 0,
         UINT classStyle = CS_HREDRAW | CS_VREDRAW,
    DWORD windowStyle = WS_OVERLAPPEDWINDOW,
    HWND hParent = NULL);
    HWND GETHWnd()    { return hWnd;  }
protected:
    HWND hWnd;
    WNDCLASSEX wc;
};
////////////////////////////////////////////////////////////
//  KWnd.cpp
# include “KWnd.h”
// Заголовок конструктора класса
KWnd :: KWnd(LPCTSTRwindowName, HINSTANCE hInst, int cmdShow,
        LRESULT (WINAPI *pWndProc)(HWND, UINT, WPARAM, LPARAM),
        LPCTSTR menuName, int x, int y, int width, int height,
        UINT classStyle, DWORD windowStyle, HWND hParent)
```

```
{
    char szClassName[] = "KWndClass";
    wc.cbSize        = sizeof(wc);
    wc.style         = classStyle;
    wc.lpfnWndProc   = pWndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.HInstance = hInst;
    wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wc.Cursor        = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground    = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName     = menuName;
    wc.lpszClassName    = szClassName;
    wc.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);
// Тело конструктора класса
// Код, отвечающий за подготовку и создание окна
// Регистрация класса окна
    if (! RegisterClassEx(&wc))  {
        char msg[100] = "Cannot register class: ";
        strcat(msg, szClassName);
        MessageBox(NULL, msg, "Error", MB_OK);
        return;     }
// Создание окна
    hWnd = CreateWindow(szClassName, windowName, windowStyle, x, y, width,
                height, hParent, (HMENU)NULL, hInst, NULL);
    if (!hWnd) {
        char msg[100] = "Cannot create window: ";
                strcat(msg, windowName);
        MessageBox(NULL, text, "Error", MB_OK);
        return;     }
// Отображение окна
    ShowWindow(hWnd, cmdShow);
}
```

Результат:



ПРИЛОЖЕНИЕ 2

Пример нарушения синхронизации при работе с общей глобальной переменной

```
#include <windows.h>
#include <stdio.h>
#include "KWnd.h"
#define N 50000000
long g_counter = 0;
//   Описание прототипов используемых функций
DWORD WINAPI ThreadFunc (LPVOID);
void IncCount();
void DecCount();
LRESULT CALLBACK WindProc(HWND, UINT, WPARAM, LPARAM);
//////////////////////////////////////////////////////////////////////////////
int WINAPI WinMain (HINSTANCE hinstance, HINSTANCE hPrevInstance,
                    LPSTR IpCmdLine, int nCmdShow)     //   главная функция
{
    MSG msg;
//   определение объекта mainWnd класса KWnd, в котором создается
//   и отображается окно
 KWnd mainWnd("BadCount", hinstance, nCmdShow, WndProc, NULL, 100, 100, 400, 100);
    while (GetMessage(&msg, NULL, 0, 0))   //   извлечение очередного сообщения
          {  TranslateMessage(&msg);      //   преобразование сообщения
             DispatchMessage(&msg);  }    // передача структуры msg в Windows
    return (msg.wParam);
}
//   Оконная процедура WndProc
LRESULT CALLBACK WndProc (HWND HWnd, UINT uMsg,
                                          WPARAM wParam, LPARAM lParam)
{   HDC hDC;
    PAINTSTRUCT ps;
    HANDLE hThread;
    char text[100];
    switch (uMsg)                          //   обработка сообщений
          {   case WM_CREATE:                  //   создание нового потока
              hThread = CreateThread(NULL, 0, ThreadFunc, NULL, 0, NULL);
              if (!hThread)
                   MessageBox (hWnd, "Error of create hThread", NULL, MB_OK);
              DecCount();
              WaitForSingleObjectChThread (ChThread, INFINITE);
//   приостановка выполнения потока на неограниченное время (IN FINITE)
              InvalideRect (hWnd, NULL, TRUE);
```

```
                break;
        case WM_PAINT:          //  перерисовка окна
                hDC = BeginPaint(hWnd, &ps);
                sprintf(text, "g_counter = %d", g_counter);
                TextOut(hDC, 20, 20, text, strlen(text));
                EndPaint(hWnd, &ps);
                break;
        case WM_DESTROY:        //  закрытие окна
                PostQuitMessage(0);
                break;
        default:
                return DefWindowProc(hWnd, uMsg, wParam, lParam);
        }
        return 0;
}
//////////////////////////////////////////////////////////////
DWORD WINAPI ThreadFunc (LPVOID lpv)  //  функция созданного потока
{   IncCount();
    Return 0;   }
void IncCount()         //  функция, вызываемая дочерним потоком
{   for (int i = 0; I < N; ++)  ++g_counter;   }
void DecCount()         //  функция, вызываемая основным потоком
{   for (int i = 0; I < N; ++)  --g_counter; }
```

ПРИЛОЖЕНИЕ 3

Синхронизация потоков

1. Синхронизация потоков при помощи событий с ручным сбросом в рамках одного процесса

```
#include <windows.h>
#include <iostream.h>
HANDLE hOutEvent[2], hAddEvent;        //  дескрипторы событий
DWORD WINAPI thread_1(LPVOID)       // первый дочерний поток
{
    for (int i = 0; i < 10; ++i)
    if (i = = 4)
    {      //      закончена половина работы
             SetEvent(hOutEvent[0]);  //  перевод события в сигнальное состояние
                WaitForSingleObject(hAddEvent, INFINITE);
    }
    return 0;
}
//////////////////////////////////////
DWORD CALLBACK thread_2(LPVOID)       // второй дочерний поток
{
    for (int i = 0; i < 10; ++i)
     if (i = = 4)
    {      //      закончена половина работы
             SetEvent(hOutEvent[1]);   // перевод события в сигнальное состояние
                WaitForSingleObject(hAddEvent, INFINITE);
    }
    return 0;
}
//////////////////////////////////////////////////////////////
int main()
{
     HANDLE  hThread_1, hThread_2;
     DWORD  IDThread_1, IDThread_2;
// создание событий с автоматическим сбросом
     hOutEvent[0] = CreateEvent(NULL, FALSE, FALSE, NULL);
     if (hOutEvent[0] == NULL)
                    return GetLastError();
     hOutEvent[1] = CreateEvent(NULL, FALSE, FALSE, NULL);
     if (hOutEvent[1] == NULL)
                    return GetLastError();
```

```
// создание событий с ручным сбросом
      hAddEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
      if (hAddEvent == NULL)
            return GetLastError();
// создание потоков
      hThread_1 = CreateThread(NULL, 0, thread_1, NULL, 0, &IDThread_1);
      if (hThread_1 == NULL)
            return GetLastError();
      hThread_2 = CreateThread(NULL, 0, thread_2, NULL, 0, &IDThread_2);
      if (hThread_2 == NULL)
            return GetLastError();
// ожидание, пока потоки выполнят половину работы
      WaitForMultipleObjects(2, hOutEvent, TRUE, INFINITE);
//    количество ожидаемых объектов – 2;
//    hOutEvent – массив дескрипторов объектов;
//    значение TRUE третьего параметра определяет,
//    что эта функция в течение интервала времени,
//    заданного четвертым параметром, ждет, пока все объекты синхронизации,
//    дескрипторы которых заданы вторым параметром,
//    перейдут в сигнальное состояние.
//    (при значении третьего параметра FALSE функция в
//    течение интервала времени, заданного четвертым параметром,
//    ждет, пока любой из объектов синхронизации, дескрипторы которых
//    заданы вторым параметром, перейдет в сигнальное состояние).
//    INFINITE – бесконечно долгое ожидание
      cout << "A half of the work is done." << endl;
      cout << "Press any key to continue." << endl;
      cin.get();
//    разрешение потокам продолжать работу
 PulseEvent(hAddEvent);   // вывод дочерних потоков из состояния ожидания
                          // и переход события в сигнальное состояние
  WaitForSingleObject(hThread_1, INFINITE); // ожидание завершения потоков
      WaitForSingleObject(hThread_2, INFINITE);
      CloseHandle(hThread_1);              // закрытие дескрипторов
      CloseHandle(hThread_2);
      CloseHandle(hOutEvent[0]);
      CloseHandle(hOutEvent[1]);
      CloseHandle(hAddEvent);
      cout << "The work is done." << endl;
      return 0;
}
```

2. Синхронизация потоков при помощи событий с автоматическим сбросом в рамках различных процессов

Родительский процесс, ожидающий наступления события

```
#include <windows.h>
#include <iostream.h>
HANDLE hInEvent;
CHAR lpEventName[] = "InEventName";
int main()
{
    DWORD dwWaitResult;
    char szAppName[] = "C:\\ConsoleProcess.exe";
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
//  создание события, отмечающее ввод символа
    hInEvent = CreateEvent(NULL, FALSE, FALSE, lpEventName);
    if (hInEvent == NULL)
            return GetLastError();
    ZeroMemory(&si, sizeof(STARTUPINFO));
    si.cb = sizeof(STARTUPINFO);
//  запуск процесса, ожидающего ввода символа
    if (!CreateProcess(szAppName, NULL, NULL, NULL, FALSE,
                    CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi))
            return 0;
    CloseHandle(pi.hProcess);       // закрытие дескрипторов процесса
    CloseHandle(pi.hThread);
//  ожидание оповещения о наступлении события о вводе символа
    dwWaitResult = WaitForSingleObject(hInEvent, INFINITE);
    if (dwWaitResult != WAIT_OBJECT_0)
        return dwWaitResult;
    cout << "A symbol has got." << endl;
    CloseHandle(hInEvent);          // закрытие дескрипторов события
    cout << "Press any key to exit.";
    cin.get();
    return 0;
}
```

Дочерний процесс, устанавливающий событие в сигнальное состояние

```
#include <windows.h>
#include <iostream.h>
HANDLE hInEvent;
CHAR lpEventName[]="InEventName";
int main()
```

```
{
    char c;
//      Получение доступа к событию
    hInEvent = OpenEvent (EVENT_MODIFY_STATE, FALSE, lpEventName);
    if (hInEvent = = NULL)
    {       //   Обработка ошибочного состояния
            cout << "Open event failed." << endl;
            cout << "Input any char to exit." << endl;
            cin.get();
          return GetLastError();
    }
    cout << "Input any char: ";
    cin >> c;
    SetEvent(hInEvent);       // перевод события в сигнальное состояние
                                                  // после ввода символа
    CloseHandle(hInEvent);    // закрытие дескриптора события в текущем
процессе
    cin.get();
    cout << "Press any key to exit." << endl;
    cin.get();
    return 0;
}
```

ПРИЛОЖЕНИЕ 4

Обмен данными через анонимный канал двух процессов

I. Вариант наследуемого дескриптора

Программа процесса-клиента анонимного канала

```
#include <windows.h>
#include <conio.h>
int main(int argc, char *argv[])
{
    HANDLE hWritePipe;
//   преобразование символьного представления дескриптора в число
    hWritePipe = (HANDLE) atoi(argv[1]);
//  ожидание команды о начале записи в анонимный канал
    _cputs("Press any key to start communication.\n");
    _getch();
    for (int i = 0; i < 10; i++)
    {                    // запись в анонимный канал
                DWORD dwBytesWritten;
                if (!WriteFile (hWritePipe, &i, sizeof(i), &dwBytesWritten, NULL))
        {         _cputs("Write to file failed.\n");      // ошибка при записи в канал
            _cputs("Press any key to finish.\n");
            _getch();
            return GetLastError();
        }
        _cprintf("The number %d is written to the pipe.\n", i);
        Sleep(500);
    }
     CloseHandle(hWritePipe);       // закрытие дескриптора канала
     _cputs("The process finished writing to the pipe.\n");
     _cputs("Press any key to exit.\n");
     _getch();
    return 0;
}
```

Программа процесса-сервера анонимного канала

```
#include <windows.h>
#include <conio.h>
int main()
{
    char lpszComLine[80];  // для командной строки
```

```
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    HANDLE hWritePipe, hReadPipe;
    SECURITY_ATTRIBUTES sa;
// установка атрибутов защиты канала
    sa.nLength = sizeof(SECURITY_ATTRIBUTES);
    sa.lpSecurityDescriptor = NULL;          // защита по умолчанию
    sa.bInheritHandle = TRUE;                // дескрипторы наследуемые
// создание анонимного канала
    if(!CreatePipe (
        &hReadPipe,     // дескриптор для чтения
        &hWritePipe,    // дескриптор для записи
        &sa,            // атрибуты защиты по умолчанию,
                // дескрипторы hReadPipe и hWritePipe наследуемые
     0))                    // размер буфера по умолчанию
      {   _cputs("Create pipe failed.\n");  // ошибка при создании канала
                _cputs("Press any key to finish.\n");
          _getch();
           return GetLastError();
      }
//  установавка атрибутов нового процесса
    ZeroMemory(&si, sizeof(STARTUPINFO));
    si.cb = sizeof(STARTUPINFO);
//  формирование командной строки дескриптором канала hWritePipe
    wsprintf (lpszComLine, "C:\\Client.exe %d", (int)hWritePipe);
//  запуск нового консольного процесса
    if (!CreateProcess (
        NULL,           // имя процесса
        lpszComLine,    // командная строка
        NULL,           // атрибуты защиты процесса по умолчанию
        NULL,           // атрибуты защиты первичного потока по умолчанию
        TRUE,           // наследуемые дескрипторы текущего процесса
                // наследуются новым процессом
        CREATE_NEW_CONSOLE, // новая консоль
        NULL,           // используем среду окружения процесса-предка
        NULL,           // текущий диск и каталог, как и в процессе-предке
        &si,            // вид главного окна - по умолчанию
        &pi))           // здесь будут дескрипторы и идентификаторы
                // нового процесса и его первичного потока
    {   _cputs("Create process failed.\n");
        _cputs("Press any key to finish.\n");
        _getch();
        return GetLastError();
```

```
    }
    CloseHandle(pi.hProcess);        // закрытие дескрипторов нового процесса
    CloseHandle(pi.hThread);         //  (в данном процессе они не используются)
    for (int i = 0; i < 10; i++)
    {             // чтение из анонимного канала
   int nData;
   DWORD dwBytesRead;
   if (!ReadFile(                 // ошибка при записи в канал
            hReadPipe,
            &nData,
            sizeof(nData),
            &dwBytesRead,
            NULL))
       {        _cputs("Read from the pipe failed.\n");
                    _cputs("Press any key to finish.\n");
                    _getch();
                    return GetLastError();
       }
        _cprintf("The number %d is read from the pipe.\n", nData);
    }
   CloseHandle(hReadPipe);          // закрытие дескриптора канала hReadPipe
   _cputs("The process finished reading from the pipe.\n");
   _cputs("Press any key to exit.\n");
   _getch();
   return 0;
}
```

II. Вариант ненаследуемого дескриптора

Программа процесса-клиента анонимного канала

```
#include <windows.h>
#include <conio.h>
int main(int argc, char *argv[])
{
    HANDLE hWritePipe;
//   преобразование символьного представления дескриптора в число
    hWritePipe = (HANDLE) atoi(argv[1]);
//  ожидание команды о начале записи в анонимный канал
    _cputs("Press any key to start communication.\n");
    _getch();
    for (int i = 0; i < 10; i++)
    {                    // запись в анонимный канал
    DWORD dwBytesWritten;
```

```
    if (!WriteFile (hWritePipe, &i, sizeof(i), &dwBytesWritten, NULL))
        {       _cputs("Write to file failed.\n");      // ошибка при записи в канал
            _cputs("Press any key to finish.\n");
            _getch();
            return GetLastError();
        }
        _cprintf("The number %d is written to the pipe.\n", i);
        Sleep(500);
    }
     CloseHandle(hWritePipe);       // закрытие дескриптора канала
     _cputs("The process finished writing to the pipe.\n");
     _cputs("Press any key to exit.\n");
     _getch();
    return 0;
}
```

Программа процесса-сервера анонимного канала

```
#include <windows.h>
#include <conio.h>
int main()
{
    char lpszComLine[80];  // для командной строки
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    HANDLE hWritePipe, hReadPipe, hInheritWritePipe;
// создание анонимного канала
    if(!CreatePipe (
        &hReadPipe,       // дескриптор для чтения
        &hWritePipe,      // дескриптор для записи
        NULL,          // атрибуты защиты по умолчанию, в этом случае
                    // дескрипторы hReadPipe и hWritePipe ненаследуемые
     0))                // размер буфера по умолчанию
      {    _cputs("Create pipe failed.\n");   // ошибка при создании канала
       _cputs("Press any key to finish.\n");
            _getch();
      return GetLastError();
       }
//   создание наследуемого дубликата дескриптора hWritePipe
     if (!DuplicateHandle (
          GetCurrentProcess(),          // дескриптор текущего процесса
        hWritePipe,                 // исходный дескриптор канала
        GetCurrentProcess(),        // дескриптор текущего процесса
        &hInheritWritePipe,         // новый дескриптор канала
```

```
        0,                          // этот параметр игнорируется
        TRUE,                       // новый дескриптор наследуемый
        DUPLICATE_SAME_ACCESS ))   // доступ не изменяем
    {     _cputs("Duplicate handle failed.\n");      // ошибка при наследовании
                                                     //  дескриптора
        _cputs("Press any key to finish.\n");
        _getch();
        return GetLastError();
    }
CloseHandle(hWritePipe);  // закрытие оригинального дескриптора hWritePipe
//   установка атрибутов нового процесса
    ZeroMemory(&si, sizeof(STARTUPINFO));
    si.cb = sizeof(STARTUPINFO);
// формирование командной строки с дубликатом дескриптора hInheritWritePipe
    wsprintf (lpszComLine, "C:\\Client.exe %d", (int)hInheritWritePipe);
//    запуск нового консольного процесса
    if (!CreateProcess (
        NULL,          // имя процесса
        lpszComLine,      // командная строка
        NULL,          // атрибуты защиты процесса по умолчанию
        NULL,          // атрибуты защиты первичного потока по умолчанию
        TRUE,          // наследуемые дескрипторы текущего процесса
                    // наследуются новым процессом
        CREATE_NEW_CONSOLE, // новая консоль
        NULL,          // используем среду окружения процесса-предка
        NULL,          // текущий диск и каталог, как и в процессе-предке
        &si,           // вид главного окна - по умолчанию
        &pi))          // здесь будут дескрипторы и идентификаторы
                    // нового процесса и его первичного потока
    {     _cputs("Create process failed.\n");
        _cputs("Press any key to finish.\n");
        _getch();
        return GetLastError();
    }
    CloseHandle(pi.hProcess);       // закрытие дескрипторов нового процесса
    CloseHandle(pi.hThread);       //  (в данном процессе они не используются)
    CloseHandle(hInheritWritePipe); // закрытие дубликата дескриптора
 // hInheritWritePipe
    for (int i = 0; i < 10; i++)
    {             // чтение из анонимного канала
    int nData;
    DWORD dwBytesRead;
    if (!ReadFile(              // ошибка при записи в канал
```

```
            hReadPipe,
                    &nData,
            sizeof(nData),
            &dwBytesRead,
            NULL))
        {       _cputs("Read from the pipe failed.\n");
            _cputs("Press any key to finish.\n");
            _getch();
            return GetLastError();
        }
        _cprintf("The number %d is read from the pipe.\n", nData);
    }
    CloseHandle(hReadPipe);          // закрытие дескриптора канала hReadPipe
    _cputs("The process finished reading from the pipe.\n");
    _cputs("Press any key to exit.\n");
    _getch();
    return 0;
}
```

ПРИЛОЖЕНИЕ 5

Перенаправление стандартного ввода-вывода

Первый процесс-клиент

```
#include <windows.h>
#include <conio.h>
#include <iostream.h>
int main()
{ HANDLE hReadFloat, hReadText; // события для синхронизации обмена данными
      char lpszReadFloat[] = "ReadFloat";
      char lpszReadText[] = "ReadText";
      hReadFloat = CreateEvent(NULL, FALSE, FALSE, lpszReadFloat);
// открытие событий
      hReadText = CreateEvent(NULL, FALSE, FALSE, lpszReadText);
       _cputs("Press any key to start communication.\n");
       _getch();       // ожидание команды о начале записи в анонимный канал
       for (int i = 0; i < 5; ++i)
       {    Sleep(500);
            cout << i << endl;       // запись целых числел в анонимный канал
       }
  //       ожидание разрешения на чтение плавающих чисел из канала
       WaitForSingleObject(hReadFloat, INFINITE);
       for (int j = 0; j < 5; ++j)
       { float nData;
     cin >> nData;      //  чтение плавающих числел из анонимного канала
       _cprintf("The number %2.1f is read from the pipe.\n", nData);
       }
  SetEvent(hReadText);   //     разрешение, что можно читать текст из канала
       cout << "This is a demo sentence." << endl;          //   передача текста
       cout << '\0' << endl;                        //     конец передачи
       _cputs("The process finished transmission of data.\n");
       _cputs("Press any key to exit.\n");
       _getch();
       CloseHandle(hReadFloat);
       CloseHandle(hReadText);
       return 0;
}
```

Второй процесс-клиент

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

```
int main()
{ HANDLE hReadFloat, hReadText; // события для синхронизации обмена данными
     char lpszReadFloat[] = "ReadFloat";
     char lpszReadText[] = "ReadText";
     hReadFloat = CreateEvent(NULL, FALSE, FALSE, lpszReadFloat);
//   открытие событий
     hReadText = CreateEvent(NULL, FALSE, FALSE, lpszReadText);
    for (int i = 0; i < 5; ++i)
    {    int nData;
            cin >> nData;          //   чтение целых чисел из анонимного канала
            _cprintf("The number %d is read from the pipe.\n", nData);
    }
    SetEvent(hReadFloat);  //    разрешение читать плавающие числа
                                                    //     из анонимного канала
    for (int j = 0; j < 5; ++j)
    {    Sleep(500);
         cout << (j*0.1) << endl;  // запись плавающих чисел в анонимный канал
    }
    WaitForSingleObject(hReadText, INFINITE);     //    ожидание разрешения
                                                    //    на чтение текста
    _cputs("The process read the text: ");          //     чтение текста
    char lpszInput[80];
    do
    {    Sleep(500);
         cin >> lpszInput;
         _cputs(lpszInput);
         _cputs(" ");
    }
    while (*lpszInput != '\0');
    _cputs("\nThe process finished transmission of data.\n");
    _cputs("Press any key to exit.\n");
    _getch();
   CloseHandle(hReadFloat);
   CloseHandle(hReadText);
   return 0;
}
```

Процесс-сервер

```
#include <windows.h>
#include <conio.h>
int main()
{    char lpszComLine1[80] = "C:\\Client1.exe";       //   имя первого клиента
```

```
    char lpszComLine2[80] = "C:\\Client2.exe";        //  имя второго клиента
          STARTUPINFO si;
          PROCESS_INFORMATION pi;
          HANDLE hWritePipe, hReadPipe;
          SECURITY_ATTRIBUTES sa;
  // установка атрибутов защиты канала
          sa.nLength = sizeof(SECURITY_ATTRIBUTES);
          sa.lpSecurityDescriptor = NULL;     //    защита по умолчанию
          sa.bInheritHandle = TRUE;        //    дескрипторы наследуемые
          if(!CreatePipe(                 //    создание анонимного канала
             &hReadPipe,                  //    дескриптор для чтения
             &hWritePipe,                 //    дескриптор для записи
        &sa,   // атрибуты защиты по умолчанию, дескрипторы наследуемые
             0))                          //    размер буфера по умолчанию
        {     _cputs("Create pipe failed.\n");//    ошибка создания канала
         _cputs("Press any key to finish.\n");
         _getch();
                    return GetLastError();
        }
//   установка атрибутов нового процесса
          ZeroMemory(&si, sizeof(STARTUPINFO));
          si.cb = sizeof(STARTUPINFO);
//    указание использовать стандартные дескрипторы
          si.dwFlags = STARTF_USESTDHANDLES;
//    установка стандартных дескрипторов (перенаправление ввода-вывода)
          si.hStdInput = hReadPipe;
          si.hStdOutput = hWritePipe;
          si.hStdError = hWritePipe;
          if (!CreateProcess( // запуск первого клиента
              NULL,             //    имя процесса
              lpszComLine1,       //    командная строка
            NULL,           //    атрибуты защиты процесса по умолчанию
            NULL,     //    атрибуты защиты первичного потока по умолчанию
            TRUE,           //    наследуемые дескрипторы текущего процесса
                        //    наследуются новым процессом
            CREATE_NEW_CONSOLE,     //    создание новой консоли
            NULL,     //    использование среды окружения процесса предка
            NULL,     //    текущего диска и каталога, как и в процессе предке
            &si,              //    вид главного окна - по умолчанию
            &pi))             //    здесь будут дескрипторы и идентификаторы
                      //    нового процесса и его первичного потока
          {     _cputs("Create process failed.\n"); //    ошибка создания процесса
        _cputs("Press any key to finish.\n");
```

```
        _getch();
        return GetLastError();
    }
    CloseHandle(pi.hProcess);  // закрытие дескрипторов первого клиента
        CloseHandle(pi.hThread);
        if (!CreateProcess( // запуск второго клиента
            NULL,           //  имя процесса
            lpszComLine2,   //  командная строка
          NULL,         //  атрибуты защиты процесса по умолчанию
          NULL,     //  атрибуты защиты первичного потока по умолчанию
          TRUE,         //  наследуемые дескрипторы текущего процесса
                        //  наследуются новым процессом
          CREATE_NEW_CONSOLE,    //  создание новой консоли
          NULL,         //  использование среды окружения процесса предка
          NULL,     //  текущего диска и каталога, как и в процессе предке
          &si,              //  вид главного окна - по умолчанию
          &pi))             //  здесь будут дескрипторы и идентификаторы
                        //  нового процесса и его первичного потока
        {   _cputs("Create process failed.\n"); //  ошибка создания процесса
      _cputs("Press any key to finish.\n");
        _getch();
        return GetLastError();
    }
    CloseHandle(pi.hProcess); //  закрытие дескрипторов второго клиента
        CloseHandle(pi.hThread);
        CloseHandle(hReadPipe);      //  закрытие дескрипторов канала
        CloseHandle(hWritePipe);
        _cputs("The clients are created.\n");
        _cputs("Press any key to exit.\n");
        _getch();
        return 0;
}
```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Побегайло А.* Системное программирование в Windows / А. Побегайло. – СПб.: БХП-Петербург, 2006.

2. *Щупак Ю.* Win32 API. Эффективная разработка приложений / Ю. Щупак. – СПб.: Питер, 2007.

3. *Рихтер Д.* Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Д. Рихтер. – Спб.: Питер, 2001.

4. *Стасышин В.М.* Управление ресурсами в ОС Windows: // Метод-дич. указания к выполнению индивидуальной работы по курсу «Управление ресурсами в вычислительных системах» / В.М. Стасышин, И.М. Куликов. – Новосибирск: Изд-во НГТУ, 2012.

ОГЛАВЛЕНИЕ

Стасышин Владимир Михайлович

УПРАВЛЕНИЕ РЕСУРСАМИ
В ОС WINDOWS

Конспект лекций

Редактор *И.Л. Кескевич*
Выпускающий редактор *И.П. Брованова*
Корректор *И.Е. Семенова*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *Л.А. Веселовская*

Подписано в печать 28.08.2013. Формат 60 × 84 1/16. Бумага офсетная. Тираж 70 экз.
Уч.-изд. л. 6,04. Печ. л. 6,5. Изд. № 74. Заказ № Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630073, г. Новосибирск, пр. К. Маркса, 20