ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ имени академика С. П. КОРОЛЕВА» (САМАРСКИЙ УНИВЕРСИТЕТ)

ПРОГРАММИРОВАНИЕ МНОГОЗАДАЧНОСТИ В WINDOWS

Самара 2017

РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

ПРОГРАММИРОВАНИЕ МНОГОЗАДАЧНОСТИ В WINDOWS

Составитель К.Е. Климентьев

Самара
Издательство Самарского университета
2017

УДК 004.451.46
ББК 32.973

Составитель ***К.Е.Климентьев***

Рецензент: к.т.н., доцент А.В. Баландин

Программирование мнозадачности в Windows: [Электронный ресурс]: метод. указания / сост. *К.Е. Климентьев*. - Самара: Изд-во Самарского университета, 2017. – 43 с. : ил. Электрон. текстовые и граф. дан. (Кбайт).- 1 эл. опт. диск (CD-ROM)

Методические указания к лабораторному практикуму по курсу «Системы реального времени».

Предназначены для студентов, изучающих в рамках направления подготовки 09.03.01 «Информатика и вычислительная техника» курс «Системы реального времени» и прочие курсы аналогичной тематики.

Содержат необходимый теоретический и справочный материал для выполнения лабораторных работ. Также могут быть использованы в курсовом проектировании и при разработке выпускной квалификационной работы.

Подготовлены на кафедре информационных систем и технологий.

УДК 004.056.55
ББК 32.973

Оглавление

Задание к лабораторной работе

Цель лабораторной работы: практическое изучение методов и средств многозадачного программирования в операционных системах семейства Windows.

Задание: написать, отладить и продемонстрировать преподавателю систему из нескольких независимых процессов (A) или потоков (B), совместно решающих квадратное уравнение (K), вычисляющих гипотенузу по двум катетам (L) или выборочную дисперсию для трех чисел (M). Процессы или потоки должны отображать на экране ход своего выполнения в виде отладочных сообщений. Система должны состоять из:

- главного процесса или потока, принимающего с клавиатуры исходные данные и выводящего на экран результат;
- нескольких служебных процессов или потоков, способных по отдельности выполнять элементарные арифметические действия - сложение, вычитание, умножение, деление, вычисление квадратного корня и т.п.

При этом использовать способ синхронизации C, D, E или F и способ передачи данных между процессами или потоками - G, H, I или J.

Таблица 1. Индивидуальные задания

Вариант	Задание	Вариант	Задание	Вариант	Задание
1	ACGK	10	BEHK	19	ACIK
2	BDHL	11	AFIL	20	BEJL
3	AEIM	12	BDJM	21	AFGM
4	BFJK	13	AEGK	22	BCHK
5	ACGL	14	BEIL	23	AEIL
6	BEHM	15	AFHM	24	BDJK
7	AFIK	16	BCJK	25	AFGK
8	BDJL	17	AEGL	26	BCHL
9	ACJM	18	BFHM	27	AEIM

Условные обозначения:

- A – независимые процессы;
- B – потоки одного процесса;
- C – события;
- D – критические секции;
- E – семафоры;
- F – мьютексы;
- G – обычные файлы;
- H – каналы;
- I – почтовые ячейки;
- J – файлы, отображаемые в память;
- K – квадратное уравнение $(x_1, x_2) = (-b \pm \sqrt{D})/2a;\ D = b^2 - 4ac;;$
- L – гипотенуза по двум катетами $c = \sqrt{a^2 + b^2}$;
- M – выборочная дисперсия трех чисел $\tilde{\sigma}_x^2 = \frac{1}{2}\sum_{i=1}^{3}(x_i - \bar{x})^2;.$

Язык программирования – любой. Использование высокоуровневых объектно-ориентированных «оберток» для API-функций запрещено!

Теоретическая часть

Многозадачность – это режим работы операционной системы, при котором параллельно могут выполняться несколько программ (задач). Для создания эффекта одновременности программы (задачи) выполняются поочередно короткими фрагментами.

Существуют две разновидности многозадачности:

- *кооперативная* – при которой задача сама принимает решение о передаче управления другой задаче;
- *вытесняющая* – при которой операционная система через небольшие *кванты времени* принудительно прерывает выполнение одной задачи, чтобы передать управление другой.

В современных версиях операционных систем Windows реализована вытесняющая многозадачность.

Процесс – программа, монопольно занимающая выделенное ей адресное пространство и владеющая некоторым набором ресурсов.

Поток – набор последовательно выполняющихся команд. В каждом процессе, т.е. в общем адресном пространстве, могут развиваться один или несколько потоков. Все потоки одного процесса располагаются в одном общем адресном пространстве и конкурируют за процессорное время друг с другом и с другими потоками других процессов. Отдельной задачей в Windows является именно поток.

Если задачи выполняются согласованно друг с другом, то говорят, что их выполнение *синхронизировано*. В стандартные библиотеки Windows включены многочисленные функции, обеспечивающие синхронизацию задач и обмен данными между ними.

1. Запуск и завершение процессов и потоков

Для запуска и завершения задач различных типов используются следующие функции:

- CreateProcess() – создает новый процесс с его главным потоком и возвращает указатель на структуру типа PROCESS_INFORMATION, содержащую дескрипторы и идентификаторы для созданных процесса и потока;
- WinExec() – производит упрощенный запуск внешнего процесса;
- ExitProcess() – вызывается изнутри процесса и завершает процесс и все его потоки;
- TerminateProcess() – запускается извне процесса и завершает процесс и все его потоки;
- CreateThread() – создает (по указателю на процедуру) новый поток (thread) внутри вызывающего потока;
- TerminateThread() – вызывается изнутри потока (thread) и завершает его;
- ExitThread() – вызывается извне потока (thread) и завершает его.

Для совместимости со старыми версиями Windows реализованы «фиберы», которые обеспечивают передачу потоками друг другу управления по правилам «кооперативной» многозадачности:

- ConvertThreadToFiber() – создает «главный фибер» из «обычного» потока;
- CreateFiber() – создает «фибер» из «главного фибера»;
- SwitchToFiber() – выполняет переключение между «фиберами»;
- GetFiberData() – позволяет «фиберу» получить свои данные;
- GetCurrentFiber() – позволяет «фиберу» получить свой дескриптор;
- DeleteFiber() – позволяет одному «фиберу» удалять другие.

Правилам совместного переключения подчиняются только «фиберы» одного процесса. Потоки остальных процессов конкурируют с ними за процессорное время, как с обычными потоками.

2. Межзадачное взаимодействие

При передаче данных между задачами важную роль играют универсальные функции передачи данных, которые работают не только с файлами:

- CreateFile() – создает или открывает уже созданный объект файлового типа, возвращая дескриптор доступа;
- ReadFile() – используя дескриптор доступа, принимает переданные данные в буфер памяти;
- WriteFile() – используя дескриптор доступа, передает данные из буфера памяти;
- CloseHandle() – закрывает объект файлового типа, уничтожая дескриптор.

2.1. Обычные файлы

Файл (file) – именованный набор данных, расположенный на внешнем носителе. (Примечание. Именованные наборы данных, расположенные в памяти – каналы, почтовые ячейки и пр., – формально являясь файлами, носят другие названия). Для доступа к файлам используются следующие функции:

- CreateFile() – создает или открывает уже существующий файл, возвращая дескриптор;
- CloseHandle() – закрывает файл (или внешнее устройство);
- ReadFile() и WriteFile() – позволяют читать и записывать данные в файл;
- SetFilwPointer() – перемещает указатель чтения-записи в файле;
- LockFile() – блокирует доступ к файлу или его региону;
- UnlockFile() – снимает блокировку с файла или его региона.

Для совместимости с ранним версиями Windows сохраняется возможность использования функций:

- _lcreat() или _lopen() – позволяют создать или открыть существующий файл, возвращая дескриптор;
- _lread() и _lwrite() – позволяют читать или записывать файл;
- _llseek() – перемещает указатель чтения-записи в файле;
- _lclose() – закрывает файл.

В Windows реализован так же режим проецирования (mapping) файлов в память, который позволяет обращаться к данным файла не при помощи функций ReadFile() и WriteFile(), а при помощи обычных операций с указателями. Например, если файл спроецирован на массив, то возможно обращение к его отдельным элементам по индексу. (Примечание. Никакого

ускорения доступа к таким файлам нет, т.к. физически данные по-прежнему остаются на диске). Для работы с проецируемыми файлами используются следующие функции:

- CreateFileMapping() – создает файл, проецируемый в память;
- MapViewOfFile() – выполняет проецирование файла в память;
- UnmapViewOfFile() – отменяет проецирования файла в память.

2.2. Каналы

Канал (pipe) – разделяемый буфер с последовательным доступом к элементам данных, использующий дисциплину «очередь». Механизм каналов двунаправленный. При создании или открытии канала в программу возвращаются два дескриптора – один для записи, другой для чтения. Для обмена данными через эти дескрипторы используются функции ReadFile() и WriteFile(). Кроме того, консольный ввод-вывод во многих языках программирования использует механизм каналов, что позволяет, переназначив стандартные дескрипторы ввода-вывода, обмениваться данными при помощи функций printf()/scanf() в Си, потоков «cin» и «cout» в Си++, процедур Read() и Write() в Паскале и т.п.

Для создания анонимного (неименованного) канала используется функция CreatePipe(). Чтобы два разных процесса могли использовать дескрипторы одного и того же неименованного канала, требуется настройка параметров безопасности, разрешающая наследование дескрипторов (т.е. передача из родительского процесса дочернему).

Канал с именем типа «\\.\pipe\имя_канала» создается при помощи функции CreateNamedPipe(). Зная это имя, иные процессы могут открыть его, используя функцию CreateFile(). Функция ConnectNamedPipe() позволяет первому процессу, открывшему канал на чтение (серверу), ждать, пока к нему не обратится другой процесс, записывающий данные (клиент).

2.3. Почтовые ячейки

Почтовая ячейка (mailslot) – разделяемый буфер с произвольным доступом к элементам данных. Механизм почтовых ячеек однонаправленный. Процессом-сервером почтовой ячейки используется функция CreateMailslot(), которая создает новую почтовую ячейку с именем типа «\\.\mailslot\имя_ячейки». Сервер может только читать данные из ячейки при помощи ReadFile(). Процессами-клиентами используется функция CreateFile(), которая позволяет открыть почтовую ячейку как обычный файл. Клиент записывает данные в ячейку при помощи WriteFile().

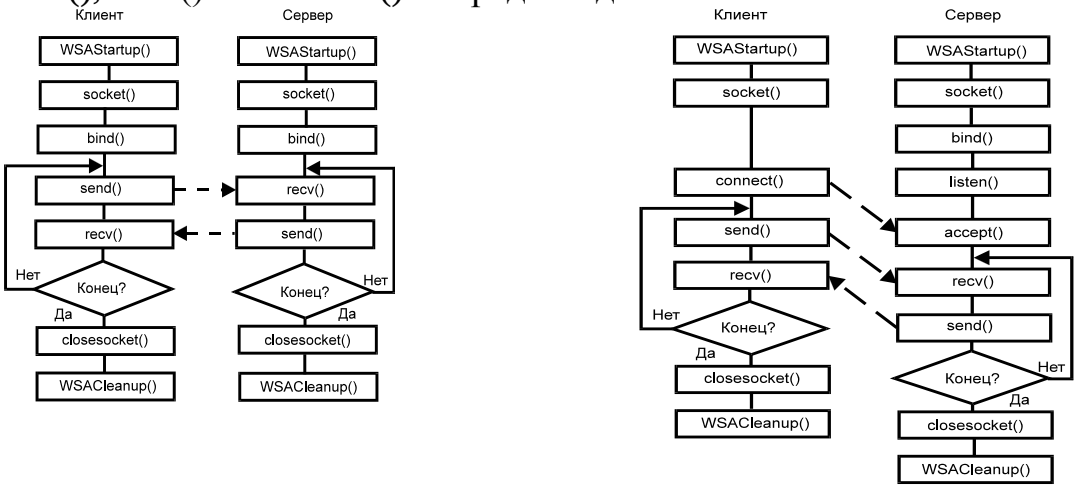
2.4. Сокеты

«Сокет» (socket) – это универсальный интерфейс для обмена данными, использующий сетевые протоколы. Один из участников информационного обмена обязательно должен быть сервером, обслуживающим запросы, а остальные – клиентами. Общая структура алгоритма взаимодействия проиллюстрирована на рис. 1.

Основные функции:

- socket() - создает объект типа «сокет»;

- closesocket() - уничтожает объект типа «сокет»;
- bind() - привязывает сокет к порту и адресу;
- listen() - «прослушивает эфир», ожидая запроса;
- accept() - устанавливает связь со стороны сервера;
- read(), recv() или recvfrom() – читают данные;
- connect() – устанавливает соединение с сервером;
- write(), send() или sendto() - передают данные.



а) По протоколу UDP б) По протоколу TCP

Рис.1. Варианты взаимодействия сервера и клиента

3. Синхронизация процессов и потоков

При синхронизации процессов и потоков важную роль играют универсальные функции ожидания, которые используются совместно с критическими секциями, мьютексами, семафорами и событиями:

- WaitForSingleObject() – приостанавливает процесс иди поток до момента перехода некотрого объекта в «сигнальное» состояние (т.е. в состояние «путь открыт»);
- WaitForMultipleObjects() – приостанавливает процесс или поток до момента перехода в сигнальное состояние одного из объектов, указанных в списке.

Кроме того, эти функции можно использовать для ожидания родительским процессом завершения дочернего.

3.1. Объект «критическая секция»

Объект «критическая секция» (critical section) позволяет синхронизировать только разные потоки одного процесса. (Примечание. В теории операционных систем «критическая секция» - фрагмент программного кода, в котором возможен доступ к «критическому ресурсу», т.е. ресурсу, к которому запрещен одновременный доступ со стороны нескольких задач. Поэтому задачи должны своевременно запрещать или разрешать друг другу вход в «критическую секцию»). В Windows процесс, занявший «критическую секцию», может занять ее повторно, но после этого обязан освободить «критическую секцию» столько раз, сколько занял. После завершения процесса «критическая секция» автоматически уничтожается. Для работы с «критической секцией» используются следующие функции:

- InitializeCriticalSection() – инициализирует новый объект;
- EnterCriticalSection() – входит в «критическую секцию» и блокирует ее для других потоков, при невозможности - ожидает;
- TryEnterCriticalSection() - – входит в «критическую секцию» и блокирует ее для других потоков, при невозможности возвращает признак занятости;
- LeaveCriticalSection() – освобождает блокировку «критической секции»;
- DeleteCriticalSection() – удаляет объект.

Дескриптор созданной критической секции представляет собой структуру служебных данных, имеющую следующий формат:

```
typedef struct _RTL_CRITICAL_SECTION {
 PRTL_CRITICAL_SECTION_DEBUG DebugInfo; // Служебное поле
 LONG LockCount;        // Общий счетчик захватов
 LONG RecursionCount;    // Счетчик захватов потоком-владельцем
 HANDLE OwningThread;    // Уникальный ID потока-владельца
 HANDLE LockSemaphore;   // Объект ядра, используемый для ожидания
 ULONG_PTR SpinCount;    // Количество циклов перед вызовом ядра
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

3.2. Мьютексы

Мьютексы (mutex) – двоичные (то есть, имеющее два состояния: «открыто» и «закрыто») семафоры, которые позволяют управлять доступом в «критическую секцию» для потоков как одного, так и различных процессов. Доступ другим потокам в «критическую секцию» может открывать только тот поток, который его закрыл. Возможен повторный вход потока в захваченную им «критическую секцию». Мьютексами используются следующие функции:

- CreateMutex() – создает новый мьютекс;
- ReleaseMutex() – открывает другим потокам доступ в «критическую секцию»;
- OpenMutex() – позволяет другому процессу открыть существующий мьютекс по имени.

Для организации ожидания доступа используются универсальные функции WaitForSingleObject() и WaitForMultipleObjects().

3.3. Семафоры

«Семафоры» (semaphore) – объекты синхронизации, позволяющие управлять доступом к произвольному ресурсу (например, к «критической секции», к буферу данных). Семафор содержит счетчик разрешенного количества обращений, 0 и отрицательные значения соответствуют состоянию «заблокировано». При «захвате» потоком ресурса счетчик уменьшается, при «отпускании» увеличивается. Поток, «захвативший» ресурс, может многократно обращаться к нему без блокировки. Семафорами используются следующие функции:

- CreateSemaphore() – создает новый семафор;
- OpenSemaphore() – позволяет другому процессу открыть существующий семафо по имени;
- ReleaseSemaphore() – «отпускает» ресурс.

Для организации ожидания доступа используются универсальные функции WaitForSingleObject() и WaitForMultipleObjects().

3.4. События

«Событие» (event) – объект, аналогичный переменной, сигнализирующей другим потокам о каком либо состоянии текущего потока (например, о том, что он захватил «критическую секцию»). Для организации синхронизации при помощи «события» используются следующие функции:

- CreateEvent() – создает новое событие, возвращая его идентификатор;
- SetEvent() – возбуждает событие, чем освобождает ожидающий поток;
- OpenEvent() – позволяет другому потоку использовать созданный объект;
- ResentEvent() – сбрасывает событие;
- PulseEvent() – освобождает все потоки, ожидающие данное событие.

Для организации ожидания доступа используются универсальные функции WaitForSingleObject() и WaitForMultipleObjects().

Литература

1. Побегайло А.П. Системное программирование в Windows. – СПб.:БХВ-Петербург, 2006. – 1056 с.
2. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. – СПб.: Питер, 2008. – 720 с.
3. Румянцев П.В. Азбука программирования в Win32 API. – М.: Горячая линия -Телеком, 1999. – 272 с.
4. Румянцев П.В. Работа с файлами в Win32 API. – М.: Горячая линия - Телеком , 2000. – 200 с.
5. Финогенов К.Г. Win32. Основы программирования. – М.: ДИАЛОГ-МИФИ, 2006. – 416 с.
6. Фролов А., Фролов Г. Программирование для Windows NT. Часть 2. – М.: Диалог-МИФИ, 1996. – 272 с.
7. Харт Дж. Системное программирование в среде Win32. – М.: Издательский дом «Вильямс», 2001. – 464 с.
8. Щупак Ю.А. Win32 API. Эффективная разработка приложений. – СПб.: Питер, 2007. – 572 с.

Приложение А.
Описание используемых функций и типов Win32/64 API

Функция CreateProcess() создаёт новый процесс и его главный поток. Новый процесс выполняет программу, расположенную в указанном файле.

Прототип:

```
BOOL CreateProcess (
 LPCTSTR lpApplicationName,                  // указатель на имя исполняемого файла
 LPTSTR lpCommandLine,                       // указатель на командную строку
 LPSECURITY_ATTRIBUTES lpProcessAttributes,  // указатель на атрибуты безопасности процесса
 LPSECURITY_ATTRIBUTES lpThreadAttributes,   // указатель на атрибуты безопасности потока
 BOOL bInheritHandles,                       // указатель на флаг наследования
 DWORD dwCreationFlags,                      // флаги создания
 LPVOID lpEnvironment,                       // указатель на новый блок среды
 LPCTSTR lpCurrentDirectory,                 // указатель на имя текущего каталога
 LPSTARTUPINFO lpStartupInfo,                // указатель на структуру STARTUPINFO
 LPPROCESS_INFORMATION lpProcessInformation  // указатель на  PROCESS_INFORMATION
 );
```

Параметры:

LpApplicationName – указатель на строку, содержащую имя программы.

LpCommandLine – указатель на строку, содержащую параметры запуска.

LpProcessAttributes и LpThreadAttributes – указатели на структуру SECURITY_ATTRIBUTES, которая определяет, будет ли возвращённый дескриптор на процесс наследуемым для дочерних процессов (если NULL, то не будет).

BInheritHandles – указывает на то, будет ли процесс наследовать дескрипторы вызывающего его процесса. Если TRUE, то каждый открытый дескриптор вызывающего процесса будет наследоваться новым процессом. Наследуемые дескрипторы имеют те же значения и права доступа, что и оригинальные.

DwCreationFlags – указывает дополнительные флаги, контролирующие класс приоритета и создания процесса, из числа следующих (см. табл. 1).

Таблица 1 - Флаги запуска

Значение	Описание
CREATE_DEFAULT_ERROR_MODE	Новый процесс не наследует режим ошибки вызывающего процесса.
CREATE_NEW_CONSOLE	Новый процесс имеет новую консоль, в противоположность наследуемой родительской консоли.
CREATE_NEW_PROCESS_GROUP	Новый процесс является корневым для новой группы процессов.
CREATE_SEPARATE_WOW_VDM	Только для Windows NT: Этот флаг устанавливается только при старте 16-битных Windows приложений.
CREATE_SHARED_WOW_VDM	Только для Windows NT: Этот флаг устанавливается только при старте 16-битных Windows приложений.
CREATE_SUSPENDED	Основной поток нового процесса создаётся в спящем режиме, и не запускается до тех пор, пока не вызвана функция ResumeTread.
CREATE_UNICODE_ ENVIRONMENT	Если установлен, то блок среды указанный параметром lpEnvironment использует символы Unicode. Если опущен, то блок среды использует ASCII.
DEBUG_PROCESS	Если установлен, вызывающий процесс становится отладчиком, а новый поток становится отлаживаемым.
DEBUG_ONLY_THIS_PROCESS	Если не установлен и вызывающий процесс является отлаживаемым, новый процесс будет ещё одним отлаживаемым процессом отладчиком которого будет являться отладчик вызывающего процесса.

DETACHED_PROCESS	Для консольных процессов, новый процесс не имеет доступа к консоли родительского процесса.

Параметр dwCreationFlags также контролирует класс приоритета процесса. По умолчанию это NORMAL_PRIORITY_CLASS, а все его дочерние процессы будут иметь IDLE_PRIORITY_CLASS. Может указываться один из следующих флагов (см. табл. 2)

Таблица 2 - Приоритеты запуска процесса

Приоритет	Описание
HIGH_PRIORITY_CLASS	Указывает, что процесс выполняет критичные ко времени задачи.
IDLE_PRIORITY_CLASS	Указывает, что потоки процесса работают только тогда, когда система не занята и прерываются потоками с более высоким классом приоритета. Примером такого процесса является хранитель экрана.
NORMAL_PRIORITY_CLASS	Указывает на нормальный процесс без специальных требований к управлению задачами.
REALTIME_PRIORITY_CLASS	Указывает, что процесс имеет максимально допустимый приоритет. Потоки с классом приоритета реального времени превалируют над потоками любых других классов приоритета, включая процессы операционной системы, выполняющие важные задачи.

LpEnvironment - указывает на блок среды. Если NULL, то новый процесс использует среду вызывающего процесса.

LpCurrentDirectory - указывает на строку, которая содержит текущее устройство и каталог для дочернего процесса.

LpStartupInfo - указатель на структуру STARTUPINFO, которая указывает, как должно появляться главное окно нового процесса. Можно при помощи GetStartupInfo() получить режимы процесса-родителя и передать их в запускаемый процесс без изменения.

LpProcessInformation - указатель на структуру PROCESS_INFORMATION, в которую записывается идентификационная информация нового процесса:

```
Typedef struct PROCESS_INFORMATION {
  HANDLE hProcess;          // Дескриптор процесса
  HANDLE hTread;            // Дескриптор потока
  DWORD dwProcessId;        // Идентификатор процесса
  DWORD dwThreadId;         // Идентификатор потока
}
```

Возвращаемые значения: при ошибке NULL.

Функция WinExec() создает новый процесс и его главный поток.

Прототип:

```
UINT WinExec (
   LPCSTR lpCmdLine,        // строка имени файла
   UINT uCmdShow            // стиль окна
  );
```

Параметры:

lpCmdLine – строка имени файла запускаемой программы. Файл должен быть размещен в текущем каталоге, или в каталоге операционной системы (например, C:\Windows), или в системном каталоге операционной системы (например, C:\Windows\System), или в одном из каталогов, размещенных в «тропе» запуска.

uCmdShow – стиль окна запускаемой программы (SW_HIDE – скрытое окно, SW_MAXIMIZE и SW_MINIMIZE – максимально и минимально возможные

размеры окна, SW_SHOWDEFAULT и SW_SHOWNORNAL – размер окна по умолчанию).

Возвращаемые значения: 0 – успех, ERROR_BAD_FORMAT – попытка запустить «не-программу»; ERROR_FILE_NOT_FOUND – файл программы не найден; ERROR_PATH_NOT_FOUND – путь не найден.

Функция ExitProcess() завершает выполнение текущего процесса.

Прототип:

```
ExitProcess(
 UINT uExitCode  // Код завершения
);
```

Параметр uExitCode – целое число, возвращаемое родительскому процессу.

Функция TerminateProcess() завершает выполнение процесса извне.

Прототип:

```
TerminateProcess(
   HANDLE hProcess,        // Дескриптор процесса
   UINT uExitCode          // Код завершения
   );
```

Параметры:

hProcess – дескриптор, полученный при запуске процесса.

uExitCode – целое число, возвращаемое родительскому процессу.

Функция CreateThread() создает новый поток в адресном пространстве процесса и возвращает описатель порожденного потока.

Прототип:

```
HANDLE CreateThread (
 LPSECURITY_ATTRIBUTES lpThreadAttributes,     // атрибуты защиты потока
 DWORD dwStackSize,                            // размер стека в байтах
 LPTHREAD_START_ROUTINE lpStartAddress,        // указатель на функцию потока
 LPVOID lpParameter,                           // передаваемый параметр
 DWORD dwCreationFlags,                        // флаги
 LPDWORD lpThreadId                            // указатель на возвращаемый идентификатор
);
```

Параметры:

lpThreadAttributes - указатель на структуру, описывающую параметры защиты потока. Если NULL, то устанавливаются атрибуты «по умолчанию».

dwStackSize - устанавливает размер стека. Если 0, то устанавливается стек, равный стеку «главного» потока.

lpStartAddress - адрес функции потока. Функция имеет один 32-битный аргумент и возвращает 32 битное значение.

lpParameter - параметр, передаваемый в функцию потока.

dwCreationFlags – флаги: 0 – немедленный запуск; CREATE_SUSPENDED - поток не запускается до вызова функции ResumeThread().

lpThreadId - указатель на 32-битную переменную, которой будет присвоено значение дескриптора.

Таблица 3 - Относительные приоритеты потоков

Идентификатор	Уровень приоритета потока
THREAD_PRIORITY_LOWEST	На 2 ниже уровня класса
THREAD_PRIORITY_BELOW_NORMAL	На 1 ниже уровня класса
THREAD_PRIORITY_NORMAL	Равен уровню класса
THREAD_PRIORITY_ABOVE_NORMAL	На 1 выше уровня класса
THREAD_PRIORITY_HIGHEST	На 2 выше уровня класса
THREAD_PRIORITY_IDLE	Равен 1 для процессов класса IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, и HIGH_PRIORITY_CLASS и равен 16 для REALTIME_PRIORITY_CLASS
THREAD_PRIORITY_TIME_CRITICAL	Равен 15 для процессов класса IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, и HIGH_PRIORITY_CLASS, и равен 31 для REALTIME_PRIORITY_CLASS

Возвращаемые значения: дескриптор созданного потока; INVALID_HANDLE_VALUE в случае ошибки.

Функция SuspendThread() приостанавливает работу указанного потока и увеличивает счетчик количества приостановок. Приостанавливаемый поток должен иметь флаг доступа THREAD_SUSPEND_RESUME. Количество приостановок не должно превышать MAXIMUM_SUSPEND_COUNT.

Прототип:

```
DWORD SuspendThread(
   HANDLE hThread            // Дескриптор приостанавливаемого потока
  );
```

Параметры:

HThread - дескриптор приостанавливаемого потока.

Возвращаемые значения: текущее значение счетчика приостановок или 0xFFFFFFFF=-1 в случае неудачи.

Функция ResumeThread() уменьшает счетчик количества приостановок и запускает поток в случае достижения 0.

Прототип:

```
DWORD ResumeThread(
   HANDLE hThread            // Дескриптор ране приостановленного потока
  );
```

Параметр:

HThread - дескриптор потока.

Возвращаемые значения: текущее значение счетчика приостановок или 0xFFFFFFFF=-1 в случае неудачи.

Функция ExitThread() завершает выполнение текущего потока.

Прототип:

```
ExitThread(
   DWORD dwExitCode // Код завершения
   );
```

Параметр:

dwExitCode – возвращаемое потоком целое число.

Функция TerminateThread() завершает выполнение потока извне.

Прототип:

```
TerminateThread(
  HANDLE hThread,  // Дескриптор потока
  DWORD dwExitCode // Код завершения
  );
```

Параметры:

hThread – дескриптор, полученный при запуске потока.

dwExitCode – возвращаемое потоком целое число.

Функция ConvertThreadToFiber() инициализирует «фибер», преобразуя в него поток, обратившийся к этой функции. Обычно служит для создания «главного фибера».

Прототип:

```
PVOID ConvertThreadToFiber (
PVOID lpParam // Параметры «фибера»
);
```

Параметры:

lpParam – блок данных, передаваемых «фиберу» в качестве параметров.

Возвращаемые значения: дескриптор созданного «фибера» или NULL в случае ошибки.

Функция CreateFiber() позволяет создавать «фиберы» из обычных потоков.

Прототип:

```
PVOID CreateFiber(
  DWORD dwStackSize,                    // Размер стека
  PFIBER_START_ROUTINE lpStartAddress,  // Стартовый адрес «фибера»
  PVOID lpParam                         // Параметры «фибера»
);
```

Параметры:

dwStackSize – желаемый размер стека (если 0, то размер составит 1Мб).

LpStartAddress – стартовый адрес процедуры, из которой создается «фибер».

LpParam – параметры, передаваемые в процедуру.

Возвращаемые значения: дескриптор созданного «фибера» или NULL в случае ошибки.

Функция SwitchToFiber() производит переключение на «фибер» с указанным дескриптором в ту точку, где выполнение этого фибера было прервано в последний раз.

Прототип:

```
VOID SwitchToFiber(
  LPVOID lpFiber // Адрес «фибера», которому передается управление
);
```

Параметр:

lpFiber – адрес того фибера, на который происходит переключение.

Функция GetFiberData() позволяет «фиберу» получить доступ к собственным параметрам .

Прототип:

```
LPVOID GetFiberData(VOID);
```

Возвращаемые значения: указатель на блок параметров.

Функция GetCurrentFiber() позволяет «фиберу» получить доступ к собственному адресу.

Прототип:

```
LPVOID GetCurrentFiber(VOID);
```

Возвращаемые значения: адрес функции, которая служит «фибером».

Функция DeleteFiber() позволяет одному «фиберу» удалить «другой».

Прототип:

```
VOID DeleteFiber(LPVOID lpFiber);
```

Параметр:

lpFiber – дескриптор удаляемого «фибера».

Функция CreateFile() создает или открывает объекты и возвращает дескриптор для получения доступа к ним: к файлам, к каналам, к почтовым ячейкам, к коммуникационным ресурсам и т.п.

Прототип:

```
HANDLE CreateFile (
  LPCTSTR lpFileName,                          // Строка имени
  DWORD dwDesiredAccess,                       // Режим доступа
  DWORD dwShareMode,                           // Режим разделения доступа
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,  // Атрибуты безопасности
  DWORD dwCreationDistribution,                // Способ открытия
  DWORD dwFlagsAndAttributes,                  // Атрибуты файла
  HANDLE hTemplateFile                         // Расширенные признаки
 );
```

Параметры:

lpFileName – строка имени объекта. Если lpFileName - путь, то максимум символов в строке определен константой MAX_PATH.

dwDesiredAccess - определяет тип доступа к объекту:

- 0 - флаг доступа к объекту, позволяет он определить атрибуты файла, не подсоединяясь к нему;
- GENERIC_READ и GENERIC_WRITE – разрешают чтение и запись, соответственно.

dwShareMode – флаги совместного доступа:

- 0 – совместный доступ запрещен;
- FILE_SHARE_DELETE – (Windows NT) доступ открыт только хозяину;
- FILE_SHARE_READ – совместный доступ для чтения;
- FILE_SHARE_WRITE – совместный доступ для записи.

lpSecurityAttributes - указатель на структуру TSecurityAttributes с правами доступа к процессам-потомкам; если NULL – то права потомков наследуются у текущего.

dwCreationDistribution - определяет способы создания и открытия:

- CREATE_NEW создает файл. Если файл уже существует, то ошибка.
- CREATE_ALWAYS создает новый файл. Функция переписывает файл, если он существует.

- OPEN_EXISTING открывает файл. Если файл не существует, то ошибка.
- OPEN_ALWAYS открывает файл, если он существует. Если файл не существует, то создает новый.
- TRUNCATE_EXISTING открывает и очищает файл. Если файл не существует, то ошибка.

dwFlagsAndAttributes - определяет признаки и флаги для файла (по умолчанию FILE_ATTRIBUTE_NORMAL).

hTemplateFile - определяет указатель с доступом GENERIC_READ к файлу шаблона, для Windows 95/98 должен быть NULL. Файл шаблона задает признаки файла и расширенные признаки для создаваемого файла.

Возвращаемые значения: дескриптор файла или INVALID_HANDLE_VALUE.

Функция CloseHandle() закрывает ранее открытый дескриптор.

Прототип:

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

Параметр: hFile – дескриптор файла, объекта синхронизации или устройства.

Возвращаемые значения: FALSE при ошибке.

Функция ReadFile() выполняет чтение данных из указанного файла или устройства.

Прототип:

```
BOOL ReadFile(
 HANDLE hFile,                    // дескриптор файла, объекта, устройства
 LPVOID lpBuffer,                 // адрес буфера  для чтения
 DWORD nNumberOfBytesToRead,      // читаемое количество байтов
 LPDWORD lpNumberOfBytesRead,     // реально прочитанное количество байтов
 LPOVERLAPPED lpOverlapped        // адрес структуры описания перекрытия
);
```

hFile - описатель объекта ядра “файл”, полученный в результате вызова функции CreateFile().

LpBuffer - адрес буфера, в который будет производиться чтение.

nNumberOfBytesToRead - количество байт, которые необходимо прочитать.

lpNumberOfBytesRead - адрес переменной, в которой после завершения операции будет размещено количество реально прочитанных байт.

lpOverlapped - указатель на структуру OVERLAPPED, управляющую асинхронным вводом-выводом:

```
typedef struct OVERLAPPED {
 DWORD Internal;              // Статус завершения операции
 DWORD InternalHigh;          // Количество переданных байтов
 DWORD Offset;                // Позиция в файле начала чтения/записи
 DWORD OffsetHigh;            // Количество байтов для передачи
 HANDLE hEvent;               // Описатель события, соответствующего завершению операции
} OVERLAPPED;
```

Если этот параметр равен NULL, то выполняется синхронная операция (т.е. с ожиданием завершения).

Возвращаемые значения: FALSE при ошибке.

Функция WriteFile() выполняет запись данных в указанный файл или устройство. Прототип:

```
BOOL WriteFile (
 HANDLE hFile,                    // дескриптор файла, объекта, устройства
 LPCVOID lpBuffer,                // адрес буфера для чтения
 DWORD nNumberOfBytesToWrite,     // записываемое количество байтов
 LPDWORD lpNumberOfBytesWritten,  // реально записанное количество байтов
 LPOVERLAPPED lpOverlapped        // адрес структуры описания перекрытия
);
```

Параметры – см. аналогичные для функции ReadFile().

Возвращаемые значения: FALSE при ошибке.

Функция SetFilePointer() перемещает в файле позицию чтения-записи. Прототип:

```
DWORD SetFilePointer(
   HANDLE hFile,                // дескриптор файла
   LONG lDistanceToMove,        // дистанция перемещения
   PLONG lpDistanceToMoveHigh,  // адрес старшего слова для дистанции
   DWORD dwMoveMethod           // способ перемещения
  );
```

Параметры:

hFile – дескриптор файла;

lDistanceToMove – дистанция для перемещения указателя;

lpDistanceToMoveHigh – адрес старшего слова для дистанции (в том случае, если размер файла превышает 4.3 Гб).

dwMoveMethod – способ перемещения указателя (FILE_BEGIN – от начала файла; FILE_CURRENT – от текущей позиции; FILE_END – от конца файла).

Возвращаемые значения: -1=0xFFFFFFFF в случае ошибки или дистанцию, на которое выполнено перемещение.

Функция LockFile() захватывает фрагмент файла, запрещая к нему доступ другим процессам.

Прототип:

```
BOOL LockFile(
   HANDLE hFile,                      // Дескриптор файла
   DWORD  dwFileOffsetLow,            // Младшая часть начальной позиции
   DWORD  dwFileOffsetHigh,           // Старшее слово начальной позиции
   DWORD  nNumberOfBytesToLockLow,    // Младшая часть длины
   DWORD  nNumberOfBytesToLockHigh    // Старшая часть длины
  );
```

Параметры:

hFile - дескриптор файла.

dwFileOffsetLow – младшая часть начальной позиции.

dwFileOffsetHigh – старшая часть начальной позиции.

nNumberOfBytesToLockLow – младшая часть длины.

nNumberOfBytesToLockHigh – старшая часть длины.

Возвращаемое значение: признак успеха (TRUE) или неудачи (FALSE).

Функция UnlockFile() освобождает ранее захваченный фрагмент файла.

Прототип:

```
BOOL UnlockFile(
```

```
  HANDLE  hFile,              // Дескриптор файла
  DWORD  dwFileOffsetLow,// Младшая часть начальной позиции
  DWORD  dwFileOffsetHigh,          // Старшее слово начальной позиции
  DWORD  cbUnlockLow,               // Младшая часть длины
  DWORD  cbUnlockHigh               // Старшая часть длины
 );
```

Описание параметров – см. в функции LockFile().

Возвращаемое значение: признак успеха (TRUE) или неудачи (FALSE).

Функция CreateFileMapping() создает из части файла именованный или неименованный проецируемый в память объект.

Параметры:

```
HANDLE CreateFileMapping(
 HANDLE  hFile,                                  // Дескриптор проецируемого файла
 LPSECURITY_ATTRIBUTES  lpFileMappingAttributes,  // Параметры безопасности
 DWORD  flProtect,                               //  Защита проецируемого объекта
 DWORD  dwMaximumSizeHigh,                       //  Старшая часть размера объекта
 DWORD  dwMaximumSizeLow,                        //  Младшая часть размера объекта
 LPCTSTR  lpName                                 //  Имя проецируемого объекта
);
```

Параметры:

hFile - дескриптор проецируемого файла; если 0xFFFFFFFF, то вместо дискового файла используется безымянный буфер в памяти;

lpFileMappingAttributes - параметры безопасности (если NULL – то по умолчанию).

flProtect - защита проецируемого объекта: PAGE_READONLY – только чтение; PAGE_READWRITE - чтение/запись.

dwMaximumSizeHigh - старшая часть размера объекта.

dwMaximumSizeLow - младшая часть размера объекта.

lpName – имя проецируемого объекта или NULL.

Возвращаемое значение: идентификатор созданного объекта или NULL (в случае ошибки).

Функция MapViewOfFile() отображает объект (например, файл) на адресное пространство текущего процесса. После этого доступ к объекту становится возможен по указателю, например, как к массиву.

Прототип:

```
LPVOID MapViewOfFile(
  HANDLE hFileMappingObject,        // Идентификатор отображаемого объекта
  DWORD  dwDesiredAccess,                 // Режим доступа
  DWORD  dwFileOffsetHigh,                // Старшие 32 бита файлового смещения
  DWORD  dwFileOffsetLow,           // Младшие 32 бита файлового смещения
  DWORD  dwNumberOfBytesToMap  // Размер отображаемой области
 );
```

Параметры:

hFileMappingObject - идентификатор отображаемого объекта;

dwDesiredAccess - режим доступа: FILE_MAP_WRITE или FILE_MAP_ALL_ACCESS – чтение/запись; FILE_MAP_READ - только чтение;

dwFileOffsetHigh - старшие 32 бита файлового смещения;

dwFileOffsetLow - младшие 32 бита файлового смещения;

dwNumberOfBytesToMap - размер отображаемой области.

Возвращаемое значение: адрес начала отображаемой области или NULL (в случае неудачи).

Функция UnmapViewOfFile() отменяет проецирование объекта на память.

Прототип:

```
BOOL UnmapViewOfFile(
  LPVOID  lpBaseAddress             // Адрес начала отображаемой области
 );
```

Параметры:

lpBaseAddress - значение, полученное в результате выполнения функции MapViewOfFile.

Возвращаемое значение: TRUE при успехе, FALSE при ошибке.

Функция WaitForSingleObject() ожидает перехода объекта синхронизации в сигнальное состояние. Так же возможно ожидать завершение процесса.

Прототип:

```
DWORD WaitForSingleObject (
 HANDLE hHandle;                  // дескриптор объекта
 DWORD dwMilliseconds;            // тайм-аут в миллисекундах
);
```

Параметры:

hHandle – идентификатор объекта синхронизации;

dwMilliseconds – тайм-аут в миллисекундах, если INFINITE, то бесконечно; если 0, то проверяется состояние объекта, но ожидание не выполняется.

Возвращаемое значение: см. табл. 4.

Таблица Б.4 - Коды завершения

Код	Описание
WAIT_ABANDONED	Поток, владевший объектом, завершился, не переведя объект в сигнальное состояние.
WAIT_OBJECT_0	Объект перешел в сигнальное состояние
WAIT_TIMEOUT	Истек срок ожидания. Обычно в этом случае генерируется ошибка, либо функция вызывается в цикле до получения другого результата
WAIT_FAILED	Произошла ошибка, например неверное значение hHandle

Функция WaitForMultipleObjects() – ожидает перевода в сигнальное состояние группы объектов.

Прототип:

```
DWORD WaitForMultipleObjects (
 DWORD nCount,       // число объектов в массиве lpHandles
 HANDLE *lpHandles,  // указатель на массив дескрипторов объектов
 BOOL bWaitAll,      // ждать всех сразу или хотя бы одного?
 DWORD dwMilliseconds // тайм-аут в миллисекундах
);
```

Параметры:

nCount – количество элементов в массиве lpHandles.

lpHandles - массив дескрипторов объектов.

bWaitAll - TRUE - ждать всех сразу; FALSE - ждать хотя бы одного.

dwMilliseconds - тайм-аут ожидания в миллисекундах; если INFINITE, то бесконечно.

Возвращаемое значение: см. табл. 5.

Таблица 5 - Коды завершения

Код	Описание
Число в диапазоне от WAIT_OBJECT_0 до WAIT_OBJECT_0 + nCount – 1	Если bWaitAll равно TRUE, то это число означает, что все объекты перешли в сигнальное состояние. Если FALSE – то, вычтя из возвращенного значения WAIT_OBJECT_0, мы получим индекс объекта в массиве lpHandles.
Число в диапазоне от WAIT_ABANDONED_0 до WAIT_ABANDONED_0 + nCount – 1	Если bWaitAll равно TRUE – это означает, что все перешли в сигнальное состояние, но хотя бы один из владевших ими потоков завершился, не сделав объект сигнальным. Если FALSE – то, вычтя из возвращенного значения WAIT_ABANDONED_0, мы получим индекс объекта в массиве lpHandles, поток, владевший которым, завершился, не сделав его сигнальным.
WAIT_TIMEOUT	Истек период ожидания
WAIT_FAILED	Произошла ошибка

Функция CreateEvent() создает новый объект синхронизации типа «событие».

Прототип:

```
HANDLE CreateEvent (
 LPSECURITY_ATTRIBUTES lpEventAttributes,        // атрибут защиты
 BOOL bManualReset,                              // тип сброса TRUE - ручной
 BOOL bInitialState,                             // начальное состояние TRUE - сигнальное
 LPCTSTR lpName                                  // имя обьекта
);
```

Параметры:

lpEventAttributes - адрес структуры SECURITYATTRIBUTES (только для Windows NT).

bManualReset - задает, будет событие переключаемым вручную (TRUE) или автоматически (FALSE).

bInitialState - задает начальное состояние, если TRUE - объект в сигнальном состоянии.

lpName - имя объекта (может быть NULL, если объект неименованный).

Возвращаемое значение: идентификатор созданного объекта или 0 в случае ошибки.

Функция OpenEvent()открывает уже существующей, ранее созданный (например, в другом процессе) объект типа «событие».

Прототип:

```
HANDLE OpenEvent(
   DWORD dwDesiredAccess,          // Права доступа к объекту
   BOOL bInheritHandle,      // Флаг наследования
   LPCTSTR lpName                  // Имя объекта
  );
```

Параметры:

dwDesiredAccess - задает права доступа к объекту и может принимать одно из следующих значений (см. табл. 6).

Таблица 6 - права доступа к событию

Значение	Описание
EVENT_ALL_ACCESS	Приложение получает полный доступ к объекту
EVENT_MODIFY_STATE	Приложение может изменять состояние объекта функциями SetEvent и ResetEvent
SYNCHRONIZE	Только для Windows NT – приложение может использовать объект только в функциях ожидания

BInheritHandle - Задает, может ли объект наследоваться дочерними процессами.

LpName - Имя объекта (может быть пустое, если объект неименованный и используется для синхронизации разных потоков внутри одного процесса).

Возвращаемое значение: идентификатор открытого объекта, либо 0 в случае ошибки.

Функция SetEvent() устанавливает объект в сигнальное состояние, то есть, снимает блокирование ожидающих потоков.

Прототип:

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

Параметры:

hEvent - идентификатор объекта.

Возвращаемое значение: выполнено успешно (TRUE) или ошибка (FALSE).

Функция ResetEvent() сбрасывает объект, устанавливая его в несигнальное состояние.

Прототип:

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

hEvent - идентификатор объекта.

Возвращаемое значение: выполнено успешно (TRUE) или ошибка (FALSE).

Функция PulseEvent() устанавливает объект в сигнальное состояние, дает отработать всем функциям ожидания, ожидающим этот объект, а затем снова сбрасывает его.

Прототип:

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

Параметры:

hEvent - идентификатор объекта.

Возвращаемое значение: выполнено успешно (TRUE) или ошибка (FALSE).

Функция CreateMutex() – создает новый мьютекс.

Прототип:

```
HANDLE CreateMutex (
  LPSECURITY_ATTRIBUTES lpMutexAttributes,        // атрибут безопастности
  BOOL bInitialOwner,                             // флаг начального владельца
  LPCTSTR lpName                                  // имя объекта
);
```

Параметры:

lpMutexAttributes – адрес структуры TsecurityAttributes.

bInitialOwner – задает, будет ли процесс владеть мьютексом сразу после создания. Если неизвестно, существует ли уже мьютекс с таким именем, программа не должна запрашивать владение объектом при создании (т.е. должна передать в качестве bInitialOwner значение FALSE).

lpName – имя объекта.

Возвращаемое значение: дескриптор созданного объекта, либо 0.

Функция OpenMutex() – открывает существующий, ранее созданный (например, в другом процессе) мьютекс.

Прототип:

```
HANDLE OpenMutex(
   DWORD dwDesiredAccess,        // Параметры доступа
   BOOL bInheritHandle,          // Права наследования
   LPCTSTR lpName                // Имя объекта
  );
```

Параметры:

dwDesiredAccess - описывает права доступа и может принимать одно из значений (см. Табл. Б.7).

Таблица 7 - Права доступа к мьютексу

Значение	Описание
MUTEX_ALL_ACCESS	Приложение получает полный доступ к объекту
SYNCHRONIZE	Только для Windows NT – приложение может использовать объект только в функциях ожидания и функции ReleaseMutex

bInheritHandle – если TRUE, то разрешена передача прав доступа к мьютексу дочерним процессам ;

lpName – строка имени мьютекса.

Возвращаемое значение: идентификатор открытого мьютекса, либо 0 в случае ошибки.

Функция ReleaseMutex() освобождает доступ к объекту, доступ к которому огражден при помощи мьютекса.

Прототип:

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

Параметр:

hMutex - дескриптор мьютекса.

Возвращаемое значение: выполнено успешно (TRUE) или ошибка (FALSE).

Функция CreateSemaphore() создает новый семафор.

Прототип:

```
HANDLE CreateSemaphore (
        LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,    // атрибут доступа
        LONG lInitialCount,                             // начальное состояние счетчика
        LONG lMaximumCount,                             // максимальное количество обращений
        LPCTSTR lpName                                  // имя объекта
);
```

Параметры:

lMaximumCount - задает максимальное значение счетчика семафора.

lInitialCount - задает начальное значение счетчика и должен быть в диапазоне от 0 до lMaximumCount.

lpName - задает имя семафора. Если в системе уже есть семафор с таким именем, то новый не создается, а возвращается идентификатор существующего семафора. В случае если семафор используется внутри одного процесса, можно создать его без имени, передав значение NULL.

Возвращаемое значение: идентификатор созданного семафора, либо 0 при ошибке.

Функция OpenSemaphore() открывает уже существующий, ранее созданный (например, в другом процессе) семафор.

Прототип:

```
HANDLE OpenMutex(
   DWORD dwDesiredAccess,        // права доступа к объекту
   BOOL bInheritHandle,          // права наследования
   LPCTSTR lpName                // Имя объекта
  );
```

Параметры:

dwDesiredAccess - задает права доступа к объекту и может принимать одно значений, см. табл. 8.

Таблица 8 - Права доступа к семафору

Значение	Описание
SEMAPHORE_ALL_ACCESS	Поток получает все права на семафор (только для программ на Си, а для программ на Delphi можно воспользоваться синонимом для событий - EVENT_ALL_ACCESS)
SEMAPHORE_MODIFY_STATE	Поток может увеличивать счетчик семафора функцией ReleaseSemaphore()
SYNCHRONIZE	Только Windows NT – поток может использовать семафор в функциях ожидания

InheritHandle - задает, может ли объект наследоваться дочерними процессами.

LpName – строка имени семафора.

Возвращаемое значение: идентификатор созданного семафора, либо 0, если создать объект не удалось.

Функция ReleaseSemaphore() уменьшает значение счетчика блокировок (а точнее, увеличивает внутреннюю переменную счетчика в интервале от lInitialCount до lMaximumCount), снимая тем самым блокировку с объекта, доступ к которому был огражден при опмощи семафора.

```
BOOL ReleaseSemaphore (
        HANDLE hSemaphore,        // хендл семафора
        LONG lReleaseCount,       // на сколько изменять счетчик
        LPLONG lpPreviousCount    // предыдущее значение
);
```

Параметры:

hSemaphore – дескриптор объекта.

LReleaseCount - число, на которое изменяется значение семафора (по умолчанию на 1).

lpPreviousCount – адрес переменной, сохраняющей предыдущее значение семафора (можно передать пустую ссылку NULL, если это значение не интересует).

Возвращаемое значение: TRUE при успехе. Если значение счетчика после выполнения функции превысит заданный для него функцией CreateSemaphore максимум, то Возвращаемые значения: FALSE, а значение семафора не изменяется.

Функция InitializeCriticalSection() инициализирует новую критическую секцию.

Прототип:

```
VOID InitializeCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection        // указатель на объект
);
```

Параметр: lpCriticalSection – возвращаемый дескриптор критической секции.

Функция EnterCriticalSection() позволяет потоку войти в критическую секцию, заблокировав другим потокам доступ. Если в этот момент ни один из потоков в процессе не владеет объектом, то поток становится владельцем критической секции и продолжает выполнение. Если секция уже захвачена другим потоком, то выполнение потока, вызвавшего функцию, приостанавливается до её освобождения. Поток, владеющий критической секцией, может многократно вызывать функцию EnterCriticalSection() без блокирования своего исполнения.

Прототип:

```
VOID EnterCriticalSection(
 LPCRITICAL_SECTION lpCriticalSection        // указатель на объект
);
```

Параметр: lpCriticalSection – дескриптор критической секции.

Функция TryEnterCriticalSection() - позволяет потоку войти в критическую секцию, заблокировав другим потокам доступ. Если в этот момент ни один из потоков в процессе не владеет объектом, то поток становится владельцем критической секции и продолжает выполнение. Если секция уже захвачена другим потоком, то выполнение продолжается, но возвращается признак FALSE.

Прототип:

```
BOOL TryEnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection        // указатель на объект
);
```

Параметр: lpCriticalSection – дескриптор критической секции.

Возвращаемое значение: признак захвата критической секции другим потоком.

Функция LeaveCriticalSection() освобождает объект независимо от количества предыдущих вызовов потоком функции EnterCriticalSection(). Если имеются другие потоки, ожидающие освобождения секции, один из них становится её владельцем и продолжает исполнение. Если поток завершился, не освободив критическую секцию, её состояние становится неопределенным, что может вызвать сбой работы программы.

Прототип:

```
VOID LeaveCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection        // указатель на объект
);
```

Параметр: lpCriticalSection - дескриптор критической секции.

Функция DeleteCriticalSection() уничтожает критическую секцию.

Прототип:

```
VOID DeleteCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection        // указатель на объект
);
```

Параметр: lpCriticalSection – дескриптор критической секции.

Функция CreatePipe() создает анонимный канал для передачи данных между двумя процессами (потоками). Если такой канал создан, то он может и должен быть единственным в системе. Запись и чтение в такой канал осуществляются при помощи функций **ReadFile()** и **WriteFile()** с соответствующими дескрипторами.

Прототип:

```
BOOL CreatePipe(
  PHANDLE  hReadPipe,                          // Адрес дескриптора для чтения
  PHANDLE  hWritePipe,                         // Адрес дескриптора для записи
  LPSECURITY_ATTRIBUTES  lpPipeAttributes,     // Указатель на атрибуты защиты
  DWORD  nSize                                 // Размер канала в байтах
 );
```

Параметры:

hReadPipe - адрес дескриптора для чтения.

hWritePipe - адрес дескриптора для записи.

lpPipeAttributes - указатель на атрибуты защиты (если NULL, то по умолчанию).

nSize – максимальный размер канала в байтах (если 0, то размер определяется автоматически).

Возвращаемое значение: TRUE при успехе, FALSE при ошибке.

Функция CreateMailSlot() создает почтовый ящик (ячейку) для передачи данных между двумя процессами, причем процессы могут располагаться как на одной машине, так и на разных узлах сети. Формат имени: «\\.\mailslot\[путь]имя».

Прототип:

```
HANDLE CreateMailslot(
  LPCTSTR  lpName,                             // Имя почтового ящика
  DWORD  nMaxMessageSize,                      // Максимальный размер сообщения
  DWORD  lReadTimeout,                         // Тайм-аут перед чтением (в мс)
  LPSECURITY_ATTRIBUTES  lpSecurityAttributes  // Указатель на атрибуты защиты
 );
```

Параметры:

lpName - имя почтового ящика

nMaxMessageSize - максимальный размер сообщения.

lReadTimeout - тайм-аут для чтения (в миллисекундах). Частные случаи: 0 – немедленно вернуть управление, если ящик пуст; MAILSLOT_WAIT_FOREVER – ожидать бесконечно.

lpSecurityAttributes - указатель на атрибуты защиты (если NULL, то по умолчанию).

Возвращаемое значение: TRUE при успехе, FALSE при ошибке.

Функция WSAStartup() инициализирует подсистему работы с сокетами.

Прототип:

```
int WSAStartup (
 WORD wVersionRequested,          // Требуемая версия WinSock
 LPWSADATA  lpWSAData             // Информация о реализации
);
```

Параметры:

wVersionRequested – требуемая версия WinSock (младший байт – версия, старший - ревизия), например 0x202;

lpWSAData – указатель на структуру, содержащую подробности реализации:

```
typedef struct WSAData {
  WORD        wVersion;          // Требуемая версия
  WORD        wHighVersion;      // Максимальная поддерживаемая версия
  char        szDescription[257];// Строка описания системы сокетов
  char        szSystemStatus[257];// Строка описания конфигурации
  unsigned short iMaxSockets;    // Максимальное число открытых сокетов
  unsigned short iMaxUdpDg;      // Максимальный размер UDP-пакета
  char FAR *  lpVendorInfo;      // Указатель на дополнительную информацию
} WSADATA;
```

Возвращаемые значения: 0 в случае успеха, 10091 – сокеты не готовы, 10092 или 10022 – требуемая версия сокетов не поддерживается.

Функция WSACleanup() завершает работу с сокетами.

Прототип:

```
int WSACleanup (void);
```

Возвращаемое значение: 0 в случае успеха, 10093 – сокеты ранее не были проинициализированы, 10050 – ошибка в сетевой подсистеме, 10036 – выполняется блокирующая операция.

Функция WSAGetLastError() Возвращает номер последней сетевой ошибки.

Прототип:

```
int WSAGetLastError (void);
```

Возвращаемое значение: см. Приложение В.

Функция socket() создает новый (неинициализированный) сокет.

Прототип:

```
SOCKET socket (
 int af,              // Тип адрес
 int type,            // Тип сокета
 int protocol         // Протокол
);
```

Параметры:

af – тип используемого адреса (1 – локальный адрес, 2 – адрес в формате TCP/IP, 6 – в формате IPX/SPX, 17 – NetBIOS, 21- Banyan и т.п.);

type – тип создаваемого сокета (1 – потоковый для TCP, 2 – дейтаграммный для UDP, 3 – формируемый вручную);

protocol – тип протокола (0 – соответствует типу сокета).

Возвращаемое значение: дескриптор сокета или 0 – в случае неудачи.

Функция closesocket() закрывает сокет.

Прототип:

```
int closesocket  (
```

```
 SOCKET sock // Сокет
);
```

Параметр: sock – дескриптор открытого ранее сокета.

Возвращаемое значение: 0 в случае успеха или –1 в случае неудачи.

Функция connect() пытается установить соединение с сокетом иного сетевого узла (обычно с потоковым по протоколу TCP).

Прототип:

```
int connect(
 SOCKET sock,                          // Сокет
 const struct socketaddr FAR* name,    // Идентификатор узла
 int namelen                           // Длина идентификатора
);
```

Параметры:

sock – дескриптор открытого ранее сокета;

name – идентификатор узла, с которым устанавливается соединение, заданный в виде:

```
struct sockaddr_in  {
  short   sin_family;          // тип адреса (см. описание функции socket() );
  u_short sin_port;            // номер порта
  struct  in_addr  sin_addr;   // 32-битовый адрес
  char    sin_zero[8];         // для других типов сетей, должны быть 0
};
```

namelen – длина идентификатора (16).

Возвращаемое значение: 0 в случае удачи, иначе -1.

Функция bind() устанавливает соответствие между сокетом и адресом текущего узла.

Прототип:

```
int bind (
 SOCKET sock,                          // Сокет
 const struct sockaddr FAR * addr,     // Идентификатор узла
 int namelen                           // Длина идентификатора
);
```

Параметры:

sock – дескриптор открытого ранее сокета;

name – идентификатор узла, с которым устанавливается соединение (см. описание функции connect());

namelen – длина идентификатора (16).

Возвращаемое значение: 0 в случае удачи, иначе -1.

Функция listen() ожидает запрос соединения от иного сокета.

Прототип:

```
int listen(
 SOCKET sock,     // Сокет
 int backlog      // Максимальное количество сообщений
);
```

Параметры:

sock – дескриптор ранее открытого сокета;

backlog – максимально допустимый размер очереди сообщений.

Возвращаемое значение: 0 в случае успеха, иначе -1.

Функция accept() устанавливает соединение с другим сокетом.

Прототип:

```
SOCKET accept (
 SOCKET sock,                 // Сокет
 struct sockaddr FAR * addr,  // Идентификатор сокета
 int FAR * addrlen            // Длина идентификатора
);
```

Параметры:

sock – дескриптор открытого ранее сокета;

name – идентификатор узла, с которым устанавливается соединение (см. описание функции connect());

namelen – длина идентификатора (16).

Возвращаемое значение: дескриптор сокета в случае успеха, иначе -1.

Функция send() посылает данные через сокет, который (или с которым) уже установлено соединение.

Прототип:

```
int send (
 SOCKET sock,                 // Сокет
 const char FAR* buf,         // Данные
 int bufsize,                 // Размер данных
 int flags                    // Флаги
);
```

Параметры:

sock – дескриптор открытого ранее сокета;

buf - буфер данных;

bufsize – количество передаваемых данных;

flags – битовые флаги (1 – «срочные данные», 4 – передавать без маршрутизации).

Возвращаемое значение: количество передаваемых (не переданных!) байтов или –1 в случае неуспеха.

Функция sendto() посылает данные через сокет другому сокету.

Прототип:

```
int sendto(
 SOCKET s,                          // Сокет
 const char FAR * buf,              // Данные
 int len,                           // Длина данных
 int flags,                         // Битовые флаги
 const struct sockaddr FAR * to,    // Идентификатор иного сокета
 int tolen                          // Длина структуры с идентификатором
);
```

Параметры:

s – дескриптор открытого ранее сокета;

buf – буфер данных;

len – количество передаваемых данных;

flags – битовые флаги;

to – идентификатор узла, куда выполняется передача данных;

tolen – длина идентификатора.

Возвращаемое значение: количество передаваемых (не переданных!) байтов или –1 в случае неуспеха.

Функция recv() считывает данные из сокета.

```
int recv (
 SOCKET sock,              // Сокет
 char FAR * buf,           // Данные
 int bufsize,              // Размер буфера
 int flags                 // Флаги
);
```

Параметры:

sock – дескриптор открытого ранее сокета;

buf – буфер данных;

bufsize – размер буфера;

flags – битовые флаги (1 – «срочные данные», 2 – прочитать данные, не удаляя их из входной очереди).

Возвращаемое значение: количество прочитанных байтов (возможно, 0) или -1 в случае ошибки.

Функция recvfrom() считывает данные из сокета с указанным идентификатором.

Прототип:

```
int recvfrom(
 SOCKET s,                 // Сокет
 char FAR * buf,           // Буфер данных
 int len,                  // Размер буфера
 int flags,                // Флаги
 struct sockaddr FAR * from,  // Идентификатор сокета
 int FAR * fromlen         // Размер структуры идентификатора
);
```

Параметры:

s – дескриптор открытого ранее сокета;

buf – буфер данных;

len – размер буфера;

flags – битовые флаги (1 – «срочные данные», 2 – прочитать данные, не удаляя их из входной очереди);

from – идентификатор узла, откуда выполняется прием данных;

fromlen – длина идентификатора.

Возвращаемое значение: количество прочитанных байтов (возможно, 0) или -1 в случае ошибки.

Функция setsockopt() устанавливает или изменяет режимы работы сокета.

Прототип:

```
int setsockopt (
 SOCKET sock,              // Сокет
 int level,                // Уровень
 int optname,              // Изменяемая опция
 const char FAR * optval,  // Новое значение
 int optlen                // Размер буфера с новым значением
);
```

Параметры:

sock - дескриптор открытого ранее сокета;

level – уровень, на котором сокет определен (0 – уровень пакетов TCP/IP, 0xFFFF – уровень сокетов);

optname – изменяемая опция;

optval – новое значение изменяемой опции.

optlen – длина буфера, содержащего новое значение.

Возвращаемое значение: 0 в случае успеха, иначе –1.

Функция gethostbyname() получает информацию о сетевом узле, заданном при помощи символического имени.

Прототип:

```
struct hostent *gethostbyname(
  const char *name  // Символическое имя узла
);
```

Параметр: name – строка символического имени (например, “pop3.hotmail.com”).

Возвращаемое значение: 0 в случае ошибки, в противном случае заполненную структуру:

```
struct hostent {
    char  *  h_name;            // «Официальное» имя узла
    char  * * h_aliases;        // Массив строк с «альтернативными» именами
    short  h_addrtype;          // Тип адреса
    short   h_length;           // Длина каждого адреса
    char  * * h_addr_list;      // Массив числовых адресов
};
```

Функция gethostbyaddr() получает информацию о сетевом узле, заданном при помощи числового адреса.

Прототип:

```
struct hostent *gethostbyaddr (
  const char * addr,            // Указатель на числовой адрес
  int len,                      // Длина адреса
 int type                       // Тип адреса
);
```

Параметры:

addr – указатель на числовой адрес;

len – длина адреса;

type – тип адреса.

Возвращаемое значение: 0 в случае ошибки или заполненную структуру hostent (см. описание функции gethostbyname()).

Функция gethostname() получает имя текущего узла сети.

Прототип:

```
int gethostname (
  char * name,        // Возвращаемое имя
  int namelen         // Длина имени
 );
```

Параметры:

name – строка с именем;

namelen – длина имени.

Возвращаемое значение: в случае успеха 0, иначе код ошибки.

Группа функций ntohs(), htons(), ntohl() и **htonl()** преобразует порядок байтов в числовых данных к виду, используемому в сетях, и обратно:

- u_short ntohs (u_short netshort) – 16-битовое число из сетевого в «нормальный»;
- u_short htons (u_short hostshort) – 16-битовое число из «нормального» в сетевой;
- u_long ntohl (u_long netlong) – 32-битовое число из сетевого в «нормальный»;
- u_long htonl (u_long hostlong) - 32-битовое число из «нормального» в сетевой.

Функция inet_addr() преобразует строку с IP-адресом вида “AAA.BBB.CCC.DDD” в числовую форму.

Прототип:

```
unsigned long inet_addr (
 const char * cp                  // Строка с адресом
);
```

Параметр: cp – строка с адресом.

Возвращаемое значение: числовой адрес или –1 в случае ошибки.

Функция inet_ntoa() преобразует IP-адрес, представленный в числовой форме, в строку вида «AAA.BBB.CCC.DDD».

Прототип:

```
char FAR * inet_ntoa (
  struct in_addr in    // Числовой адрес
 );
```

Параметр: in – числовой IP-адрес в одном из видов:

```
struct in_addr {
  union {
    struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;          // Четыре байта
    struct { u_short s_w1,s_w2; } S_un_w;                   // Два 16-битовых слова
    u_long S_addr;                                          // 32-битовое число
} S_un;
```

Возвращаемое значение: при ошибке 0.

Приложение Б. Примеры программирования

Пример 1. Демонстрация запуска потоков и синхронизации их при помощи критических секций. Главный поток («поставщик») поочередно передает через глобальную переменную дочернему потоку («потребителю») символы строки, а тот отображает их на экране. При этом соблюдаются неодновременность и попеременность доступа.

```
#include <stdio.h>
#include <windows.h>
CRITICAL_SECTION cs1, cs2, cs3;
char buffer;  // Глобальный буфер
DWORD WINAPI Receiver( void *tmp ) {          // Поток - приемник данных
 while (1) {
  if (cs3.LockCount==-1) {                    // Секция свободна?
   EnterCriticalSection (&cs3);
   EnterCriticalSection (&cs1);               // Жду, чтобы можно было принимать
   printf("%c", buffer);                      // Принять данные
   LeaveCriticalSection (&cs1);
   LeaveCriticalSection (&cs2);               // Принял, можно снова передавать
   }
  }
}
main() {  // Поток - передатчик данных
 DWORD t; char  s[] = {"Hello world!"}; int i = 0;
 InitializeCriticalSection (&cs1);           // Инициализация критической секции
 InitializeCriticalSection (&cs2);           // Инициализация критической секции
 InitializeCriticalSection (&cs3);           // Инициализация критической секции
 EnterCriticalSection (&cs3);                // Читать пока нельзя
 CreateThread(NULL, 0, Receiver, NULL, 0, &t);  // Запустить поток
 while (s[i]) {
  if (cs2.LockCount==-1) {                   // Секция свободна?
   EnterCriticalSection (&cs2);              // Жду, чтобы можно было передавать
   EnterCriticalSection (&cs1);
   buffer = s[i++];           // Передача данных
   LeaveCriticalSection (&cs1); // Передал, можно снова читать
   LeaveCriticalSection (&cs3);
   }
  }
}
```

Пример 2. Демонстрация запуска «фиберов» («волокон») и передачи между ними данных при помощи сокетов. Главный «фибер» запускает два дочерних: сервер и клиент. Клиент передает серверу строки, тот переворачивает их и возвращает обратно, после чего клиент отображает перевернутые строки на экране.

```
#include <stdio.h>
#define _WIN32_WINNT 0x400
#include <windows.h>
#include <winsock2.h>
#include <locale.h>
#define MAXBUF 1024
void *f0, *f1, *f2;

void WINAPI Fiber1(void *q) { // «Фибер» - сервер
 char buf[MAXBUF]; WSADATA d; SOCKET my_sock;
 sockaddr_in local_addr, client_addr;
```

```
 WSAStartup(0x202, &d);
 my_sock=socket(AF_INET,SOCK_DGRAM,0);
 local_addr.sin_family=AF_INET;
 local_addr.sin_addr.s_addr=INADDR_ANY;
 local_addr.sin_port=htons(6969);
 bind(my_sock,(sockaddr *) &local_addr, sizeof(local_addr));
 int client_addr_size = sizeof(client_addr);
 while (1) {
  SwitchToFiber(f2);
  int bsize = recvfrom(my_sock, buf,  MAXBUF, 0,  (sockaddr *) &client_addr, &client_addr_size);
  buf[bsize]=0;
  printf(buf);
  for (int i=0; i<bsize/2; i++) {
      char b=buf[i]; buf[i] = buf[bsize-i-1]; buf[bsize-1-i]=b;
}
  sendto(my_sock, buf, strlen(buf), 0,   (sockaddr *)&client_addr, sizeof(client_addr));
 }
 closesocket(my_sock);
 WSACleanup();
}

void WINAPI Fiber2(void *q) { // «Фибер» - клиент
  HOSTENT *hst;
  char* mess[]={"Леша на полке клопа нашел", "Около Миши молоко", "Ешь немытого ты меньше", "Иди, Сеня, не сиди",
              "А Леву Алла увела", "Я не стар, брат Сеня", "И любит Сева вестибюли"};
  char buf[MAXBUF];  WSADATA d;  SOCKET my_sock;
  sockaddr_in dest_addr, server_addr;
  WSAStartup(0x202, &d);
  my_sock=socket(AF_INET, SOCK_DGRAM, 0);
  dest_addr.sin_family=AF_INET;
  dest_addr.sin_port=htons(6969);
  if (inet_addr("127.0.0.1"))
    dest_addr.sin_addr.s_addr=inet_addr("127.0.0.1");
  else
    if (hst=gethostbyname("127.0.0.1"))
      dest_addr.sin_addr.s_addr=((unsigned long **) hst->h_addr_list)[0][0];
  int server_addr_size=sizeof(server_addr);
  int i = 0;
  while (1) {
   if (!mess[i][0]) SwitchToFiber(f0);
   CharToOem(mess[i], buf);
   sendto(my_sock, buf, strlen(buf), 0,  (sockaddr *) &dest_addr, sizeof(dest_addr));
   SwitchToFiber(f1);
   int n=recvfrom(my_sock, buf, sizeof(buf)-1, 0, (sockaddr *) &server_addr, &server_addr_size);
   buf[n]=0;
   printf(" <-> %s\n", buf);
   i++;
  }
  closesocket(my_sock);
  WSACleanup();
}
main() { // Главный поток
 f1 = CreateFiber(0, Fiber1, NULL);
 f2 = CreateFiber(0, Fiber2, NULL);
 f0 = ConvertThreadToFiber(NULL); // Объявить себя волокном
 SwitchToFiber(f1);
}
```

Пример 3. Демонстрация запуска параллельных процессов и организации обмена между ними при помощи анонимных каналов. Родительский процесс запускает дочерний процесс и переназначает его стандартные дескрипторы

ввода-вывода на свои каналы. После этого дочерний процесс общается с родительским при помощи стандартных функций консольного ввода-вывода.

```
// Anon1 – родительский процесс
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <Windows.h>
#define BUFLEN 16
char mess[]={"Hello!\n"};
char buf[BUFLEN];
int main( void ) {
 HANDLE r1, w1, r2, w2; DWORD n;
 STARTUPINFO si;
 PROCESS_INFORMATION pi;
 SECURITY_ATTRIBUTES sa;
 sa.nLength = sizeof(SECURITY_ATTRIBUTES);
 sa.lpSecurityDescriptor = NULL;
 sa.bInheritHandle = TRUE;
 CreatePipe(&r1, &w1, &sa, 4096);       // Создать 1-й канал
 CreatePipe(&r2, &w2, &sa, 4096);       // Создать 2-ой канал
 memset(&si, 0, sizeof(si));
 si.dwFlags    =  STARTF_USESTDHANDLES;
 si.hStdInput  = r2;                    // Переназначить stdin
 si.hStdOutput = w1;                    // Переназначить stdout
 si.hStdError  = GetStdHandle(STD_ERROR_HANDLE);
 si.cb   = sizeof(STARTUPINFO);
 CreateProcess(".\\anon2.exe", "" , NULL, NULL, TRUE, 0 , NULL, NULL, &si, &pi);
 CloseHandle(pi.hThread);
 CloseHandle(w1);                       // Эти дескрипторы...
 CloseHandle(r2);                       // ...не нужны
 WriteFile(w2, mess, strlen(mess), &n, NULL );    // Записать "Hello!"
 ReadFile (r1, buf,  BUFLEN, &n, NULL );          // Прочитать ответ
 fprintf(stderr, "%s", buf);                      // Отобразить ответ
 WaitForSingleObject(pi.hProcess, INFINITE);     // Ждать завершения
 CloseHandle(w2);
 CloseHandle(r1);
 }

// Anon2 – дочерний процесс
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#define BUFLEN 16
char buf[BUFLEN];
main() {
// scanf("%s", buf);              // Тоже можно
// fscanf(stdin, "%s", buf);      // Тоже можно
// gets(buf);                     // Тоже можно
 cin >> buf;                    // Прочитать из канала
 for (int i=0;i<strlen(buf);i++) {          // Изменить строку
  if (buf[i]=='l') buf[i]='1';  if (buf[i]=='o') buf[i]='0';
 }
 cout << buf;                   // Записать в канал
// printf("%s", buf);             // Тоже можно
// fprintf(stdout, "%s", buf);    // Тоже можно
// puts(buf);                     // Тоже можно
}
```

Пример 4. Демонстрация «упрощенного» запуска параллельных процессов, обмена между ними при помощи именованных каналов и синхронизации посредством мьютексов. Главный процесс передает дочернему строку, тот дожидается передачи и отображает полученную строку на экране.

```
// Главный процесс - писатель
#include <stdio.h>
#include <windows.h>
int main() {
 char data[] = {"Hello world!"};
 DWORD n; HANDLE m;
 HANDLE pipe = CreateNamedPipe("\\\\.\\pipe\\my_pipe", PIPE_ACCESS_OUTBOUND, PIPE_TYPE_BYTE, 1, 0, 0, 0, NULL);
 WinExec(".\\mut2.exe", SW_SHOW);
 while ((m = OpenMutex(MUTEX_ALL_ACCESS, TRUE, "mymut"))==NULL);
 WaitForSingleObject(m, INFINITE);
 // ConnectNamedPipe(pipe, NULL); // Можно вместо мьютекса
 WriteFile( pipe, data, strlen(data), &n, NULL);
 CloseHandle(pipe);
 char c; scanf("%c", &c);
}

// Дочерный процесс - читатель
#include <stdio.h>
#include <windows.h>
int main() {
 char buffer[128]; DWORD n;
 HANDLE pipe = CreateFile("\\\\.\\pipe\\my_pipe", GENERIC_READ,  FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
                          OPEN_EXISTING,  FILE_ATTRIBUTE_NORMAL, NULL);
 HANDLE m = CreateMutex(NULL, TRUE, "mymut");
 ReleaseMutex(m);
 ReadFile( pipe, buffer, 127 , &n, NULL );
 for (int i=0;i<n;i++) printf("%c", buffer[i]);
 CloseHandle(pipe);
 char c; scanf("%c", &c);
}
```

Пример 5. Демонстрация запуска параллельных процессов и организации обмена между ними при помощи «почтовых ячеек», синхронизация посредством семафоров. Главный процесс выполняет роль сервера, создает почтовую ячейку, дожидается запуска процесса-клиента и передачи данных, после чего читает эти данные и отображает их на экране.

```
// Sem1 - Главный процесс (читатель)
#include <stdio.h>
#include <windows.h>
#define BUFLEN 32
main() {
 STARTUPINFO si;
 PROCESS_INFORMATION pi;
 char buffer[BUFLEN];
 unsigned long n;  // Сколько прочитано
 HANDLE s = CreateSemaphore(NULL, 0, 1, "mysem");      // Создать закрытый семафор
 HANDLE m = CreateMailslot("\\\\.\\mailslot\\myslot", 0, MAILSLOT_WAIT_FOREVER, NULL);
 ZeroMemory(&si, sizeof(STARTUPINFO));
 CreateProcess(".\\sem2.exe", "", NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
 WaitForSingleObject(s, INFINITE);                     // Ждать открытия семафора и...
 ReadFile(m, &buffer, BUFLEN, &n, NULL);               // ... читать
 for (int i=0;i<n;i++) printf("%c", buffer[i]);
```

```
}

// Sem2 -  Дочерний процесс (писатель)
#include <stdio.h>
#include <windows.h>
main() {
 char message[] = {"Hello world!"};
 unsigned long n;  // Сколько записано
 HANDLE s = OpenSemaphore(SEMAPHORE_ALL_ACCESS, TRUE, "mysem");
 HANDLE m = CreateFile("\\\\.\\mailslot\\myslot", GENERIC_WRITE,  FILE_SHARE_READ, NULL, OPEN_EXISTING,
                                FILE_ATTRIBUTE_NORMAL, 0);
 WriteFile(m, message, strlen(message), &n, NULL);          // Записать данные и...
 ReleaseSemaphore(s, 1, NULL);                              // ...открыть семафор
}
```

Пример 6. Демонстрация запуска потоков, обмена данными между ними при помощи файлов и синхронизации при помощи «событий». Главный поток работает с файлом при помощи операций чтения-записи, а дочерний – как с массивом, предварительно отобразив файл на память.

```
#include <windows.h>
#include <iostream.h>
DWORD WINAPI mythread1(void *q) { // Поток-читатель
 HANDLE e1 = OpenEvent(EVENT_ALL_ACCESS, TRUE, "ev1");
 HANDLE e3 = OpenEvent(EVENT_ALL_ACCESS, TRUE, "ev3");
 while(1) {
  WaitForSingleObject(e1, INFINITE); // Ждать разрешения
  HANDLE f  = CreateFile(".\\file.dat",GENERIC_READ|GENERIC_WRITE,  FILE_SHARE_READ|FILE_SHARE_WRITE,
                                NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL); // Открыть файл
  HANDLE m = CreateFileMapping(f, 0, PAGE_READONLY, 0, 0, NULL);
  unsigned char *p = (unsigned char *) MapViewOfFile(m, FILE_MAP_READ, 0, 0, 0);
  cout << p[0];                    // Отобразить переданную букву
  CloseHandle(m);
  CloseHandle(f);
  SetEvent(e3);                    // Доложить о выполнении
 }
}
int main() { // Главный поток (писатель)
 DWORD n; int i=0;
 char s[] = {"Hello world!"};
 HANDLE e1 = CreateEvent(NULL, FALSE, FALSE, "ev1");
 HANDLE e3 = CreateEvent(NULL, FALSE, FALSE, "ev3");
 HANDLE h1 = CreateThread(NULL, 0, &mythread1, NULL, 0, NULL);
 while (s[i]) {
 HANDLE h  = CreateFile(".\\file.dat",GENERIC_READ|GENERIC_WRITE,  FILE_SHARE_READ|FILE_SHARE_WRITE,
                               NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL); // Создать файл
  WriteFile(h, &s[i], 1, &n, NULL);          // Передать букву
  CloseHandle(h);                            // Закрыть файл
  SetEvent(e1);                              // Разрешить отображение на экран
  WaitForSingleObject(e3, INFINITE);    // Ждать доклада
  i++;
 }
 CloseHandle(e1); CloseHandle(e3);
}
```

Приложение В. Коды ошибок

Таблица В.1 - Ошибки Win API, возвращаемые функцией GetLastError()

Код	Описание ошибки
0	Операция успешно завершена
1	Неверная функция
2	Не удается найти указанный файл
3	Системе не удается найти указанный путь
4	Системе не удается открыть файл
5	Отказано в доступе
6	Неверный дескриптор
7	Повреждены управляющие блоки памяти
8	Недостаточно памяти для обработки команды
9	Неверный адрес управляющего блока памяти
10	Ошибка в среде
11	Попытка загрузить программу, имеющую неверный формат
12	Код доступа неверен
13	Недопустимые данные
14	Недостаточно памяти для завершения операции
15	Системе не удается найти указанный диск
16	Не удается удалить папку
17	Не удается переместить файл на другой диск
18	Больше файлов не осталось
19	Носитель защищен от записи
20	Не удается найти указанное устройство
21	Устройство не готово
22	Устройство не опознает команду
23	Ошибка CRC
24	Длина команды слишком велика
25	Не удается найти заданную область или дорожку на диске
26	Нет доступа к диску или дискете
27	Не удается найти сектор на диске
28	Нет бумаги в принтере
29	Не удается произвести запись на устройство
30	Не удается произвести чтение с устройства
31	Устройство не работает
32	Файл занят другим процессом
33	Файл заблокирован другим процессом
36	Слишком много файлов открыто для совместного доступа
38	Достигнут конец файла
39	Нет места на диске
50	Запрос не поддерживается
51	Сетевой путь не найден
52	Имя уже существует в сети
53	Сетевой путь не найден
54	Сеть занята
55	Сетевой ресурс или устройство недоступны
57	Аппаратная ошибка сетевого адаптера
58	Сервер не может выполнить операцию
59	Непредвиденная сетевая ошибка
61	Очередь печати переполнена
62	На сервере нет места для ожидающего печати файла
63	Файл удален из очереди печати
64	Указанное сетевое имя недоступно
65	Нет доступа к сети
66	Неверно указан тип сетевого ресурса
67	Не найдено сетевое имя
68	Превышен предел числа имен для сетевого адаптера локального компьютера

71	Число подключений к узлу сети достигло предела
72	Работа указанного принтера или дискового накопителя была остановлена
80	Файл уже существует
82	Не удается создать файл или папку
84	Недостаточно памяти для обработки запроса
85	Имя локального устройства уже используется
86	Сетевой пароль указан неверно
87	Параметр задан неверно
88	Ошибка записи в сети
89	Не удается запустить другой процесс
100	Не удается создать еще один системный семафор
101	Семафор занят другим процессом
102	Семафор установлен и не может быть закрыт
103	Семафор не может быть установлен повторно
104	Запросы к семафорам во время обработки прерываний не допускаются
105	Этот семафор более не принадлежит использовавшему его процессу
107	Программа была остановлена, так как нужный диск отсутствует
108	Диск занят или заблокирован другим процессом
109	Канал закрыт
110	Не удается открыть указанное устройство или файл
111	Слишком длинное имя файла
112	Недостаточно места на диске
119	Ошибка обработки команды
120	Функция не поддерживается
121	Превышен таймаут семафора
122	Область данных, переданная по системному вызову, слишком мала
123	Синтаксическая ошибка в имени файла, имени папки или метке тома
124	Неверный уровень системного вызова
125	У диска отсутствует метка тома
126	Не найден указанный модуль
127	Не найдена указанная процедура
128	Дочерние процессы, окончания которых требуется ожидать, отсутствуют
130	Неверная операция доступа к разделу диска
131	Попытка поместить указатель на файл перед началом файла
132	Указатель на файл не может быть установлен на заданное устройство или файл
144	Эта папка не является подпапкой корневой папки
145	Папка не пуста
146	Указанный путь используется для отображенного диска
147	Недостаточно ресурсов для обработки команды
148	Указанный путь в настоящее время использовать нельзя
154	Длина метки тома превосходит предел, установленный для файловой системы
160	Неверны один или несколько аргументов
161	Указан недопустимый путь
162	Сигнал уже находится в состоянии обработки
164	Создание дополнительных потоков команд невозможно
167	Не удается снять блокировку с области файла
170	Требуемый ресурс занят
173	Запрос на блокировку соответствует определенной области
174	Файловая система не поддерживает указанные изменения типа блокировки
180	Системой обнаружен неверный номер сегмента
183	Невозможно создать файл, так как он уже существует
186	Передан неверный флаг
187	Не найдено указанное имя системного семафора
196	Операционная система не может запустить это приложение
197	Конфигурация ОС не рассчитана на запуск этого приложения
199	Операционная система не может запустить это приложение
200	Сегмент кода должен быть меньше 64 КБ
203	Системе не удается найти указанный параметр среды
205	Ни один из процессов в дереве команды не имеет обработчика сигналов

206	Имя файла или его расширение имеет слишком большую длину
208	Подстановочные знаки * и/или ? заданы неверно
209	Отправляемый сигнал неверен
210	Не удается установить обработчик сигналов
212	Сегмент заблокирован и не может быть перемещен
214	Слишком много динамически подключаемых модулей
215	Вызовы LoadModule() не могут быть вложены
220	Файл заблокирован другим пользователем
221	Перед сохранением изменений необходимо разблокировать файл
222	Сохраняемый или полученный файл заблокирован
223	Размер файла превышает максимум, сохранение файла невозможно
224	Доступ запрещен
225	Операция не завершена, поскольку файл содержит вирус
226	Это файл содержит вирус и не может быть открыт
229	Канал локальный
230	Неправильное состояние канала
231	Все копии канала заняты
232	Идет закрытие канала
233	С обоих концов канала отсутствуют процессы
234	Имеются дополнительные данные
240	Сеанс был прекращен
254	Имя дополнительного атрибута было задано неверно
255	Дополнительные атрибуты несовместимы между собой
258	Время ожидания операции истекло
259	Дополнительные данные отсутствуют
266	Не удается использовать функции копирования
267	Неверно задано имя папки
275	Дополнительные атрибуты не уместились в буфере
282	Установленная файловая система не поддерживает дополнительные атрибуты
288	Попытка освободить не принадлежащий процессу объект синхронизации
298	Слишком много попыток занесения события для семафора
299	Запрос ReadProcessMemory() или WriteProcessMemory() выполнен частично

Таблица В.2 – Ошибки WinSock, возвращаемые функцией WsaGetLastError()

Код	Описание ошибки
10004	Выполнение блокирующей операции прервано
10013	Отказано в доступе к сокету
10014	Неверный указатель при вызове функции
10022	Неправильный параметр функции
10024	Слишком много открытых сокетов
10035	Ресурс временно недоступен
10036	Одна блокирующая операция уже выполняется
10037	Одна неблокирующая операция уже выполняется
10038	Сокет уже закрыт
10039	Требуется адрес назначения
10040	Сообщение слишком длинное при передаче дейтаграммы
10041	Протокол не поддерживается для данного сокета
10042	Неправильная опция или уровень заданы в функциях опций сокетов
10043	Запрошенный протокол не сконфигурирован для работы с системе
10044	Сокет данного типа не поддерживается
10045	Операция с сокетом не поддерживается
10046	Семейство протоколов не поддерживается
10047	Адрес не поддерживается для выбранного протокола
10048	Адрес и порт уже используются
10049	Невозможно использовать запрошенный адрес для привязки в порту
10050	Сеть неработоспособна
10051	Сеть недоступна, маршрутизация не настроена
10052	Соединение разорвано из-за сбоя при выполнении операции

10053	Соединение разорвано
10054	Соединение на удаленном узле разорвано
10055	Не места в буфере или очереди
10056	Сокет уже подсоединен
10057	Сокет не подсоединен
10058	Невозможен обмен через сокет, поскольку выполнен shutdown
10060	Исчерпано время таймаута
10061	Удаленный узел отказал в соединении, возможно, не запущен сервер
10064	Компьютер, с которым производится попытка соединения, выключен
10065	К удаленному компьютеру не найдет маршрут пересылки пакетов
10067	Запущено слишком много процессов, использующих WinSocket
10091	Сетевая подсистема недоступна
10092	Неверная версия библиотеки winsockdll
10093	Подсистема сокетов не проинициализирована при помощи WSAStartup()
10109	Запрошенный тип класса не найден
10101	Удаленный компьютер инициировал завершение соединения
11001	Запрошенное имя компьютера не найдено
11002	Временная ошибка при определении адреса узла
11003	Фатальная ошибка при определении адреса узла
11004	Числовой адрес для данного имени отсутствует

Методические материалы

ПРОГРАММИРОВАНИЕ МНОГОЗАДАЧНОСТИ В WINDOWS

Методические указания

Составитель ***Климентьев Константин Евгеньевич***

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ имени академика С.П.Королева»
(Самарский университет)
443086, САМАРА, МОСКОВСКОЕ ШОССЕ, 34

Изд-во Самарского университета
443086, Самара, Московское шоссе, 34.