Министерство образования Российской Федерации

**Комсомольский-на-Амуре государственный университет**

**Факультет компьютерных технологий**

**кафедра «Математического обеспечения и применения ЭВМ»**

**Тихомиров В.А.**

**ПРОЦЕССЫ, ПОТОКИ и НИТИ В**

**ОПЕРАЦИОННОЙ СИСТЕМЕ**

**WINDOWS**

лабораторная работа по курсу

“Операционные системы”

##### Комсомольск-на-Амуре

Цель лабораторной работы: Изучить теоретические вопросы управления потоками в ОС Windows и освоить приемы практической реализации этого управления с использованием системных функций.

Задание на лабораторную работу:

* внимательно изучить ниже прилагаемый теоретический материал;
* набрать и проверить работоспособность программных модулей, представленных на листингах 1-7. Тексты программ вместе с результатами их тестирования вставить в отчет;
* выполнить индивидуальное задание в соответствии с выданным преподавателем вариантом:

|  |  |
| --- | --- |
| № вар. | Содержание задания |
| 1 | Программа должна запускать другую программу, путь которой указан в командной строке и анализировать код ее завершения. |
| 2 | Составить программу, которая считает количество счастливых билетов в трамвайном рулоне и выводит их на экран в десять потоков. Каждый поток заведует своим диапазоном номеров: 000000 – 199999,  200000-299999, 300000-399999 … 900000 – 999999. |
| 3 | Программа создает три потока, в которых делятся друг на друга случайные числа пока их число не станет 100000 или не произойдет деление на 0. Анализируется код возврата. |
| 4 | Программа создает десять потоков. Каждый поток создает свою матрицу размером 1000х1000 элементов и заполняет ее случайными целыми числами. Далее каждый поток находит среднее арифметическое среди чисел матрицы и выводит результат на экран. |
| 5 | Программа создает три потока и каждом запускает расчет разных сложных функций. После завершения потоков в главную программу возвращаются характеристики рабочего набора этих потоков и результаты выполнения расчетов. Вся информация выводится на экран. |
| 6 | Главный поток создает матрицу 10000\*10000 элементов и заполняет ее случайными числами. Затем поражается три потока, один – считает среднее арифметическое матрицы, второй – среднее геометрическое, а третий – средне квадратичное. Результаты выводятся на экран. Так же на экран выводится метка достижения каждого потока середины своих вычислений. |
| 7 | Составить программу для ответа на вопрос: как много потоков может создать приложение, прежде чем ОС начнет «беспокоиться». Загрузку процессора контролировать системным монитором. |
| 8 | Пять потоков генерируют случайные целые числа в диапазоне от 1 до 49. Шестой поток – анализирует эти числа. Если появляется пара одинаковых чисел – выводится сообщение ПАРА и номера потоков этой пары. Если три числа одинаковы – ТРИ, Четыре – ЧЕТЫРЕ и пять – ПЯТЬ. Через 60 секунд работы программа завершается. Если не было ни одного совпадения – выводится об этом сообщение. |
| 9 | Главный поток преобразует себя в нить. Затем создает еще пять нитей. Главная нить по очереди передает управления каждой из пяти нитей. Те, выполняя действия (любые) возвращают управление главной нити. |
| 10 | Создать три потока, контролирующие бюджет Смитта, Ганса и Ивана через локальную память потоков (TLS). Потоки обращаются к функции назначения выплат (генератор случайных чисел в интервале от   –1000 до +1000). Функция передает обращающемуся потоку выработанную сумму, которую поток добавляет контролируемому абоненту. Вывести на экран суммы, которые накопятся у абонентов через 10 секунд работы программы. |
| 11 | Создать три потока и, используя технологию APC, выполнить по очереди назначенные этим потокам функции АРС. |

* тщательно откомментированный текст составленной программы вместе с результатами ее тестирования представить в отчете. Там же привести теоретический материал, на основе которого выполнялось задание.
* отчет оформить в электронной форме согласно СТП КнАГТУ.

**Введение**

Запуская программу в Windows, вы создаете *процесс.* В этом нет ничего уди­вительного, так как в других операционных системах обычно происходит то же самое. Все же в других операционных системах дела обстоят несколько иначе. Дело в том, что в Windows процесс владеет оперативной памятью, открытыми файла­ми и другими ресурсами. Однако в Windows процесс не исполняется. Исполня­ется программный *поток.* В момент создания процесс получает в собственное владение единственный программный поток. Поток (который на самом деле яв­ляется всего лишь набором процессорных регистров) обладает указателем на точку исполнения (команда, выполняемая в текущий момент) и указателем на стек (где хранятся собственные локальные переменные, принадлежащие потоку).

Какая разница между процессом и потоком? Если каждый процесс обладает только одним потоком, то разницы фактически никакой. Однако поток может создавать другие потоки. А те потоки могут создавать еще потоки. Два процесса не могут обладать общими ресурсами, если при этом они не используют специ­альных механизмов операционной системы. Напротив, все потоки, принадлежа­щие одному процессу, имеют доступ ко всем ресурсам этого процесса, включая память, открытые файлы и другие ресурсы.

Чтобы запустить новую про­грамму в Windows система использует вызов **CreateProcess**. Новый процесс является полностью новым и име­ет очень мало общего с породившим его процессом. В частности, новый процесс не сможет получить доступа к данным родительского процесса, не воспользовав­шись для этого специальными механизмами операционной системы. Конечно, в рамках текущего процесса можно создать несколько программных потоков, од­нако все они будут иметь доступ к единственному экземпляру начальных дан­ных. Таким образом, если один из потоков модифицирует эти данные, это ска­жется на работе всех остальных потоков.

Конечно же, для решения проблемы можно использовать множество различ­ных способов. Например, можно поместить начальные данные в область памяти общего доступа и позволить каждому из процессов скопировать данные в соб­ственное адресное пространство. Наконец, можно поместить начальные данные в файл, к которому будут иметь доступ все процессы.

Многозадачность Win32 существенно отличается от многозадачности Win 16. В Windows 3.1 для того, чтобы несколько программ могли выполняться одновре­менно, все они должны определенным образом взаимодействовать между собой. При этом переключение между программами может произойти только в момент, когда выполняющаяся в данный момент программа обращается к операционной системе специальным образом. Если исполняемая в данный момент программа перестала обращаться к системе, все остальные программы теряют возможность продолжить работу. Таким образом, вся система может зависнуть по вине одной программы. Такой метод называется *кооперативной* многозадачностью. В Win32 многозадачность организована более надежно. В любой момент времени опера­ционная система имеет право временно прекратить выполнение одной програм­мы и переключить процессор на выполнение другой программы. Переключение происходит вне зависимости от желания каждой из программ.

Такой подход называется *приоритетной* или *вытесняющей* многозадачностью и обладает массой преимуществ. В рабочей среде Win32 некорректно написанная программа не сможет блокировать работу всей остальной системы. Медленно работающие программы не замедлят выполнения остальных процессов. Однако программисты, разрабатывающие программное обеспечение для приоритетной многозадачной среды, обязаны учитывать множество особенностей, на которые можно было не обращать внимания в Win 16.

Программа, работающая в Windows 3.1, обладает полным контролем над всем компьютером до тех пор, пока она не обратиться к одному из специальных си­стемных вызовов. Что это значит? Если вы открываете файл и начинаете за­пись данных в этот файл, ни одна другая программа не сможет обратиться к этому файлу до тех пор, пока вы сами этого не захотите. Однако при использовании приоритетной многозадачности это утверждение перестает быть верным, Предположим, вы открыли файл и начали записывать в него информацию. Однако в процессе выполнения этой процедуры Windows может прервать ис­полнение вашей программы. Управление будет передано другому программно­му потоку. Что произойдет, если другой поток попытается открыть тот же самый файл и произвести в него запись? Если ваш поток открыл файл для эксклюзивного доступа, другой поток просто не сможет открыть файл. Возник­нет ошибка доступа.

Чтобы предотвратить возникновение подобных проблем, необходимо проек­тировать приложения с учетом всех особенностей исполнения программ в при­оритетной многозадачной среде. При этом может потребоваться использование специальных механизмов синхронизации и межпроцессорного обмена данными.

Процессы

Если нужно запустить новую программу, вы должны создать новый процесс. Для этого используется стандартный системный вызов **CreateProcess**. Однако пользо­ваться им не очень удобно. Приходится определять значение множества аргу­ментов. Что делать, если вы хотите запустить WordPad только для того, чтобы отобразить содержимое файла README.TXT? Для этой цели можно использо­вать менее сложный механизм,

Какой из вызовов может оказаться полезным в данной ситуации? Проще все­го использовать **WinExec**, Документация Microsoft рекомендует вместо **WinExec** ис­пользовать **CreateProcess**, однако, если для выполнения того или иного действия вполне подходит **WinExec**, почему нельзя использовать именно эту функцию? **WinExec** работает чрезвычайно просто. Фактически внутри **WinExec** происходит обращение к функции **LoadModule** (еще один устаревший системный вызов), ко­торая вызывает CreateProcess со значениями аргументов по умолчанию. При об­ращении к **WinExec** необходимо сообщить имя программы (полный путь или ко­роткое имя исполняемого файла, расположенного в пути поиска), а также способ отображения окна программы (константа, используемая функцией **ShowWindow**, — SW\_SHOW, SW\_HIDE и т, д.), Конечно, программа может проигнорировать этот аргумент и запуститься так, как она сама того пожелает,

В случае если произошла ошибка, функция **WinExec** возвращает значение, мень­шее 32. Если программа успешно запущена, функция возвращает дескриптор но­вой программы (который не может быть меньше 32), После запуска новой про­граммы функция **WinExec** немедленно передает управление программе, из которой был осуществлен вызов, то есть она не ждет момента, пока вновь запущенная про­грамма завершит работу.

Еще один простой вызов, который можно использовать для запуска про­грамм, — это Shell Execute. Этот вызов во многом напоминает **WinExec**, однако он поддерживает обработку типов файлов, зарегистрированных графической оболоч­кой операционной системы. Например, если при помощи **ShellExecute** вы попробуете запустить файл с расширением ТХТ, будет запущена программа Notepad или любая другая программа, которая используется в вашей системе для просмот­ра текстовых файлов. В качестве аргументов функция **ShellExecute** принимает дескриптор окна (на случай, если возникнет необходимость в сообщениях об ошибках) и операционную строку, такую как open (открыть), print (распечатать) или explore (исследовать). В качестве операционной строки можно передать NULL-строку. В этом случае указанный вами файл будет открыт (open). Также функции Shell Execute необходимо сообщить имя файла и любые параметры командной строки (обычно NULL). Наконец, последние два аргумента — это текущий каталог и константа функции **ShowWindow** (как и в случае с **WinExec**).

Возвращаемое значение точно такое же, как и у W-inExec. Если вы указываете в качестве третьего аргумента функции **ShellExecute** имя исполняемого файла, вы можете не использовать другие аргументы, кроме аргумента параметров команд­ной строки и константы **ShowWindow**. Для файлов документов (например, ТХТ или DOC) значение этих аргументов обычно равно NULL.

Функцию **ShellExecute** можно использовать, например, для того, чтобы открыть корневой каталог диска С:

**ShellExecute(handle, "open", "c:\\", NULL, NULL, SW\_SHOWNORMAL);**

Вы можете заменить строку «open» на строку «explore», а также указать в ка­честве третьего параметра имя абсолютно любого каталога.

Другим аналогичным системным вызовом является вызов **ShellExecuteEx**. Этот вызов фактически является полным аналогом **ShellExecute**, однако в качестве ар­гумента он принимает указатель на структуру, поля которой во многом совпада­ют с аргументами вызова **ShellExecute**. Помимо этого по завершении своей рабо­ты вызов **ShellExecuteEx** помещает в одно из полей этой структуры дескриптор вновь запущенного процесса.

Использовать **WinExec** и **ShellExecute** совсем не сложно. Пример простой кон­сольной программы приведен в листинге 1. Программа открывает текстовый файл, используя при этом два различных способа. Сначала происходит обраще­ние к функции W-inExec, которой напрямую сообщается имя программы текстово­го редактора и имя текстового файла, который должна открыть эта программа. Затем для открытия того же файла используется системный вызов **ShellExecute** (при этом, скорее всего, будет запущен WordPad или другая программа, ответ­ственная в вашей системе за открытие текстовых файлов).

**Листинг 1. Открытие текстового файла**

#include <windows.h>

#include <iostream.h>

void main(int argc, char \*argv[])

{

cout<<"Открыто с помощью WinExec\n";

// Открытие целевого файла

if (WinExec("notepad readme.txt",SW\_SHOW)<32)

MessageBox(NULL,"Не могу выполнить WinExec",NULL,MB\_OK);

cout<<"Нажмите Enter\n";

MessageBox(NULL,"Нажмите OK для продолжения","Program Launched",MB\_OK);

cout<<"Открыто с помощью ShellExecute\n";

if (ShellExecute(NULL,"open","readme.txt",NULL,NULL,SW\_SHOW)<(HANDLE)32)

MessageBox(NULL,"Не могу выполнить ShellExecute\n",NULL,MB\_OK);

}

Вызов CreateProcess

Вызовы наподобие **ShellExecute** и **WinExec** очень удобно использовать для выпол­нения простейших действий вроде открытия файлов и запуска программ. Если же вы хотите создать новый процесс, используя при этом некоторые дополни­тельные параметры, вам не обойтись без системного вызова **CreateProcess**. Опи­сание аргументов, принимаемых этим вызовом, приведено в табл. 1 и 2.

**Аргументы вызова CreateProcess Таблица 1**

|  |  |
| --- | --- |
| **Аргумент** | **Описание** |
| lpApplicationName | Имя программы (или NULL, если имя программы указано  в командной строке) |
| lpCommandLine | Командная строка |
| lpProcessAttributes | Атрибуты безопасности для дескриптора процесса,  возвращаемого функцией |
| lpThreadAttributes | Атрибуты безопасности для дескриптора потока,  возвращаемого функцией |
| bInheritHandlers | Указывает, наследует ли новыйЛ1роцесс дескрипторы,  принадлежащие текущему процессу |
| dwCreationFlags | Параметры создания процесса (см. табл. 2) |
| lpEnvironment | Значения переменных окружения (или NULL, в случае  если наследуется текущее окружение) |
| lpCurrentDirectory | Текущий каталог (или NULL, если используется текущий каталог  текущего процесса) |
| lpStartupInfo | Структура STARTUPINFO, содержащая информацию о запуске процесса |
| lpProcessInformation | Возвращаемые функцией дескрипторы и идентификаторы ID  процесса и потока |

|  |  |
| --- | --- |
| **Флаг** | **Значение** |
| CREATE\_DEFAULT\_ERROR\_MODE | He наследовать текущий режим сообщений об ошибках (см. SetErrorMode) |
| CREATE\_NEW\_CONSOLE | Создать новую консоль |
| CREATE\_NEW\_PROCESS\_GROUP | Создать новую группу процессов |
| CREATE\_SEPARATE\_WOW\_VDM | Запустить 16-битное приложение в его собственном  адресном пространстве |
| CREATE\_SHARED\_WOW\_VDM | Запустить 16-битное приложение в адресном пространстве общего доступа |
| CREATE\_SUSPENDED | Создать процесс в приостановленном состоянии  (см, ResumeThread) |
| CREATE\_UNICODE\_ENVIRONMENT | Блок переменных окружения записан в формате  UNICODE |
| DEBUG\_PROCESS | Запустить процесс в отладочном режиме |
| DEBUG\_ONLY\_THIS\_PROCESS | Предотвратить отладку процесса текущим отладчиком (используется при отладке родительского процесса) |
| DETACHED\_PROCESS | Новый консольный процесс не имеет доступа к консоли родительского консольного процесса |

Аргумент lpStartupInfo - это указатель на структуру STARTUPINFO, Поля этой структуры содержат заголовок консоли, начальный размер и позицию нового окна и перенаправление стандартных потоков ввода/вывода. Новая программа может проигнорировать все эти параметры в зависимости от собственного желания. Поле dwFlags этой структуры содержит флаги, установленные в соответствии с тем, какие из остальных полей структуры вы хотели бы использовать при запуске новой программы. Например, если вы сбросите флаг STARTF\_USEPOSITION, поля dwX и dwY структуры STARTUPINFO, содержащие координаты основного окна запускаемой про­граммы, будут проигнорированы. Часто нет надобности использовать какие-либо из этих полей. В этом случае вы все равно обязаны передать функции **CreateProcess** корректный указатель на пустую структуру STARTUPINFO. He забудьте указать раз­мер структуры в поле cd и присвоить полю dwFlags значение 0.

Функция **CreateProcess** записывает в аргумент lpProcessInfomation указатель на структуру, содержащую дескрипторы и идентификаторы ID нового процесса и по­тока. Доступ к этим дескрипторам определяется аргументами lpProcessAttributes и lpThreadAttributes.

Обратите внимание, что некоторые аргументы функции CreateProcess относятся к консольным приложениям. Другие имеют значение для всех типов программ. Зачастую при обращении к **CreateProcess** можно не заполнять структуру STARTUPINFO, однако в любом случае вы обязаны передать функции указатель на существую­щую в памяти структуру, даже если эта структура не заполнена. Функция Create Process возвращает значение типа BOOL, но по завершении работы чрезвычайно полезная для программиста информация размещается функцией в структуре типа PROCESS\_INFORMATION, указатель на которую возвращается при помощи параметра lpProcessInformafion этой функции. В структуре PROCESS\_INFORMATION содержится идентификатор и дескриптор нового процесса, а также идентификатор и дескрип­тор самого первого потока, принадлежащего новому процессу. Эти сведения мо­гут использоваться для того, чтобы сообщить о новом процессе другим програм­мам, а также для того, чтобы контролировать новый процесс,

При создании процесса с использованием вызова CreateProcess можно пере­дать новому процессу по наследству некоторые вещи, в частности дескрипторы открытых файлов. Однако, к сожалению, это редко когда бывает полезным, за ис­ключением случаев, когда вы хотите объединить стандартные потоки ввода/вы­вода нескольких консольных приложений. В этой ситуации дескрипторы файлов стандартных потоков ввода/вывода обладают заранее определенными значения­ми. Даже если новый процесс получает по наследству дескриптор открытого файла, он не может определить его значения, если только оно не определено за­ранее. Конечно, вы можете передать дескриптор в командной строке или в пере­менной окружения, однако это не очень удобный вариант.

Обладай дескриптором процесса, вы можете управлять процессом при помо­щи вызовов, перечисленных в табл. 3, Чтобы определить момент завершения процесса, можно воспользоваться одним из нескольких методов. Во-первых, можно использовать вызов **GetExitCodeProcess**. Этот вызов возвращает либо значение STILL\_ACTIVE (если процесс все еще продолжает работу), либо код завершения процесса (если процесс завершен), В качестве одногоиз аргументов вы передае­те этой функции указатель на переменную, в которую помещается возвращаемое значение. Узнать дескриптор текущего процесса можно при помощи функции **GetCurrentProcess**, Чтобы управлять процессом из другого процесса, вы должны обратиться к функции **OpenProcess**, которой необходимо передать идентификатор ID-процесса. Чтобы открыть процесс для желаемого доступа, вы должны обла­дать необходимыми разрешениями на доступ к процессу. Процесс, создающий новый процесс при помощи функции CreateProcess, уже обладает дескриптором нового процесса (этот дескриптор помещается в структуру PROCESS\_INFORMATION).

**Таблица 3.** Системные вызовы управления процессом

|  |  |  |
| --- | --- | --- |
| **Вызов** | **Назначение** | **Требуемый уровень доступа** |
| CreateRemoteThread | Создает поток в другом  процессе | PROCESS\_CREATE\_THREAD |
| GetExltCodeProcess | Возвращает код  завершения процесса | PROCESS\_QUERY\_INFORMATION |
| GetGuiRecources | Определяет, сколько объектов USER или GDI  (Graphical Device Interface) используется  процессом | PROCESS\_QUERY\_INFORMATION |
| SetPriorityClass | Устанавливает базовый  приоритет процесса | PROCESS\_SET\_INFORMATION |
| GetPriorityClass | Возвращает базовый  приоритет процесса | PROCESS\_QUERY\_INFORMATION |
| SetProcessAfflnltyMask | Определяет, какие  из процессоров  используются процессом  в качестве основных | PROCESS\_SET\_INFORMATION |
| GetProcessAffinityMask | Устанавливает, какие  из процессоров  используются процессом  в качестве основных | PROCESS\_QUERY\_INFORMATION |
| SetProcessPriorityBoost | Позволяет или запрещает Windows динамически изменять приоритет процесса | PROCESS\_SET\_INFORMATION |
| GetProcessPriorityBoost | Возвращает статус  изменения приоритета  процесса | PROCESS\_GET\_INFORMATION |
| SetProcessShutDownParameters | Определяет, в каком порядке система закрывает процессы при завершении работы всей системы | N/A (необходимо вызвать изнутри процесса) |
| GetProcessShutDownParameters | Возвращает статус механизма завершения работы системы | PROCESS\_QUERY\_INFORMATION N/A (необходимо вызвать изнутри процесса) |
| SetProcessWorkingSetSize | Устанавливает минмальный и максимальный до­пустимый объем физической оперативной памяти, используемый процессом | PROCESS\_SET\_INFORMATION |
| GetProcessWorkingSetSize | Возвращает информацию  об использовании физической памяти процессом | PROCESS\_GET\_INFORMATION |
| TerminateProcess | Корректное завершение  работы процесса | PROCESS\_TERMINATE |
| ExitProcess | Немедленное  завершение процесса | N/A (необходимо вызвать  изнутри процесса) |
| GetProcessVersion | Возвращает версию Windows, в среде которой хотел бы работать процесс | N/A (использует ID процесса) |
| GetProcessTimes | Возвращает степень использования CPU процессом | PROCESS\_GET\_INFORMATION |
| GetStartUpInfo | Возвращает структуру  STARTUPINFO, переданную процессу при обращении к CreateProcess | N/A (необходимо вызвать  изнутри процесса) |

Еще один способ определения текущего состояния процесса подразумевает использование функции **WaitForSingleObject**.. Основное назначение **WaitForSingleObject** — определить, находится ли некоторый дескриптор в сигнальном состоянии. Дескриптор процесса переходит в сигнальное состояние тогда, когда процесс завершает свою работу. При обращении к функции **WaitForSingleObject** вы указываете дескриптор процесса и интервал времени в миллисекундах. Если интервал времени равен 0, функция завершает работу немедленно, возвращая текущее состояние процесса. Если интервал времени равен константе INFINIT, функция будет ждать до тех пор, пока интересующий вас процесс не завершит работу. Если вы укажете конкретное цифровое значение интервала времени, функ­ция будет ожидать завершения процесса в течение указанного времени, а затем вернет управление вызвавшей ее программе. Если в течение указанного времени интересующий вас процесс завершит работу, функция **WaitForSingleObject** вернет управление вызвавшей программе и сообщит ей, что дескриптор целевого про­цесса перешел в сигнальное состояние. В противном случае эта функция вернет негативный ответ.

Вне зависимости от того, в каком состоянии находится дескриптор целевого процесса, функция **WaitForSingleObject** также возвращает значение, отражающее ее успешное выполнение. Имейте в виду: то обстоятельство, что дескриптор так и не перешел в сигнальное состояние, не является ошибкой. Чтобы определить состояние процесса, необходимо сравнить значение, которое вернула функция, со значениями WAIT\_OBJECT\_0 (сигнальное состояние) и WAIT\_TIMEOUT (процесс продол­жает функционировать). В случае ошибки функция вернет значение WAIT\_FAILED. Есть еще одно значение, которое может вернуть эта функция. Это значение — WAIT\_ABANDONED, однако, работая с процессами, вы никогда не столкнетесь с этим

значением. Чтобы иметь возможность ожидать завершения процесса, вы долж­ны обладать открытым дескриптором этого процесса с привилегией SYNCHRONIZE.

Помните, что идентификатор ID-процесса — это не то же самое, что дескрип­тор процесса. Дескрипторы процессов нельзя просто так передавать из процесса в процесс. Это означает, что если вы хотите управлять процессом из другого процесса, вы обязаны прежде всего каким-либо образом получить дескриптор управляемого процесса. Для идентификации процесса другими процессами служит идентифика­тор ID этого процесса. Этот идентификатор можно передать из процесса в процесс. Преобразовать идентификатор ID-процесса в его дескриптор можно при помощи функции **OpenProcess**, однако для этого вы должны обладать необходимыми при­вилегиями. Узнать идентификатор текущего процесса можно при помощи функ­ции **GetCurrentProcessId**. Используя этот вызов, вы можете узнать идентификатор собственного процесса и передать этот идентификатор другому процессу. Получив ID вашего процесса, другой процесс сможет открыть его дескриптор.

При обращении к функции **OpenProcess** необходимо указать требуемый уровень доступа к открываемому процессу. Иногда получить доступ к процессу на любом из возможных уровней нельзя, поэтому, выбирая уровень, старайтесь использо­вать именно тот, который нужен вам для выполнения вашей задачи, и не более того. Например, чтобы узнать код завершения процесса, достаточно владеть уров­нем доступа PROCESS\_QUERY\_INFORMATION. Чтобы иметь возможность завершить ра­боту процесса, необходимо обладать уровнем доступа PROCESS\_TERMINATE. Уровни доступа к процессу перечислены в табл. 3. Вы можете запросить предоставле­ние вам полного набора прав доступа к процессу. Для этого предназначен уро­вень доступа PROCESS\_ALL\_ACCESS.

При помощи вызова **LoginUser** программа, обладающая необходимой привиле­гией (конкретно SE\_TCB\_NAME), может определить *лексему* (token) подключенного к системе пользователя. Обладая этой лексемой, можно запустить какой-либо процесс от имени пользователя системы. Другими словами, действия, которые выполняет процесс, будут рассматриваться системой как действия, выполняемые пользователем системы. Запуск процесса от имени пользователя осуществляет­ся при помощи вызова **CreateProcessAsUser** (только в Windows 2000). При этом программа будет запущена от лица пользователя, обладающего указанной лексемой.

В листингах 2 и 3 приведены два простых консольных приложения. Пер­вая программа (MASTER) запускает вторую (SLAVE) и переходит в режим ожи­дания. Программа SLAVE читает идентификатор процесса (PID, Process Identifier) запустившей ее программы из командной строки и ожидает завершения работы программы MASTER. В командной строке программы MASTER вы можете ука­зать полный путь к исполняемому файлу программы SLAVE.

Обе программы иллюстрируют несколько важных технологий:

* использование **CreateProcess**;
* использование **OpenProcess**;
* использование **WaitForSingleObject**.

Обратите внимание, что MASTER не использует большую часть аргументов функции **CreateProcess**, поэтому в данном конкретном случае для запуска SLAVE вполне можно использовать вызов **WinExec** (см. комментарии в листинге 2).

Листинг 2 Master

#include <windows.h>

#include <iostream.h>

#include <stdio.h>

#include <string.h>

void main(int argc, char \*argv[])

{

char cmd[128];

if (argc!=1)

strcpy(cmd,argv[1]);

else

strcpy(cmd,"slave.exe");

int pid=GetCurrentProcessId();

sprintf(cmd+strlen(cmd)," %d",pid);

cout<<"Master: Starting:" << cmd << "\n";

cout.flush();

// Здесь пример использования WinExec

// if (WinExec(cmd,SW\_SHOW)<32)

// Если требуется использовать возможности CreateProcess:

STARTUPINFO info;

memset(&info,0,sizeof(info));

info.cb=sizeof(info);

PROCESS\_INFORMATION pinfo;

if (!CreateProcess(NULL,cmd,NULL,NULL,FALSE,NORMAL\_PRIORITY\_CLASS,

NULL,NULL,&info,&pinfo))

{

cout<<"Master: Slave процесс не запущен\n";

cout<<"Master: попытайтесь указать имя процесса в ком. строке\n";

}

cout<<"Master: Sleeping\n";

cout.flush();

Sleep(15000);

cout<<"Master: Exiting\n";

exit(0);

}

Листинг 3 Slave

#include <windows.h>

#include <iostream.h>

#include <stdio.h> // temp

void main(int argc,char \*argv[])

{

if (argc!=2)

{

cerr<<"Slave: Запустите MASTER.EXE\n";

exit(1);

}

int pid=atoi(argv[1]);

HANDLE process=OpenProcess(PROCESS\_QUERY\_INFORMATION|SYNCHRONIZE,FALSE,pid);

if (!process) cout<<"Slave: Ошибка при открытии процесса\n";

cout<<"Slave: Подождите завершения работы МАСТЕРА\n";

cout.flush();

if (WaitForSingleObject(process,INFINITE)==STATUS\_WAIT\_0)

cout<<"Slave: МАСТЕР завершил работу\n";

else

cout<<"Slave: Неизвестная ошибка\n";

exit(0);

}

Задания и рабочие наборы

Обычно каждому процессу в Windows 2000 назначается так называемый *рабочий набор* (working set), Рабочий набор процесса определяет, какой объем физической памяти диспетчер памяти Windows пытается сохранить за данным процессом, В рабочем наборе указывается минимальное и максимальное количество страниц памяти, которые должны принадлежать данному процессу. Узнать текущий размер можно при помощи вызова **GetProcessWorkingSize**. Если объем памяти, доступный для других приложений, становится неприемлемо маленьким, система может нарушить границы, установленные в рабочем наборе той или иной программы.

Обладая необходимыми для этого привилегиями, программа может изменить собственный рабочий набор при помощи функции **SetProcessWorkingSetSize**. Если обе границы становятся равными 0xFFFFFFFF, Windows сбрасывает на диск всю память, принадлежащую процессу.

Еще одним методом управления процессами в Windows является *зада­ние* (job). Задание — это группа связанных между собой процессов. Создать за­дание можно при помощи функции **CreateJobObject**, а открыть существующее за­дание можно при помощи функции **OpenJobObject**. Обладая дескриптором задания, вы можете добавить к нему любые другие процессы при помощи функции Assign **ProcessToJobObject**,

Зачем может потребоваться объединять процессы в группу? Этому может быть несколько причин. Например, используя функцию **TerminateJobObject**, вы можете разом завершить работу всех процессов одного задания. При помощи вызова **SetInforinationJobObject** вы можете установить рабочий набор одновременно для всей группы процессов, входящих в задание. Этот же вызов можно использовать для назначения ограничений всем процессам, входящим в задание. В частности, вы можете запретить всему заданию обращаться к системному вызову ExitWindows, читать содержимое системного буфера обмена или использовать какие-либо дес­крипторы (например, дескрипторы окон) других процессов. Если вы запретили процессам одного задания доступ к пользовательским дескрипторам, процесс, не принадлежащий заданию, может предоставить процессам задания доступ к пользо­вательскому дескриптору при помощи функции **UserHandleGrantAccess**,

Помимо прочего операционная система позволяет получать статистику, свя­занную с процессами задания. Для этого служит вызов QueryInformationObject. Используя этот вызов, вы можете получить информацию о том, какую нагрузку на центральный процессор создают процессы, входящие в состав задания, а так­же другую подобную информацию. Кроме того, при помощи этого вызова мож­но получить информацию о параметрах конфигурации задания, значения кото­рым присваиваются при помощи функции **SetInformationJobObject**.

Потоки

При создании процесса в системе появляется новый программный поток, при­надлежащий этому процессу. Вначале любой только что созданный процесс об­ладает лишь одним потоком. Этот поток может создавать новые потоки, а эти новые потоки, в свою очередь, могут создавать другие новые потоки. Процесс продолжает свое существование до тех пор, пока в его владении находится по крайней мере один программный поток (или до тех пор, пока не произойдет что-либо, в результате чего весь процесс или все задание не прекратит работу, напри­мер, обращение к функции **TerminateProcess**).

Зачем процессу несколько потоков? Потоки могут выполнять какие-либо дей­ствия в фоновом режиме относительно вашей основной программы. Например, вы можете создать новый программный поток, который будет в фоновом режи­ме осуществлять вывод информации на принтер. Потоки удобно использовать также в случае, если блокирование или подвисание какой-либо процедуры не должно стать причиной нарушений функционирования основной программы. Например, в то время как основная программа выполняет сложные математичес­кие вычисления, отдельный программный поток может осуществлять обмен дан­ными через асинхронный последовательный канал связи (например, через модем). В случае замедления передачи данных через канал или в случае подвисания мо­дема функционирование основной программы не будет нарушено.

Базовый системный вызов, предназначенный для создания потока, — это **CreateThread**. Однако на практике программисты фактически никогда его не использу­ют. Почему? Потому что этот вызов создает абсолютно пустой поток, который в изначальном виде не может использоваться вашей библиотекой (С или MFC).

Если для разработки программы вы используете C++ или MFC, вы должны воспользоваться для создания потока специальной функцией, включенной в со­став применяемой вами библиотеки. Эта функция не только создает новый про­граммный поток, но и размещает в нем специфический код, который необходим для начала работы потока. Благодаря этому коду библиотека сможет работать с только что созданным программным потоком. Если для создания потока вы об­ратитесь к функции **CreateThread**, скорее всего, ваша библиотека сработает некор­ректно.

В состав большинства библиотек обычно входит специальная функция, пред­назначенная для создания нового потока. Например, в составе стандартной биб­лиотеки C++ присутствуют функции **\_beginthread** и **\_beginthreadex** (функция **\_beginthreadex** принимает несколько дополнительных аргументов и позволяет создавать потоки, находящиеся в приостановленном состоянии, а также она воз­вращает вам дескриптор нового потока). Для завершения работы потока также используют­ся специальные функции. Эти функции обеспечивают корректное завершение работы потока средствами используемой вами библиотеки. В библиотеке C++ для этой цели предназначены функции **\_endthread** и **\_endthreadex** (в зависимости от того, какая из функций использовалась для того, чтобы создать поток).

Если в своей программе вы намерены использовать вызов **\_beginthread** или любую другую подобную функцию, вы должны изменить конфигурацию вашего проекта таким образом, чтобы при компиляции использовалась многопоточная версия библиотеки. Этот параметр настраивается при помощи вкладки C/C++ в разделе Code Generation settings (Настройки генерации кода). Параметр имеет на­звание Use Run-Time Library (Используемая библиотека времени исполнения).

Помните: Windows распределяет процессорное время не между процессами, а между потоками. Каждый поток в системе работает независимо. Количество про­цессорного времени, выделяемое конкретному потоку, определяется многими факторами. Во-первых, каждый процесс обладает собственным базовым уровнем приоритета, который присваивается всем принадлежащим ему потокам. Во-вто­рых, каждый из потоков обладает собственным приоритетом, который добавля­ется к базовому значению. Наконец, сама операционная система имеет право ди­намически изменить приоритет некоторого потока, если посчитает, что так будет лучше. Например, если система обнаружит, что устройство ввода готово передать одному из потоков какие-либо данные, и ожидает, пока управление будет пере­дано этому потоку, система может временно повысить приоритет этого потока.

Для управления потоками служат системные вызовы, перечисленные в табл. 4.

Системные вызовы для управления потоками **Таблица 4.**

|  |  |  |
| --- | --- | --- |
| Функция | Требуемый уровень доступа | Описание |
| AttachThreadInput | Нет | Связывает обработку ввода потока с другим потоком. Позволяет передавать фокус ввода окнам другого потока, а также использовать общее состояние ввода |
| CreateRemoteThread | PROCESS\_CREATE\_THREAD | Создает поток в другом процессе |
| CreateThread | Нет | Создает поток в текущем процессе |
| ExitThread | Нет | Завершает работу текущего потока |
| TerminateThread | THREAD\_TERMINATE | Останавливает работу потока без выполнения необходимых под­го­то­ви­тельных процедур |
| GetCurrentThread | Нет | Возвращает дескриптор текущего потока |
| GetCurrentThreadId | Нет | Возвращает идентификатор ID текущего потока |
| GetExitCodeThread | THREAD\_QUERY\_INFORMATION | Возвращает код завершения потока |
| SetThreadPriority | THREAD\_SET\_INFORMATION | Устанавливает уровень  приоритета потока |
| GetThreadPriority | THREAD\_QUERY\_INFORMATION | Возвращает уровень приоритета потока |
| SetThreadPriority Boost | THREAD\_SET\_INFORMATION | Разрешает или запрещает динамическое изменение уровня приоритета данного потока |
| GetThreadPriorityBoost | THREAD\_QUERY\_INFORMATION | Возвращает статус динамического изменения приоритета для данного процесса |
| GetThreadTimes | THREAD\_QUERY\_INFORMATION | Возвращает время, когда был создан или уничтожен поток и какое количество процессорного времени было им использовано |
| SuspendThread | THREAD\_SUSPEND\_RESUME | Временно приостанавливает выполнение потока |
| ResumeThread | THREAD\_SUSPEND\_RESUME | Продолжает работу потока, приостановленного в результате обращения к SuspendThread |
| SetThreadAfflnltyMask | THREAD\_SET\_INFORMATION | Устанавливает, какие процессоры могут использоваться для выполнения данного потока |
| SetThreadIdealProcessor | THREAD\_SET\_INFORMATION | Устанавливает, какой процессор предпочтительней использовать для выполнения потока |
| SwItchToThread | Нет | Переключает процессор на выполнение другого потока (неизвестно, какого именно) |

Пример

Очень часто компьютеры в художественных фильмах обладают совершенно не­удобными, неестественными, а иногда и отталкивающими пользовательскими интерфейсами. Обычно если происходит что-то критическое, компьютер сопро­вождает это событие леденящим душу звуковым сигналом, и все зрители пони­мают: катастрофа совсем рядом.

Конечно, нам с вами не удастся воспроизвести что-либо подобное, однако попробовать стоит. Обратите внимание на листинг 4. Эта простая программа выводит на экран простое диалоговое окно с текстовым сообщением и при этом каждую секунду издает короткий звуковой сигнал, Для воспроизведения этого сигнала используется отдельный программный поток.

Листинг 4. Тревога!

#include <windows.h>

#include <process.h> // необходимо для обращения к \_beginthread

HANDLE mainthread;

void beepthread(void \*)

{

DWORD xitcode;

while (GetExitCodeThread(mainthread,&xitcode)&&xitcode==STILL\_ACTIVE)

{

MessageBeep(-1);

Sleep(1000);

}

}

void main()

{

mainthread=GetCurrentThread();

\_beginthread(beepthread,0,NULL);

MessageBox(NULL,"Red Alert","Alert",MB\_OK);

}

Основная программа сохраняет дескриптор потока в глобальной переменной. Это позволяет потоку, воспроизводящему звук, определить момент, когда основ­ная программа завершит свою работу. В этот момент поток, воспроизводящий звук, также завершит свою работу.

Конечно, для того чтобы установить факт завершения работы основного потока, можно использовать вызов **WaitForSingleObject**, однако в нашем случае удобнее обратиться к **GetExitCodeThread**. Если эта функция возвращает значение STILL\_ACTIVE, значит, основной поток продолжает функционирование, а следовательно, поток beepthread может продолжать пищать.

Что произойдет, если дочерний поток не будет следить за завершением рабо­ты родительского потока? В нашем случае — ничего. Программа будет вести себя абсолютно также. Попробуйте заменить оператор while в листинге 4 на оператор while(1). Все работает так же, как и до модификации исходного кода. Дело в м, что при завершении работы функции main библиотека времени исполнения С обращается к exit, a exit вызывает функцию **ExitProcess**, которая автоматически уничтожает все потоки процесса.

Если говорить о библиотеке времени исполнения С, то можно отметить, что создаваемый поток не обращается к функциям этой библиотеки, поэтому для его создания вместо специальной библиотечной функции \_**beginthread** можно исполь­зовать базовый системный вызов **CreateProcess**.

Локальная память потоков

Иногда бывает удобно использовать некоторую область памяти, которая отно­сится к конкретному потоку, но не является локальной переменной. Представь­те, что вы разрабатываете сетевой сервер. Каждый из клиентов сервера обслужи­вается отдельным потоком. Чтобы выполнить некоторое действие от имени клиента, потоки обращаются к некоторой функции (допустим, ее имя — DoWork). Существуют причины, по которым вы хотите ограничить количество обращений каждого из клиентов к этой функции. Допустим, каждый из клиентов (а следо­вательно, каждый из потоков) имеет право обратиться к этой функции не бо­лее 10 раз.

Каким образом вы можете организовать подобный счетчик обращений? Гло­бальная переменная не подходит. Если вы используете глобальную переменную, ее значение будет увеличиваться каждый раз при обращении любого потока к функции DoWork. Таким образом, если десять потоков обратятся к этой функции по одному разу, значение счетчика станет равным 10, а это не то, что нам нужно. Локальная переменная тоже не подходит, так как при каждом очередном обра­щении к функции DoWork локальной переменной будет присваиваться нулевое значение.

Конечно, проблему можно решить несколькими способами. Например, каж­дый поток может выделить специальную область памяти, принадлежащую лич­но ему и каждый раз при обращении к DoWork передавать этой функции указатель на эту область в качестве аргумента. Возможно, этот способ подойдет для реше­ния конкретно данной проблемы, однако, если вам потребуется несколько подоб­ных переменных, вы можете столкнуться с новыми проблемами.

Рассмотрим решение, предлагаемое операционной системой. Предположим, что вы создаете таблицу с двумя колонками. Таким образом, каждая запись таблицы имеет два поля: в первом поле содержится идентификатор ID-потока, а во вто­ром — 32-битное число. Каждый раз при обращении потока к функции DoWork функция определяет ID текущего потока (при помощи вызова **GetCurrentThreadID**) и ищет его в таблице. Если запись с таким идентификатором есть, функция ис­

пользует эту запись. Если такой записи нет, в таблицу вносится новая запись, содержащая ID текущего потока. В любом случае в вашем распоряжении оказы­вается табличная запись, принадлежащая текущему потоку. Во втором поле этой записи можно хранить счетчик обращения к функции DoWork для потока с задан­ным ID. Если поток завершает работу, запись с соответствующим ID удаляется из таблицы, и на ее месте можно расположить запись, принадлежащую другому потоку.

Именно по такому принципу работает *локальная память потоков* (Thread Local Storage, TLS). Для каждого из процессов создается набор внутренних таблиц. Windows может создать до 64 таких таблиц для каждого процесса. Таким обра­зом, вы можете использовать 64 различные переменные TLS. Когда вы обращае­тесь к **TlsAlloc**, Windows выделяет вам одну таблицу TLS. После этого вы може­те использовать вызовы **TlsSetValue** и **TlsGetValue** для того, чтобы установить или прочитать значение из таблицы. При этом операционная система обращается именно к той записи в таблице, которая соответствует текущему потоку. Если таблица вам больше не нужна, вы можете обратиться к функции **TlsFree** для того, чтобы освободить ее. В каждой записи таблицы можно хранить любое 32-битное число, однако чаще всего таблица TLS используется для хранения указателей на класс или структуру, содержащую все необходимые для потока переменные.

Конечно, TLS — не единственный метод решения проблемы. В частности, вы можете разработать собственное решение, которое будет работать подобно TLS. Другой вариант предусматривает передачу функции в качестве одного из аргу­ментов указателя на область памяти, принадлежащую потоку.

Листинг 5 демонстрирует применение обоих методов. Основная программа создает два потока (t\_l и t\_2). Каждый из потоков случайным образом выбирает два числа (п1 и п2) и обращается к двум разным функциям (fl и f2) количество раз, соответствующее этим числам. Функция fl хранит количество обращений в таблице TLS, которую создает функция main. Чтобы извлечь количество обраще­ний к функции fl из таблицы TLS, в программе используется функция **getflcount**.

Функция f2 также следит за количеством обращений, однако эта функция хранит счетчик в локальной переменной, которая передается ей как аргумент. Этот метод приемлем в ситуации, когда основная программа имеет возможность со­здать локальную переменную.

**Листинг 5.** Использование TLS

#include <windows.h>

#include <iostream.h>

#include <process.h>

#include <stdlib.h>

#include <time.h>

volatile int t1done=0;

volatile int t2done=0;

DWORD index;

void display(char \*display, int f1, int f2)

{

cout<<display<<" f1="<<f1<<" f2="<<f2<<"\n";

}

int getf1count()

{

return (int)TlsGetValue(index);

}

void f1()

{

int ct=(int)TlsGetValue(index);

TlsSetValue(index,(void \*)++ct);

// здесь можете вставить полезный код

}

void f2(int &ctr)

{

ctr++;

// здесь можете вставить полезный код

}

// main thread

void t\_1(void \*)

{

int callf1=0;

int callf2=0;

int n1,n2;

srand( (unsigned)time( NULL ) +100);

n1=rand()%9+1; // количество вызовов f1

n2=rand()%9+1; // количество вызовов f2

display("T1: Вызывов",n1,n2);

while (n1--) f1();

callf1=getf1count();

while (n2--) f2(callf2);

display("T1: Результат",callf1,callf2);

t1done=1;

}

void t\_2(void \*)

{

int callf1=0;

int callf2=0;

int n1,n2;

srand( (unsigned)time( NULL )+245 );

n1=rand()%9+1; // количество вызовов f1

n2=rand()%9+1; // количество вызовов f2

display("T2: Вызывов",n1,n2);

while (n1--) f1();

callf1=getf1count();

while (n2--) f2(callf2);

display("T2: Результат",callf1,callf2);

t2done=1;

}

void main()

{

index=TlsAlloc();

\_beginthread(t\_1,0,NULL);

\_beginthread(t\_2,0,NULL);

while (t1done==0||t2done==0);

TlsFree(index);

cout.flush();

MessageBox(NULL,"Done","Debug",MB\_OK);

}

Нити

Переключение между потоками осуществляется в соответствии с принципом приоритетности. Если система решает, что ваше время истекло, она «отбирает» центральный процессор у вашего потока и передает его в распоряжение другого потока. Однако в некоторых ситуациях может потребоваться более изощренный контроль над передачей управления между участками кода программы. Именно в этих случаях применяют *нити* (fibers).

Преобразовать поток в нить можно при помощи функции **ConvertThreadToFiber**. Каждый поток может обладать несколькими нитями. Когда система переключа­ется с одного потока на выполнение другого, начинается выполнение текущей нити для данного потока. Если в потоке только одна нить, то вы, очевидно, не замените разницы — код, который раньше был частью потока, стал частью нити. Однако если в потоке несколько нитей, то вы можете управлять переключением между ними при помощи функции **SwitchToFiber**. Создать новую нить в текущем потоке можно при помощи функции **CreateFiber**.

Зачем нужны нити? Нити напоминают несколько усложненную форму длин­ного перехода (подобно функции long jump библиотеки С). В Windows 2000 нити используются нечасто, однако в некоторых других операционных системах по­добный механизм находит широкое применение. Таким образом, нити чрезвычайно удобны в случае, если требуется адаптировать для Windows код, написан­ный для какой-либо другой операционной системы (например, Unix), в которой при помощи подобного механизма имитируется выполнение нескольких потоков в рамках одного процесса.

Для операционной системы каждая нить — это поток, который выполняет эту нить в текущий момент. Единственное отличие состоит в том, что нить может контролировать, какой из других нитей того же самого потока будет передано управление. Обратите внимание на то, что переключение между потоками осу­ществляется в прежнем режиме. Нити контролируют взаимную передачу управ­ления только в рамках потока, к которому они принадлежат. Другие потоки вы­полняются независимо от этого.

Простой пример использования нитей приводится в листинге 6. Поток main преобразует сам себя в нить, а затем создает еще три нити. После этого главная нить потока по очереди передает управление каждой из трех остальных нитей. Выполнив действие, каждая из трех нитей возвращает управление главной нити.

Если вам не требуется осуществлять контроль над переключением между вы­полнением разных участков кода, вы вполне сможете обойтись без использова­ния нитей. Обычно большинство задач, связанных с параллельным выполнени­ем различных действий, можно решить при помощи потоков.

**Листинг 6**. Использование нитей

#include <windows.h>

#undef MessageBox

#define MessageBox(s) MessageBoxA(NULL,s,"Fiber Demo",MB\_OK)

void \*fiber[4];

void fiber0(void \*) // основная нить

{

while (1)

{

for (int i=1;i<4;i++)

SwitchToFiber(fiber[i]);

}

}

void CALLBACK fiber1(void \*)

{

while (1)

{

MessageBox("Fiber 1");

SwitchToFiber(fiber[0]);

}

}

void CALLBACK fiber2(void \*)

{

while (1)

{

MessageBox("Fiber 2");

SwitchToFiber(fiber[0]);

}

}

void CALLBACK fiber3(void \*)

{

while (1)

{

MessageBox("Fiber 3");

SwitchToFiber(fiber[0]);

}

}

void main()

{

fiber[0]=ConvertThreadToFiber(NULL);

fiber[1]=CreateFiber(0,fiber1,NULL);

fiber[2]=CreateFiber(0,fiber2,NULL);

fiber[3]=CreateFiber(0,fiber3,NULL);

fiber0(NULL);

}

АРС

Модуль Windows Executive поддерживает несколько необычную технологию под названием Asynchronous Procedure Call (АРС). Эта технология позволяет одно­му потоку асинхронно инициировать выполнение некоторой функции в другом потоке. Иными словами, по команде первого потока выполнение второго потока асинхронно прерывается, второй поток выполняет некоторую функцию, после чего выполнение второго потока продолжается. Звучит впечатляюще, однако, к сожалению, Win32 накладывает на потоки АСР некоторые немаловажные огра­ничения.

Система не может просто так взять и прервать выполнение последовательно­сти команд потока, так как это может привести к возникновению проблем синх­ронизации и т. п. Чтобы не возникло подобных проблем, система разрешает по­току выполнить назначенную для него функцию АРС только в случае, если поток находится в так называемом *«настороженном»* состоянии (alertable state). Пере­вести поток в «настороженное» состояние можно при помощи вызовов **SleepEx, SIngalObjectAndWait, WaitForSingleObjectEx, WaitForMultipleObjectsEx** и **MsgWaitForMultipleObjectsEx**.

На самом деле вызов АРС, предназначенный для выполнения в некотором потоке, ставится в очередь, соответствующую этому потоку. Как только поток переходит в «настороженное» состояние, немедленно происходит последователь­ное выполнение всех функций АРС, находящихся в очереди, соответствующей этому потоку. Если функция АРС поставлена в очередь потока до того, как сам поток начал работу, эта функция будет выполнена в самом начале выполнения потока.

Технология АРС широко используется многими встроенными механизмами Windows (например, для организации *перекрывающегося* (overlapped) ввода/вы­вода. Чтобы поставить функцию АРС в оче­редь некоторого потока, следует использовать системный вызов **QueueUserAPC**. В качестве аргументов эта функция принимает указатель на функцию АРС, дес­криптор потока и данные, которые необходимо передать функции АРС. При об­ращении к **QueueUserAPC** указанная функция АРС ставится в очередь для вы­полнения в указанном потоке. Как только поток переходит в «настороженное» состояние, он одну за другой выполняет все функции АРС, поставленные в его очередь.

Исходный код простой программы, использующей **QueueUserAPC**, приведен в листинге 7. Функция main программы создает новый поток и на короткое вре­мя переходит в режим ожидания для того, чтобы поток успел начать работу. После этого программа обращается к функции QueueUserAPC. Теперь, как только создан­ный нами поток (tl) перейдет в «настороженное» состояние, он выполнит назна­ченную для него функцию АРС (apcfunc). В нашем примере поток переходит в «настороженное» состояние при помощи вызова SleepEx (но не Sleep). Однако, если основная программа не будет ждать, пока новый поток начнет работу, а сразу же, создав поток, поставит в его очередь функцию АРС, эта функция может быть выполнена в момент, когда поток создан, но не успел начать работу. Чтобы про­верить это, закомментируйте вызов Sleep в функции main.

**Листинг 7.** Использование АРС

#include <windows.h>

#include <iostream.h>

volatile int waiting=0;

void CALLBACK apcfunc(DWORD)

{

waiting=1;

}

DWORD CALLBACK t1(LPVOID)

{

cout<<"Поток начал работу\n";

cout.flush();

Sleep(15000);

cout<<"Переходим в настороженное состояние\n";

cout.flush();

SleepEx(10000,TRUE);

MessageBox(NULL,"T1 завершил работу","Debug",MB\_OK);

return 0;

}

void main()

{

HANDLE thread;

DWORD tid;

thread=CreateThread(NULL,0,t1,0,0,&tid);

Sleep(5000); // ждем, пока поток начнет работу

QueueUserAPC(apcfunc,(HANDLE)thread,0);

while (waiting==0);

MessageBox(NULL,"Поток выполнил функцию APC","Debug",MB\_OK);

}

Насколько полезной может оказаться технология АРС для прикладного про­граммиста? На самом деле при создании программ редко кто использует АРС напрямую, однако эта технология широко используется многими полезными ме­ханизмами Windows, такими как, например, перекрывающийся (overlapped) ввод/ вывод.

Заключение

Если раньше вы не имели дело с многозадачными средами, возможно, вам по­требуется некоторое время для того, чтобы изучить тонкости организации мно­гозадачности в Windows, однако, скорее всего, понятия потоков, процессов, ни­тей и функций АРС не будут казаться вам слишком сложными. Однако, если в прошлом вы работали в других операционных системах, возможно механизмы многозадачности, свойственные Windows, покажутся вам непривычными, стран­ными и может быть даже отталкивающими.

Чтобы спроектировать многопоточное приложение, требуется дополнительное время и усилия, однако преимущества многопоточной технологии очевидны. При помощи многопоточности можно решить множество проблем. Кроме того, часто многопоточная программа работает быстрее, чем ее однопоточный аналог. Это особенно заметно на многопроцессорных компьютерах.

Во итогам всего вышеизложенного материала можно сделать следующие резюме:

Самый простой способ запуска программы

Несмотря на то что Microsoft не рекомендует использовать **WinExec**, во многих ситуациях этот устаревший вызов является самым удобным способом запуска внешней программы из вашего кода. Чтобы запустить программу, достаточно пе­редать в качестве первого аргумента командную строку. Второй аргумент потре­буется в ситуации, если вы хотите установить метод отображения основного окна запускаемой программы. Чаще всего второй аргумент вызова **WinExec** получает значение SW\_SHOW, однако вместо этого можно использовать SW\_MINIMIZED, SM\_MAXIMIZED или другие подобные константы. Допустимые значения второго аргумента пере­числены в определении функции **ShowWindow**. Имейте в виду, что запускаемая про­грамма может проигнорировать второй аргумент и осуществить отображение соб­ственного главного окна по своему усмотрению.

**WinExec** не обладает гибкостью **CreateProcess**, однако этот вызов значительно проще использовать. Если запустить программу не удалось, функция возвращает значение, меньшее 32. В противном случае возвращается дескриптор экземпляра запущенной программы. Пример использования **WinExec** приведен в листинге 1.

Более сложный способ запуска программ

В некоторых случаях удобнее использовать другой устаревший системный вы­зов с именем **ShellExecute**. Этот вызов позволяет направить графической оболоч­ке Windows запрос на открытие (open), изучение (explore) или распечатку (print) некоторого документа или каталога. Функцию **ShellExecute** можно использовать для запуска любых исполняемых файлов, но вместо этого вы можете передать этой функции имя документа или файла данных, а оболочка сама определит, какую из программ использовать для обработки этого документа или файла. Помимо этого функция **ShellExecute** позволяет открыть дисковый каталог. Пример использования **ShellExecute** приведен в листинге 1.

Запуск программ методом для настоящих специалистов

Существуют две основные причины, по которым вы можете использовать функ­цию **CreateProcess** для запуска программ. Во-первых, возможно, вам необходим дополнительный контроль над процедурой запуска. Во-вторых, возможно, вы принадлежите к категории людей, которые не ходят проторенными тропами, не любят простых решений и предпочитают справляться с проблемами, используя при этом самые сложные методы.

В качестве аргументов **CreateProcess** принимает имя программы и командную строку. Имя программы можно включить в состав командной строки. Кроме того, в функцию **CreateProcess** следует передать две структуры SECURITY\_ATTRIBUTES, coдержащие информацию о дескрипторах процесса и потока, возвращаемых этой функцией. Структуры содержат информацию о наследовании дескрипторов дочерними процессами, а также об уровнях доступа к этим дескрипторам. Например, если родительский процесс должен синхронизировать себя с дочерним про­цессом, ему потребуется уровень доступа SYNCHRONIZE.

Помимо этого при использовании **CreateProcess** в эту функцию следует пере­дать некоторые управляющие флаги, строки окружения, текущий каталог и струк­туру, в которой будут размещены дескрипторы нового процесса и потока, а так­же их идентификаторы ID.

Как видите, прежде чем обратиться к CreateProcess, требуется выполнить не­мало подготовительной работы. Возможно, если вам нужно всего лишь запустить Notepad, чтобы отобразить на экране содержимое файла README.TXT, будет удоб­нее использовать **WinExec** или **ShellExecute**, о которых мы говорили ранее. Пример использования **CreateProcess** приведен в листинге 3.

Ожидание завершения программы, работа с кодом завершения

Если вы хотите, чтобы ваша программа ожидала завершения работы какого-либо другого процесса, вы можете воспользоваться как минимум двумя способами. Прежде всего можно использовать вызов **GetExitCodeProcess**, который служит для определения кода завершения процесса. Код завершения процесса — это про­стое число, значение которого устанавливается процессом чаще всего для того, чтобы сообщить внешнему миру об успехе или неудаче своего выполнения.

Процесс может установить значение кода завершения в момент завершения своей работы. Для этого можно использовать вызовы **ExitProcess** или **TerminateProcess**. Конечно же, консольная программа может обратиться к exit или вернуть значение функции **main** или **WinMain**. Кроме того, значение кода завершения мо­жет быть установлено в результате возникновения неподдерживаемого програм­мой исключения.

Функция **GetExitCodeProcess** читает значение кода завершения процесса (что­бы воспользоваться этим вызовом, вы должны обладать уровнем доступа PROCESS\_ QUERY\_INFORMATION к дескриптору процесса). Если процесс все еще продолжает ра­ботать, вместо кода завершения функция возвращает значение STILL\_ACTIVE.

Другой способ ожидания завершения программы подразумевает использова­ние вызова **WaitForSingleObject**, которому передается дескриптор процесса (для этого требуется уровень доступа SYNCHRONIZE). По вашему желанию вызов может проверить состояние процесса и немедленно вернуть управление основной про­грамме, также он может ожидать завершения процесса в течение некоторого ус­танавливаемого вами интервала времени. Наконец, вызов может ожидать завер­шения процесса неограниченное время. Пример использования этого вызова приведен в листинге 3.

Существует множество других методов работы с процессами. Соответствую­щие вызовы перечислены в табл. 3.

Создание потока при помощи Windows API

Самым низкоуровневым вызовом, осуществляющим создание нового программ­ного потока, является вызов **CreateThread**. Однако этот вызов не осуществляет подготовительных процедур, настраивающих используемую вами библиотеку (C++ или MFC), на работу в многопоточной среде. Чтобы выполнить все подготовительные процедуры, а затем создать новый поток, следует воспользоваться специальным вызовом, входящим в состав соответствующей библиотеки.

Однако, если когда-либо вам потребуется воспользоваться **CreateThread**, вы можете сделать это без каких-либо затруднений. Пример использования этой функции содержится в листинге 7. Вызовы, предназначенные для работы с по­токами, перечислены в табл. 4.

Создание нового потока при помощи стандартной библиотеки C++

Стандартная библиотека C++ поддерживает вызовы **\_beginthread** и **\_beginthreadex**. При обращении к этим вызовам происходит подготовка библиотеки к функцио­нированию в многопоточной среде, после чего создается новый программный поток. При использовании этих вызовов следует также настроить проект на ис­пользование многопоточной версии библиотеки. Для этого на вкладке C/C++ в категории Code Generation (Генерация кода) при помощи списка Use Run Time Library (Используемая библиотека времени исполнения) выберите любую многопоточ­ную библиотеку. Оба вызова возвращают дескриптор нового потока, который можно использовать при обращении к вызовам, предназначенным для работы с потоками.

Функция **\_beginthreadex** -используется в случае, если необходимо запустить поток в приостановленном состоянии, указать атрибуты безопасности или полу­чить идентификатор ID-потока.

Чтобы остановить работу потока, запущенного при помощи **\_beginthread**, об­ратитесь к функции **\_endthread**. Если для создания потока использовалась функ­ция **\_beginthreadex**, для завершения потока следует использовать функцию \_endthreadex. Функция **\_endthread** автоматически закрывает дескриптор потока, однако функция **\_endthreadex** этого не делает. Поэтому при использовании **\_endthreadex** следует закрыть дескриптор самостоятельно при помощи функции **CloseHandle**.

Отображение окон поверх других окон

В Windows существует понятие потока переднего плана (это поток, владеющий окном, расположенным на переднем плане по отношению к другим окнам про­цесса). Такой поток называют английским термином *foreground thread.* Когда дру­гие потоки создают новые окна, эти окна могут появиться на заднем плане, вме­сто того чтобы возникнуть на переднем плане.

Чтобы исправить такое положение вещей, можно воспользоваться функцией **SetForegroundWindow**, которая переносит новое окно на передний план. Однако, если вы хотите отобразить простое информационное окошко (message box), вы може­те столкнуться с проблемой. Дело в том, что окна типа **MessageBox** создаются и уничтожаются операционной системой фактически без вашего участия, поэтому ваша программа не может получить дескриптор окна типа **MessageBox**. Чтобы разместить окно **MessageBox** на переднем плане, необходимо использовать при его создании флаг MB\_SETFOREGROUND.

Получение дескрипторов процесса и потока

Если вы хотите работать с процессом или потоком, вы должны обладать его дес­криптором. Мало того, вы должны обладать необходимым уровнем доступа. Проще всего получить необходимый дескриптор с необходимым уровнем доступа при помощи вызова **CreateProcess**. Эта функция возвращает вызывающей программе дескрипторы нового процесса и соответствующего ему потока, при этом программе предоставляются необходимые права доступа.

Что делать, если вы хотите управлять процессом, который вы сами не созда­вали? Для этого вам потребуется идентификатор ID этого процесса. Функция **OpenProcess** позволяет, исходя из ID процесса, определить его дескриптор. Этот вызов может оказаться полезным в случае, если вы хотите управлять процессом, который был создан из другого процесса. Дело в том, что передавать дескриптор процесса из процесса в процесс нельзя. Вместо этого допускается передача иден­тификатора ID процесса. Получив ID процесса, передайте его функции **OpenProcess**. Сообщите также необходимые права доступа. Функция вернет дескриптор про­цесса, обладающего указанным идентификатором ID.

Работа с потоками несколько отличается от работы с процессами. Если вы хотите предоставить другому процессу возможность доступа к вашему потоку, вы должны выполнить два вызова. Сначала при помощи функции **GetCurrentThread** необходимо получить специальный дескриптор потока. Этот дескриптор необхо­димо передать функции **DupUcateHandle**, которая создаст дескриптор, который можно передавать другим процессам.

Ожидание завершения потока, работа с кодом завершения

Чтобы подождать завершения работы потока, можно воспользоваться как мини­мум двумя способами. Во-первых, можно использовать функцию **GetExi'tCodeThread**. Основное назначение этого вызова — получение кода завершения потока. Код завершения потока — это число, обычно используемое для оповещения других программ об успехе или неудаче выполнения потока.

Поток может установить значение собственного кода завершения при помо­щи функций **ExitThread** или **TerminateThread**. Конечно же, код завершения можно вернуть в качестве возвращаемого значения главной функции потока.

Функция **GetExitCodeThread** возвращает код завершения потока, завершивше­го свою работу (для получения кода вы должны обладать уровнем доступа THREAD\_ QUERY\_INFORMATION). Однако, если в момент обращения к этой функции поток все еще продолжает функционировать, функция возвращает значение STILL\_ACTIVE.

Второй способ ожидания завершения работы подразумевает использование функции **WaitForSingleObject** в отношении дескриптора потока (для этого нужно обладать уровнем доступа SYNCHRONIZE). В зависимости от вашего желания эта функ­ция может либо проверить текущее состояние потока и немедленно вернуть управ­ление вызвавшей программе, либо ожидать завершения потока в течение заранее установленного промежутка времени, либо ожидать завершения потока неограни­ченное время. Пример использования этой функции приведен в листинге 3.

Вызовы, которые вы можете использовать для работы с потоками, перечисле­ны в табл. 5

**Члены класса CwinThread Таблица 5**

|  |  |
| --- | --- |
| **Член** | **Описание** |
| CreateThread | Создает поток. Вместо этого используйте AfxCreateThread |
| m\_pMainWnd | Основное окно потока |
| m\_pActiveWnd | Текущее активное окно потока |
| m\_bAutoDelete | Если равно «истине», после завершения работы потока объект автоматически уничтожается. По умолчанию равно «истине» |
| m\_hThread | Дескриптор потока |
| m\_nThreadID | Идентификатор ID потока |
| GetThreadPriority | Возвращает уровень приоритета потока |
| SetThreadPriority | Устанавливает уровень приоритета потока |
| SuspendThread | Приостанавливает выполнение потока |
| ResumeThread | Продолжает выполнение потока |
| PostThreadMessage | Помещает сообщение в очередь событий потока |
| Initlnstance | Эту функцию необходимо переопределить для того, чтобы определить новый инициализационный код потока |
| Run | Эту функцию необходимо переопределить для того, чтобы перенастроить цикл обработки сообщений потока |
| PreTranslateMessage | Фильтрует сообщения и обрабатывает нажатия на «горячие»  клавиши |
| PumpMessage | Низкоуровневый модуль получения сообщений |
| Onldle | Вызывается в случае, если сообщения отсутствуют |
| IsIdleMessage | Проверяет наличие специальных сообщений MFC |
| Exitlnstance | Содержит код завершения работы потока. Вы можете переопределить эту функцию |
| ProcessWndProcException | Обрабатывает неподдерживаемые исключения |
| ProcessMessageFilter | Фильтрует сообщения |
| GetMainWnd | Возвращает указатель на основное окно потока |

Использование локальной памяти потока

Локальная память потока (Thread Local Storage, TLS) обеспечивает хранение переменных, являющихся собственностью этого потока. Каждый процесс может иметь до 64 таких переменных. Каждой переменной TLS соответствует таблица с количеством строк, соответствующих количеству потоков, принадлежащих про­цессу. В строке, соответствующей некоторому потоку процесса, хранится значе­ние переменной TLS, соответствующее этому потоку. Таблицу TLS можно выделить в памяти при помощи функции **TlsAlloc**. Эта функция резервирует индекс, который хранится в глобальной переменной. Индекс идентифицирует значение, являющееся глобальным и вместе с тем уникальным для данного потока.

Если поток желает получить доступ к такой переменной, он обращается к **TlsSetValue** или **TlsGetValue**, передавая этим вызовам индекс, возвращаемый функ­цией **TlsAlloc**. Это значит, что для создания переменной TLS функцию **TlsAlloc** необходимо вызвать один раз, в самом начале вашего кода. После этого каждый поток будет использовать один и тот же индекс для доступа к своей собственной переменной TLS.

Если переменная TLS больше не нужна, вы можете освободить память, пере­дав индекс переменной в функцию **TlsFree**. При этом в дальнейшем индекс мо­жет использоваться для идентификации другой переменной TLS. Пример исполь­зования TLS приведен в листинге 5.

Использование нитей

Если вы хотите, чтобы потоки вашего процесса взаимодействовали между собой, вы должны воспользоваться методами синхронизации (след. лаб. работа). Однако в некоторых случаях проще использовать механизм, напоминающий потоки, одна­ко позволяющий различным фрагментам кода контролировать передачу управ­ления от одного из них к другому. Именно для этого предназначены нити.

Создать самую первую нить нельзя. Вместо этого вы должны преобразовать в нить один из существующих потоков. Это выполняется при помощи функции **ConvertThreadToFiber**. В результате поток превращается в нить. Когда Windows передает этому потоку управление, на самом деле выполняется не код потока, а код активной нити текущего потока. Пока поток обладает только одной нитью, исполняемый код активной нити — это то же самое, что код потока.

Однако первая нить может создать другие нити (для этого используется **CreateFiber**). Активная нить потока остается активной до тех пор, пока она не обратится к вызову **SwitchToFiber**. После этого активной становится другая нить. Новая активная нить продолжает быть активной, пока она сама, в свою очередь, не обратится к вызову **SwitchToFiber**.

Существуют также некоторые другие вызовы, при помощи которых можно управлять нитями. Вызов **DeleteFiber** служит для удаления нити. При помощи функции **GetCurrentFiber** можно узнать, какая из нитей является активной в дан­ный момент. Вызов **GetFiberData** возвращает аргумент, переданный основной функ­ции нити. Пример использования нитей приведен в листинге 6

Альтернативы потокам

Механизм многопоточности чрезвычайно удобен для организации выполнения программ в фоновом режиме, однако это не единственная технология, которой вы можете воспользоваться в подобных ситуациях. Если разрабатываемые вами процедуры почти не взаимодействуют между собой, вы всегда можете реализо­вать их как отдельные программы. Операционная система Windows предлагает множество способов, при помощи которых вы можете организовать взаимодей­ствие между такими программами. Вместе с тем обмен данными между процес­сами, конечно же, осуществляется значительно медленнее, чем между потоками.

В некоторых ситуациях вместо обработки данных при помощи нескольких параллельных потоков можно применить обработку данных в свободное от об­работки системных сообщений время (idle processing). Программа может выпол­нять некоторые действия в интервалы времени, когда не поступает никаких сис­темных сообщений. Чтобы реализовать такой подход в обычном С, следует заменить традиционный вызов **GetMessage** обращением к **PeekMessage**. Если **PeekMessage** сообщает о том, что очередь сообщений пуста, ваша программа, не занятая обработкой сообщений, может выполнить небольшой объем другой по­лезной работы.