Министерство образования Российской Федерации

**Комсомольский-на-Амуре государственный университет**

**Факультет компьютерных технологий**

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

**Тихомиров В.А.**

**СИНХРОНИЗАЦИЯ ПОТОКОВ В**

**ОПЕРАЦИОННОЙ СИСТЕМЕ**

**WINDOWS**

лабораторная работа по курсу

“Операционные системы”

##### Комсомольск-на-Амуре

Цель лабораторной работы: Изучить теоретические вопросы управления синхронизацией программных потоков 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 1-4. Тексты программ вместе с результатами их тестирования вставить в отчет;
* выполнить индивидуальное задание в соответствии с выданным преподавателем вариантом:

|  |  |
| --- | --- |
| № вар. | Содержание задания |
| 1 | Один поток готовит матрицу в памяти M = 1000х1000 байт со случайными числами от 0 до 255. Другой поток в это время принимает с клавиатуры два числа X, Y, а третий поток – готовит на экране окно для вывода результатов расчетов. Как только данные с клавиатуры введены – из подготовленного массива выбирается байт с индексом (X,Y) и выводится на экран в окне третьего потока. |
| 2 | Имеется файл F1, в котором записано 20 слов (можно больше). Три потока генерируют случайные числа в диапазоне 0 – 100. Если сгенерированное число больше 90, поток генерирует случайное число n от 1 до 20, открывает файл F1, берет из него слово под номером n, открывает файл F2 и вписывает взятое слово в него файлы F1 и F2 закрываются. Каждый поток должен записать в файл F2 по три слова. Полученное предложение приложите в отчет. |
| 3 | Три потока генерируют случайные числа в диапазоне от 0 до 1000. Если в потоке число попадется больше 900, поток выводит на экран (в случайном месте) окно, в котором непрерывно создаются одним потоком - закрашенные окружности случайных радиусов, другим потоком – прямоугольники, а третьим – треугольники. Одновременно на экране может быть не более пяти окон. Пользователь может закрыть одно из них. Тут же появляется другое. |
| 4 | Пять потоков генерируют случайные числа в интервале от 0 до 1000. Если свободен мьютекс, поток вызывает функцию BIZNES и передает ей свой номер и сгенерированное число. Функция присуммирует полученное число к счету под номером обратившегося потока. Когда все потоки обратятся к функции BIZNES по 10 раз, программа выводит на экран итоговые суммы, накопившиеся на всех пяти счетах. |
| 5 | Когда пользователь нажимает на клавишу по экрану снизу вверх двигается значок \*. При нажатии на слева направо двигается значок >. Одновременно на экране может быть не более 3-х значков каждого типа. |
| 6 | Приложение запускается три раза и создает три окна (последующие запуски к созданию окон не приводят). Когда приложение запускается с ключом /R все окна закрываются. |
| 7 | Создать приложение 1, после запуска которого оно ждет, пока не будет запущено приложение 2 и дважды не будет запущено приложение 3, только тогда оно выводит сообщение «УСЛОВИЕ ВЫПОНЕНО» |
| 8 | Один поток читает файл, указанный в первом параметре программы, второй поток читает файл, указанный во втором параметре программы. Третий поток сравнивает файлы и выводит на экран номера и значения несовпадающих байтов. |
| 9 | 1) Создать поток расчёта сложной функции по случайным входным данным.  2) Создать поток вывода результатов расчёта через каждые 3 сек.  3) Синхронизировать потоки.  4) Реализовать операции полного останова потоков, перевода потоков в режим ожидания, запуска потоков.  Дополнительные требования: - потоки работают бесконечно |
| 10 | Главный поток создает массив A - 60 \* 60 элементов и массив B - 60 \* 60 элементов и обнуляет их. После сего создаются пять потоков, которые заполняют массив А случайными значениями в интервале от 1 до 65535. Необходимо так синхронизовать потоки, чтобы они не затирали данные, введенные друг другом. В массив В при этом помещаются номера потоков, заполнивших соответствующую ячейку.  Массив В вывести на экран. |

* 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

Синхронизация потоков важна по нескольким причинам. В разных ситуациях для этой цели удобнее использовать разные механизмы операционной системы. Рассмотрим несколько возможных вариантов обеспечения синхронизации:

Вариант 1

Предположим, что вы разрабатываете программу различных вариантов обработ­ки данных. Допустим, вы хотите максимизировать прибыли при разных вариан­тах производства, или вы ищете кратчайший путь через лабиринт или оптими­зируете рабочий график. Для решения каждой из этих задач вы должны подготовить начальные данные, а затем проверить каждый из возможных вари­антов их изменения. Подготовкой начальных данных обычно занимается основ­ной поток программы, а анализом каждого из вариантов — отдельный дополни­тельный поток. Например, если вам надо проанализировать десять вариантов производства продукции, вы должны запустить десять потоков. Если существу­ет сто различных путей через лабиринт, вам потребуется сто потоков, каждый из которых будет вычислять длину одного из путей. Когда основной поток завер­шит подготовку начальных данных, дополнительные потоки должны приступить к анализу этих данных при различных условиях. Когда дополнительные потоки завершат все вычисления, основной поток должен сравнить результаты их рабо­ты и сообщить об этих результатах пользователю.

Предположим, что вы планируете начать функционирование всех дополнитель­ных потоков сразу же в начале работы программы. При этом во время, когда основ­ная программа будет заниматься подготовкой начальных данных, дополнительные потоки будут выполнять необходимые для работы каждого из них инициализационные процедуры. Однако дополнительные потоки не должны приступать к вычислениям до тех пор, пока основная программа не завершит подготовку на­чальных данных. Таким образом, завершив инициализацию, дополнительные по­токи должны перейти в режим ожидания, чтобы подождать наступления момен­та, когда начальные данные будут полностью готовы к обработке.

Для подобных случаев в Windows используется объект, который называется *событием* (event). Событие не имеет ничего общего с системными сообщениями Windows. Говоря точнее, событие — это флаг общего доступа для нескольких по­токов. В нашем примере дополнительные потоки будут ожидать до тех пор, пока основная программа не установит для них этот общий флаг. Существует несколько типов событий, каждый из которых предназначен для конкретной цели.

Чтобы сообщить основной программе о том, что дополнительный поток завер­шил анализ некоторого варианта, можно также использовать события. Однако вместо этого можно воспользоваться одним из вызов **GetExitCodeThread**, **WaitForSingleObject** или **WaltForMultipleObjects**. В нашем примере это вполне приемлемое решение, так как в случае, если поток завершает вычисления, надобность в нем отпадает и он может прекратить свое существование.

Вариант 2

Предположим, что вы разрабатываете набор программ, каждая из которых вно­сит информацию об ошибках в общий файл журнала. Что случится, если одна из программ откроет файл, начнет запись, после чего Windows прервет ее исполне­ние и передаст управление другой программе? Предположим, что другая про­грамма также попытается открыть файл и произвести запись в этот файл. Оче­видно, что корректно завершить подобную операцию не удастся. Характер ошибки будет зависеть от того, в каком режиме первая программа открыла файл. Если файл был открыт для эксклюзивного доступа, вторая программа не сможет его открыть. Если файл был открыт для общего доступа, в процессе записи данные, вносимые в файл обеими программами, перемешаются, что вызовет путаницу.

Решить проблему можно при помощи флага, который должен проверяться всеми программами, использующими файл. Если программа хочет открыть файл, она обязана установить флаг, но перед этим необходимо проверить текущее со­стояние этого флага. Если флаг уже установлен, значит, в данный момент файл используется другой программой. В этом случае ваша программа должна подож­дать до тех пор, пока флаг не будет сброшен. Только после этого можно вновь попытаться открыть файл. Таким образом, алгоритм работы с флагом выглядит следующим образом:

1. Прочитать значение флага.

2. Если флаг установлен, перейти к шагу 1.

3. Установить флаг.

4. Открыть файл.

5. Записать в файл сообщение.

6. Закрыть файл.

7. Сбросить флаг.

Вроде бы этот алгоритм должен работать, однако представим, что Windows прервет выполнение программы сразу же после выполнения шага 1, но перед выполнением шага 2. Допустим, первая программа читает значение флага, кото­рый находится в сброшенном состоянии, но не успевает проверить это значение. В этот момент управление передается второй программе, которая также читает значение флага и, убедившись в том, что флаг сброшен, устанавливает его и от­крывает файл для записи, Когда управление возвращается первой программе, она проверяет состояние флага, используя его старое значение, которое при этом уже не является соответствующим действительности. Таким образом, первая програм­ма устанавливает уже установленный флаг и пытается открыть файл, уже откры­тый другой программой. Для многопроцессорных систем эта проблема еще бо­лее актуальна, так как разные потоки могут одновременно выполняться на разных процессорах.

Чтобы решить проблему, нужно каким-либо образом блокировать доступ к флагу других программ, в то время как одна из программ читает и тестирует его значение. Windows позволяет добиться этого несколькими разными способами. В частности, вы можете воспользоваться вызовом **InterlockedIncrement**. Однако это достаточно низкоуровневый вызов, и вам вряд ли придется часто его использо­вать. Интерфейс Win32 обладает более удобными средствами решения подобных проблем.

Чаще всего для обработки подобных ситуаций используется механизм под названием *мьютекс* (mutex, сокращение от *mutual exclusion —* взаимное исклю­чение). Поток может получить мьютекс в собственное владение. При этом другие потоки не смогут завладеть мьютексом до тех пор, пока первый поток не освобо­дит его. Таким образом, в любой момент времени мьютекс может принадлежать только одному потоку, что исключает конфликты между потоками.

Вариант 3

Представьте клиентскую программу, обращающуюся к серверу базы данных че­рез сеть. Чтобы ограничить нагрузку на сервер, создаваемую каждой рабочей станцией, вы приняли решение ограничить количество запросов, издаваемых каж­дой рабочей станцией. Было решено, что одна рабочая станция не может адресо­вать серверу базы данных более трех запросов одновременно. Каждый запрос инициируется отдельным потоком, поэтому использовать обычный счетчик нельзя

Для решения проблемы можно создать три мыотекса. Поток, желающий об­ратиться к серверу, обязан завладеть одним из мыотексов. Если ни одного сво­бодного мыотекса не осталось, поток должен подождать. Более удобный cпocoб подразумевает использование семафоров, *Семафор —* это объект синхронизации содержащий в себе счетчик (с начальным значением, например, три). Каждый раз. когда поток использует семафор, значение счетчика уменьшается на единицу. Когда значение счетчика становится равным нулю, семафор становится недоступ­ным для каких-либо потоков.

Таким образом, семафор позволяет с легкостью ограничить сетевой трафик, создаваемый одной рабочей станцией. Прежде чем обратиться к серверу, поток обязан проанализировать значение счетчика. Если счетчик сема4юра сравнялся с нулем, поток обязан подождать.

Еще немного о синхронизации

Как видно из приведенных примеров, обеспечение совместной работы несколь­ких потоков — важная задача, которую приходится решать при разработке многих программ. В любой ситуации для решения проблемы вы можете воспользо­ваться несколькими способами, предусматривающими использование разных механизмов Windows.

Работая в многозадачной рабочей среде, не стоит забывать о возможности воз­никновения взаимной блокировки потоков *(deadlock).* Представьте, что один по­ток собирается открыть файл А, а затем — файл В. В то же время другой поток намерен открыть файл В, а затем — файл А. Предположим, что первый поток только что открыл файл А, но при попытке открыть файл В он обнаруживает, что этот файл уже открыт вторым потоком. Первый поток переходит в режим ожи­дания, надеясь на то, что в скором времени файл В освободится. Однако в то же время второй поток, только что открывший файл В, обнаруживает, что файл А уже открыт первым потоком. Второй поток надеется, что в скором времени файл А освободится, и также переходит в режим ожидания. Таким образом, оба потока начинают ждать друг друга, и их выполнение останавливается.

Если ни один из двух потоков не освободит занятые им ресурсы (в нашем случае — файлы), оба они «подвиснут» и не смогут продолжить работу. Это и называется взаимной блокировкой потоков (deadlock). Существует множество способов решения проблемы. Например, можно организовать работу потока та­ким образом, чтобы он освобождал все занятые им ресурсы в случае, если он не может получить доступ к одному из необходимых ресурсов. Позже поток может повторить попытку получить доступ ко всем необходимым ресурсам. Для реше­ния проблемы можно использовать мьютекс. В нашем достаточно простом при­мере можно организовать работу потоков таким образом, чтобы каждый из них прежде всего открывал файл А и лишь затем пытался открыть файл В. К сожа­лению, организовать работу программ именно таким образом удается далеко не всегда.

Проблема взаимной блокировки потоков не нова. Дискуссии на эту тему можно обнаружить в документах, относящихся не только к Windows, но также и ко мно­гим другим операционным системам. Таким образом, решение проблемы можно позаимствовать из многих других источников, например из мира Unix.

Подробнее о синхронизации

Основные объекты, предназначенные для синхронизации (события, мьютексы и семафоры), во многом схожи.

Чтобы создать тот или иной объект синхронизации, поток чаще всего обра­щается к специальной функции (например, вызов **CreateEvent** создает событие). Если вы работаете с потоками одного процесса, вы можете не именовать объект синхронизации. Если же вы намерены обеспечить доступ к объекту из несколь­ких разных процессов, вы должны присвоить этому объекту некоторое имя. При подборе имени необходимо учитывать регистр символов, а также нельзя исполь­зовать символ обратной косой (\). Имена объектов синхронизации разных типов принадлежат к одному пространству имен. Это значит, что вы не можете назвать событие именем **Shabash**, а позже присвоить это же имя семафору. Событие и се­мафор должны обладать отличающимися именами. Если вы не присвоили объекту имя, другие процессы смогут обратиться к этому объекту только при помощи наследования.

Вызов, создающий объект, возвращает дескриптор этого объекта. Потоки, при­надлежащие одному процессу, могут использовать один и тот же дескриптор.

Другие процессы могут наследовать дескриптор, однако чаще всего это не совсем удобно. Дело в том, что в этой ситуации вы все равно должны передать дескрип­тор дочернему процессу через командную строку, переменную окружения или любой другой подобный механизм. В противном случае новый процесс не будет знать, какой именно дескриптор соответствует объекту. Вместо того чтобы пере­давать дескриптор по наследству, можно воспользоваться вызовом открытия объекта. Например, чтобы открыть событие, следует обратиться к **OpenEvent** и пе­редать этой функции в качестве параметра имя события, присвоенное ему при создании. Этот вызов возвращает дескриптор объекта, который можно использо­вать для взаимодействия с объектом.

Иногда нельзя сказать заранее, какому из потоков в первую очередь потребу­ется создать объект. В этом случае можно обратиться к функции создания объекта из обоих потоков. Если поток пытается создать объект, который уже существует, вызов создания объекта сработает как вызов открытия объекта. Если требуется точно знать, существует объект или нет, воспользуйтесь вызовом открытия и проверьте, не указывает ли его возвращаемое значение на ошибку отсутствия объекта.

Если у вас уже есть дескриптор (полученный либо при помощи вызова созда­ния, либо при помощи вызова открытия, либо при помощи наследования), вы можете определить, находится ли объект в *сигнальном состоянии* (signaled state). Для каждого типа объектов термин «сигнальное состояние» имеет разный смысл. Событие находится в сигнальном состоянии в случае, если соответствующий флаг установлен. Мьютекс находится в сигнальном состоянии в случае, если он никому не принадлежит (установив это, ваш поток может стать владельцем мьютекса). Семафор находится в сигнальном состоянии, если его счетчик боль­ше нуля.

Чтобы определить состояние объекта, вы можете воспользоваться вызовом **WaitForSingleObject**. Обращаясь к этой функции, вы можете указать, какой имен­но период времени следует ожидать перехода объекта в сигнальное состояние. Время указывается в миллисекундах. Если время равно нулю, функция просто проверит текущее состояние объекта и немедленно вернет управление вызвавшей программе. Если указать интервал времени, равный константе INFINITE, функция будет ожидать перехода объекта в сигнальное состояние бесконечно долго.

Если поток ожидает перехода объекта в сигнальное состояние, он фактически не требует для своего функционирования дополнительного процессорного вре­мени. Таким образом, перевод потока в режим ожидания объекта синхронизации весьма эффективен с точки зрения производительности всей системы. В некото­рых случаях требуется, чтобы поток следил за состоянием сразу нескольких объек­тов. Для этой цели используется вызов **WaitForMultipleObjects**. Позже мы позна­комимся с некоторыми другими вызовами, позволяющими ожидать изменения состояния объекта.

Вызов **MaitForSingleObject** возвращает несколько различных значений в зави­симости от результатов проверки состояния объекта синхронизации. Если объект перешел в сигнальное состояние, функция возвращает значение WAIT\_OBJECT\_0. Если указанный при вызове период времени истек, а объект все еще не перешел в сиг­нальное состояние, функция возвращает WAIT\_TIMEOUT. Наконец, если процесс, вла­деющий мьютексом, завершает работу с открытым мьютексом, система передает этот мьютекс ожидающему потоку, а вызов возвращает значение WAIT\_ABANDONED.

Если мьютекс больше не нужен, вы можете освободить его при помощи вызо­ва **ReleaseMutex**. Чтобы освободить занятый вами ранее семафор, используйте вызов **ReleaseSemaphore**. При этом счетчик семафора будет увеличен на единицу. Обра­тите внимание, что эти вызовы — не то же самое, что **CloseHandle**. Вызовы **ReleaseMutex** и **ReleaseSemaphore** просто освобождают мьютекс и увеличивают зна­чение счетчика семаформа, в то время как вызов **CloseHandle** полностью закрыва­ет объект для доступа. Вызов **CloseHandle** можно использовать не только для закрытия объектов синхронизации, но также для закрытия любых других деск­рипторов, которыми владеет процесс.

Блокированные вызовы

Если вы хотите обратиться к переменной без опасений, что в момент обращения операционная система передаст управление другому потоку, вы можете восполь­зоваться одним из *блокированных вызовов* (interlocked call), которые являются частью Windows API. Блокированные вызовы перечислены в табл. 1. Вы може­те использовать эти вызовы для безопасной работы с переменными в многопо­точной среде, однако зачастую большинство встающих перед вами проблем, свя­занных с многопоточностью, удобнее решать при помощи событий, семафоров и мьютексов.

Блокированные вызовы Таблица 1.

|  |  |
| --- | --- |
| **Вызов** | **Назначение** |
| InterlockedDecrement Interlockedlncrement InterlockedExchange  InterlockedExchangeAdd InterlockedCompareExchange | Вычитает единицу из блокированной переменной  Добавляет единицу к блокированной переменной  Меняет местами значение блокированной переменной и значение другой переменной  Добавляет значение к блокированной переменной  Сравнивает значение блокированной переменной с некоторым значением и в случае, если значения равны, меняет местами значение блокированной переменной и значение некоторой другой переменной |

Блокированные вызовы из табл. 1 работают с 32-битными значениями, вы­ровненными по границе 32 бит. Если вы намерены синхронизировать работу по­токов, вы можете использовать обычную глобальную переменную. Если синхро­низируется работа процессов, переменную необходимо поместить в общую память.

Наверное, самым полезным является вызов **InterlockedCompareExchange**. Блоки­рованная переменная сравнивается с некоторым значением. Если переменная и значение равны, функция меняет местами значения блокированной переменной и любой другой переменной по вашему выбору. При этом вы можете быть уве­ренными в том, что в течение всей этой процедуры ни один другой поток не смо­жет получить доступ к блокированной переменной.

Например, если вы используете флаг tflag и хотите присвоить ему значение 1 только в случае, если этот флаг равен 0, вы можете написать следующий код:

…

PVOID var=(PVOID)&tflag;

PVOID newval=(PVOID)l;

PVOID compare=( PVOID)0;

InterlockedCompareExchange(&var, newval, compare);

if (newval=1) . . . ; // флаг был равен 1

else {

//флаг был равен 0, а сейчас он равен 1

. . .

}

Подробнее о событиях

Вызовы, предназначенные для работы с событиями, перечислены в табл. 2. Со­бытия могут быть мануальными (manual) и единичными (single). Тип события указывается при его создании. *Мануалъное событие (manual event) —* это не про­сто общий флаг для нескольких потоков. Оно выполняет несколько более слож­ные функции. Любой поток может установить это событие при помощи вызова **SetEvent** или сбросить (очистить) его при помощи вызова **ResetEvent**. Если собы­тие установлено, оно останется в этом состоянии сколь угодно долгое время, вне зависимости от того, сколько потоков ожидают установки этого события. Если вы обратитесь к функции **PulseEvent**, все потоки, ожидающие этого события, по­лучат сообщение о том, что событие произошло. После этого событие автомати­чески сбросится. Другими словами, любой поток, обратившийся к функции **UaitForSingleObject** и в связи с этим находящийся в состоянии ожидания, полу­чит возможность продолжить работу. Когда все потоки, ожидающие события, продолжат работу, система автоматически сбросит событие.

Такая логика работы приемлема далеко не всегда. Возможно, что для работы вашей программы требуется, чтобы в случае возникновения события на него реа­гировал только один из потоков, в то время как все остальные потоки продолжа­ли ждать. Для этой цели можно использовать единичные события. Если при по­мощи **SetEvent** вы установите *единичное событие* (single event), только один ожидающий поток будет оповещен об этом событии и, соответственно, сможет продолжить работу. После этого система автоматически сбросит событие. Если в момент установки события не существует ни одного ожидающего потока, собы­тие останется в сигнальном положении до тех пор, пока в системе не появится какой-либо ожидающий это событие поток. В отличие от **SetEvent** при использо­вании **PulseEvent**, если в системе нет ожидающих событие потоков, событие бу­дет сброшено немедленно.

В каких ситуациях лучше то или иное событие? Все зависит от того, какие цели вы перед собой ставите. В начале главы был приведен пример программы, использующей несколько потоков для анализа исходных данных с использова­нием разных алгоритмов (вариант 1). Очевидно, что в этом случае удобнее ис­пользовать мануальное событие. Основная программа подготавливает исходные данные, а затем устанавливает мануальное событие, среагировав на которое, все дочерние потоки одновременно приступают к обработке этих данных.

Рассмотрим программу, которая умножает большое количество огромных мат­риц (возможно, для сетевого анализа, имитации аэродинамической трубы или обработки изображений). Порядок, в котором будут умножаться эти матрицы, не важен. Основная программа создает несколько потоков, каждый из которых чи­тает матрицу с жесткого диска. Когда первая матрица готова к обработке, поток устанавливает событие. Прежде чем приступать к вычислениям, каждый после-

дующий поток ждет, пока будет установлено событие. После завершения умно­жения каждый из потоков вновь устанавливает событие. В данной ситуации удоб­нее использовать единичное событие.

Вызовы работы с событиямиТаблица 2.

|  |  |
| --- | --- |
| **Вызов** | **Назначение** |
| CreateEvent OpenEvent SetEvent ResetEvent PulseEvent | Создает событие или открывает уже существующее событие  Открывает существующее событие  Устанавливает событие (переводит его в сигнальное состояние)  Сбрасывает событие  Переводит событие в сигнальное состояние на период времени, пока на это событие не прореагируют все ожидающие его потоки; после чего событие сбрасывается |

Пример простой тестовой программы, иллюстрирующий работу с единичны­ми событиями, приведен в листинге 1. Запустите программу без аргументов. Программа перейдет в режим ожидания события. После этого вновь запустите программу, но при этом передайте ей какой-либо аргумент (все равно какой). Программа установит событие, на которое прореагирует предыдущая запущен­ная вами копия программы, которая выйдет из режима ожидания события и за­вершит работу.

Листинг 1.

#include <windows.h>

#include <iostream.h>

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

{

// создаем или открываем имеющееся

HANDLE ev=CreateEvent(NULL, FALSE, FALSE, "nt5bbevent");

if (argc==1)

{

cout<<"Подождите событие!\n";

cout.flush();

WaitForSingleObject(ev, INFINITE);

cout<<"Событие произошло!\n";

cout.flush();

}

else

{

cout<<"Установка события\n";

SetEvent(ev);

}

CloseHandle(ev);

}

Событие, устанавливаемое программой, является единичным событием. Что­бы убедиться в этом, вы можете запустить несколько копий программы без аргу­ментов, а затем запустить программу с аргументом. Обратите внимание, что при каждом последующем запуске программы с аргументом завершать работу будет только одна из запущенных ранее копий программы.

Подробнее о мьютексах

Если поток является владельцем мьютекса, он обладает правом эксклюзивного использования ресурса, который защищается этим мьютексом. Ни один другой поток не может завладеть мьютексом, который уже принадлежит одному из по­токов. Вместе с тем поток, являющийся владельцем мьютекса, может попытать­ся стать владельцем мьютекса повторно. Если поток присваивал себе мьютекс несколько раз, он обязан освободить его такое же количество раз. Вызовы, пред­назначенные для работы с мьютексами, перечислены в табл. 3.

Вызовы для работы с мьютексамиТаблица 3.

|  |  |
| --- | --- |
| **Вызов** | **Предназначение** |
| CreateMutex OpenMutex ReleaseMutex | Создает новый мьютекс или открывает уже существующий  Открывает существующий мьютекс  Освобождает мьютекс и делает его доступным для других потоков |

Возможность одного потока несколько раз подряд завладеть одним и тем же мьютексом с первого взгляда может показаться нелогичной, однако, немного по­размыслив, можно прийти к выводу, что подобное поведение мьютексов вполне оправдано. Представим, что программа использует мьютекс для защиты файла журнала сообщений об ошибках. Функция Е1 предназначена для записи сообще­ния в этот файл. Эта функция получает мьютекс, записывает сообщение в файл, используя при этом уже открытый файловый дескриптор, после чего освобожда­ет мьютекс. Предположим, что функция Е2 тоже записывает в файл сообщение об ошибке, однако в процессе выполнения эта функция обращается к Е1. Так как Е2 выполняет часть работы самостоятельно, а часть работы поручает выполнить Е1, функция Е2 также обязана получить в собственное владение мьютекс. В конце своей работы функция Е2 обязана этот мьютекс освободить.

Предположим, поток А обращается к Е2, а чуть позже поток В обращается к Е1. В этой ситуации поток В обязан подождать, пока мьютекс освободится. В соот­ветствии с логикой вещей так и должно быть. Что происходит при выполнении потока А? Функция Е2 открывает мьютекс, выполняет некоторые действия, пос­ле чего обращается к функции Е1. Если бы поток не обладал возможностью по­вторно завладеть мьютексом, он не смог бы продолжить работу. Ведь функция Е1 попыталась бы завладеть мьютексом, который уже принадлежит потоку (поток А завладел мьютексом в процессе выполнения функции Е2).

К счастью, Windows обрабатывает подобные ситуации корректно. Когда Е1 от имени потока А пытается завладеть мьютексом, операционная система обнару­живает, что мьютекс уже принадлежит этому потоку, поэтому функции Е1 разре­шается завладеть мьютексом повторно. Когда Е1 освобождает мьютекс, Windows понимает, что функция Е2 все еще нуждается в мьютексе, поэтому право потока А на владение мьютексом сохраняется до тех пор, пока функция Е2 не освободит мьютекс.

Вы должны понимать, что мыотексы — это все равно что общественные дорожные инспекторы, которые могут только указать на нарушение правил, но не имеют права оштрафовать или привлечь к ответственности. Если мьютекс защищает файл, он сможет выполнить свою функцию только в случае, если перед обращением к файлу каждый из ваших потоков будет проверять состояние этого мьютекса.

Операционная система никак не связывает мьютексы с какими-либо ресурсами. Windows расценивает мьютекс как объект общего доступа, который можно пере­вести в сигнальное состояние или сбросить. Потоки должны самостоятельно ана­лизировать текущее состояние мьютексов и продолжать функционирование в соответствии с тем, для чего предназначен тот или иной мьютекс.

Чаще всего мьютексы используются для синхронизации процессов, однако в некоторых случаях мьютексы используют для обеспечения синхронной работы нескольких потоков в рамках одного процесса. Конечно, для синхронизации по­токов допускается использовать мьютексы, однако более высокой производитель­ности можно достичь при использовании *критических секций* (critical section), Критическая секция — это форма мьютекса, предназначенная для использования в рамках одного процесса. Помимо этого критическая секция обладает рядом других ограничений, о которых будет рассказано несколько позже в данной главе,

Листинг 2 содержит исходный код простой программы, демонстрирующей применение основных функций работы с мьютексами. Запустите программу. Программа получает в собственное владение мьютекс и отображает на экране простое окно сообщения (message box). He убирайте это окно с экрана. Вместо этого запустите вторую копию программы. Вторая копия не сможет завладеть мьютексом до тех пор, пока первая копия не завершит работу. Другими словами, каждая новая копия программы будет ждать, пока вы не щелкнете по кнопке диалогового окна, тем самым убрав это окно с экрана и завершив работу програм­мы. Если вы запустили несколько копий программы, в любой момент времени только одна из них сможет вывести на экран окно сообщения.

**Листинг 2.** Тестирование работы с мьютексом

#include <windows.h>

#include <iostream.h>

void main()

{

HANDLE mtx=CreateMutex(NULL, FALSE, "NT5BBMTX");

cout<<"Проверить мьютекс\n";

cout.flush();

WaitForSingleObject(mtx, INFINITE);

cout<<"Мьютекс получен!\n";

cout.flush();

MessageBox(NULL,"Программа захватила мьютекс", "MUTEX", MB\_OK);

ReleaseMutex(mtx);

CloseHandle(mtx);

}

Подробнее о семафорах

Если вы понимаете, как работают события и мьютексы, изучить работу семафо­ров не составит для вас особого труда. При создании семафора необходимо ука­зать начальное значение счетчика. Вызовы, предназначенные для работы с сема­форами, перечислены в табл. 4. Семафоры работают очень просто. Каждый раз, когда программа обращается к семафору, значение счетчика семафора уменьша­ется на единицу. Когда значение счетчика становится равным нулю, семафор ста­новится недоступным.

Таблица 4

|  |  |
| --- | --- |
| **Вызов** | **Назначение** |
| CreateSemaphore OpenSemaphore ReleaseSemaphore | Создает новый семафор или открывает существующий  Открывает существующий семафор  Добавляет некоторое значение (обычно 1) к счетчику семафора, делая его доступным для большего количества потоков |

Если поток обращается к семафору дважды, значение счетчика семафора умень­шается на две единицы. В этом семафоры отличаются от мьютексов, так как если поток несколько раз обращается к мьютексу, это расценивается как одно oбpaщение.

Листинг 3 содержит простую программу, демонстрирующую работу семафора. Прежде чем отобразить на экране окно сообщения, программа пытается обратиться к семафору. Вы можете повторно запустить программу. Когда на экране появля­ются три окна сообщения, четвертое и все последующие обращения к семафору блокируются. Таким образом, вне зависимости от того, сколько программ ожи­дают освобождения семафора, на экране будет отображено не более трех окон сообщений. Как только вы закроете одно из окон, соответствующая программа освободит семафор и завершит работу. В этот же момент одна из программ, ожи­дающих освобождения семафора, получит к нему доступ и выведет на экран окно сообщения.

**Листинг 3.** Демонстрация работы семафора

#include <windows.h>

#include <iostream.h>

void main()

{

HANDLE sem=CreateSemaphore(NULL, 3, 3, "nt5bbsem");

cout<<"Проверить состояние семафора\n";

cout.flush();

WaitForSingleObject(sem, INFINITE);

cout<<"Получен доступ к семафору\n";

cout.flush();

MessageBox(NULL, "Получен доступ к семафору", "Semaphore", MB\_OK);

ReleaseSemaphore(sem, 1, NULL);

CloseHandle(sem);

}

Безопасная синхронизация

С синхронизацией связана одна серьезная проблема, которая может стать при­чиной нарушения безопасности вашей системы. Предположим, вы разрабатыва­ете программу для управления пусковой установкой ядерной ракеты. Пусковую установку обслуживает несколько процессов: один осуществляет обслуживание ракеты, другой настраивает навигационное оборудование, третий выполняет сбор данных и выдает доклад о состоянии, а четвертый предназначен для управления запуском. В любой момент времени с пусковой установкой работает только один процесс. Чтобы реализовать раздельный доступ разных процессов к установке, вы используете мьютекс. Каждый процесс владеет мьютексом ровно столько,

сколько требуется для выполнения его функций. Программа спроектирована та­ким образом, чтобы мьютекс не оставался занятым слишком долгое время.

Код каждого процесса содержит обращение к функции создания мьютекса. На практике мьютекс создается первым запущенным процессом. Остальные процес­сы открывают уже существующий мьютекс и анализируют его состояние. Пред­положим, что вражеский агент узнал имя вашего мьютекса и написал програм­му, которая открывает мьютекс, получает его в собственное пользование и держит его занятым вечно. Пока вы не поймете, что произошло, и не уничтожите вра­жеский процесс, ни один из ваших процессов не сможет получить доступ к пус­ковой установке.

Проблему можно решить, избежав именование мьютекса. Для этого основная программа может создать безымянный мьютекс, а затем запустить дочерние процессы и приказать каждому из них унаследовать дескриптор открытого мьютекса. Чтобы передать новым процессам значение дескриптора мьютекса, основная! программа может использовать командную строку.

В результате каждый из дочерних процессов сможет обратиться к мьютексу, однако даже если вражеская программа узнает значение дескриптора, она не сможет его использовать. Конечно же, защитить дескриптор можно при помощи других способов (например, при помощи стандартных механизмов безопасности) Windows. Вместе с тем описанный способ прост и при правильном применении весьма эффективен.

Использование вызова WaitForMultipleObjects

В некоторых ситуациях поток должен ждать перехода в сигнальное состояние нескольких объектов синхронизации. Для этого удобно использовать вызов **WaitForMultipleObjects**. В качестве одного из аргументов этот вызов принимает массив дескрипторов объектов синхронизации. Если любой из этих объектов переходит в сигнальное состояние, вызов возвращает вам информацию о том, какой именно объект изменил свое состояние. Как и при использовании других подобных вызовов, вы можете указать период времени, в течение которого следует ожидать изменение состояния объектов. Период времени может быть любым, включая 0 и INFINITE (неограниченное время).

Функция **WaitForMultipleObjects** возвращает управление вызвавшему потоку **в** случае, если хотя бы один из ожидаемых объектов изменил свое состояние. Что| делать, если вы хотите, чтобы управление было возвращено только в случае, если абсолютно все объекты изменили свое состояние? Для этого необходимо присвоить аргументу bWaitAll этой функции значение TRUE.

Еще одним вызовом, позволяющим ожидать изменения состояния нескольких объектов, является вызов **MsgWaitForMultipleObjectsEx**. Этот вызов позволяет ждать не только изменения состояния объектов, но также передачи программе каких-либо системных сообщений. В частности, программа может ожидать изменения состояния объекта и в то же время реагировать на системные сообщения, такие как, например, WM\_PAINT.

Вызов **MsgWaitForMultipleObjectsEx** можно использовать также для ожидания одновременно и объектов, и сообщений. Другими словами, функция вернет управление потоку только в случае, если все указанные объекты перейдут в сигнальное состояние и при этом будут приняты все указанные сообщения. Конечно, в этой ситуации вы не сможете обрабатывать системные сообщения до тех пор, пока все интересующие объекты не перейдут в сигнальное состояние. Это обстоятель­ство существенно снижает полезность подобного вызова.

При переходе в сигнальное состояние одного из ожидаемых объектов вызов **WaitForMultipleObjects** возвращает значение из заданного диапазона, указывающее на то, какой именно объект изменил состояние. Если поток ожидал изменения состояния нескольких объектов, значение, возвращенное функцией **MaitForMultipleObjects**, непредсказуемо, однако оно все равно принадлежит задан­ному диапазону. Диапазон включает в себя числа от WAIT\_OBJECT\_0 до WAIT\_OBJECT\_0 плюс количество элементов в массиве дескрипторов объектов синхронизации. Зна­чение WAIT ABANDON\_0 (совпадает с WAIT\_ABANDON) является нижней границей диа­пазона возвращаемых значений для мьютекса, использование которого было пре­рвано. И наконец, значение WAIT\_TIMEOUT обозначает, что истекло время, в течение которого функция **WaitForMultipleObjects** должна была ожидать изменения состо­яния объектов.

Ожидание объектов в настороженном состоянии

Для ожидания изменения состояния объектов можно использовать три других вызова: **WaitForSingleObjectEx, MsgWaitForMultipleObjectsEx** и **WaitForMultipleObjectsEx**. В отличие аналогов, в именах которых отсутствует суффикс Ех, эти вызовы по­зволяют ожидать изменения состояния объектов в так называемом «насторожен­ном» (alertable) состоянии.

Что такое настороженное состояние? Если поток находится в состоянии на­стороженного ожидания, он может выполнять асинхронные вызовы процедур (Asynchronous Procedure Call, APC). Чаще всего функции АРС используются для организации асинхронного ввода/вывода. Ваш поток может при­казать выполнить функцию АРС в другом, уже существующем и работающем потоке, асинхронно для этого потока. Для этого ваш поток может использовать вызов **QueueUserAPC**. Однако для этого другой поток должен находиться в «на­стороженном» состоянии. Если другой поток не находится в «настороженном» состоянии, функция АРС ставится в очередь на исполнение в указанном потоке. Исполнение всех функций, стоящих в очереди для некоторого потока, проис­ходит в момент, когда поток переходит в «настороженное» состояние. Перевес­ти поток в «настороженное» состояние можно либо при помощи упомянутых выше трех вызовов (**WaitForSingleObjectEx, MsgWaitForMultipleObjectsEx** и **WaitForMultipleObjectsEx**), либо при помощи вызова **SleepEx**.

Таймер синхронизации

Еще одним способом вызова функции АРС является *таймер синхронизации* (waitable timer). Для создания нового или открытия уже существующего тай­мера синхронизации можно использовать функции **CreateWaitableTimer** и **OpenWaitableTimer**, которые работают в точности так же, как работают аналогич­ные функции, предназначенные для создания и открытия других объектов синх­ронизации. Таймер переходит в сигнальное состояние в момент, когда истекает соответствующий ему временной интервал. Таймер синхронизации можно настроить таким образом, что в момент истечения его периода времени происхо­дит вызов функции АРС в некотором потоке. Как и в других подобных случаях, вызов АРС не сработает до тех пор, пока целевой поток не перейдет в насторо­женное состояние.

Это может показаться странным, однако поток, который ожидает срабатыва­ния таймера, не может быть предназначенным для выполнения связанной с тай­мером функции АРС. Дело в том, что когда таймер срабатывает, поток выходит из состояния ожидания и поэтому не может быть использован для выполнения функции АРС.

Вызовы **CreateWaitableTimer** и **OpenWaitableTimer** возвращают дескриптор таймера. Чтобы начать отсчет времени таймером, необходимо обратиться к функции **SetWaitableTimer**. Период времени указывается как значение типа LARGE\_INTEGER, рав­ное количеству единиц времени, каждая из которых равна 100 наносекундам. На самом деле используемое аппаратное обеспечение может не поддерживать изме­рение времени с точностью до 100 наносекунд. Если указан положительный пе­риод времени, считается, что это абсолютное время. Отрицательные значения обозначают относительное время. Можно настроить таймер таким образом, что­бы он сработал только один раз, а можно приказать ему срабатывать через рав­ные промежутки времени.

Когда период времени истекает, таймер переходит в сигнальное состояние. Можно настроить таймер таким образом, чтобы при переходе его в сигнальное состояние автоматически выполнялся вызов функции АРС. Остановить таймер можно при помощи вызова **Cance1Waitab1eTimer**.

Критические секции

Обычно, если требуется предотвратить одновременный доступ к ресурсу несколь­ких программам, вы используете мьютекс. Можно ли блокировать одновремен­ное обращение к некоторой функции нескольких потоков одного процесса? Как сделать так, чтобы в любой момент времени к функции имел право обратиться только один поток? Конечно же, для этой цели также можно использовать мью­текс. Спроектируйте программу так, чтобы перед обращением к функции каж­дый поток пытался завладеть мьютексом.

Это решит проблему, однако использование мьютекса — не самое эффектив­ное решение в данной ситуации. Обращение к мьютексу снижает производитель­ность программы. Это связано с тем, что мьютекс предназначен для обслужива­ния нескольких разных процессов. Если вы намерены обеспечить синхронную работу нескольких потоков одного и того же процесса, вы можете воспользоваться другим, более простым и эффективным решением. Для обработки подобных си­туаций Windows предлагает использовать *критические секции* (critical sections) (см. табл. 5). Критическая секция анализирует значение специальной перемен­ной процесса, которая используется как флаг, предотвращающий исполнение некоторого участка кода несколькими потоками одновременно.

Чтобы защитить участок кода от совместного использования несколькими потоками, определите переменную типа CRITICAL\_SECTION и передайте ее адрес функции **InitializeCnticalSection**. К этой функции следует обращаться из глав­ного потока вашей программы (обычно из потока main). В начале критической секции кода следует расположить обращение к функции **EnterCriticalSection**. Если в этот момент критическая секция исполняется каким-либо другим потоком, ис­полнение потока, обратившегося к **EnterCnticalSection**, будет блокировано до тех пор, пока критический участок не освободится. Если вы не хотите блокировать работу потока и желаете, чтобы во время ожидания освобождения критической секции поток выполнял какую-либо другую полезную работу, вы можете исполь­зовать вызов **TryEnterCriticalSection**. Этот вызов возвращает значение TRUE в случае, если критический участок кода никем не занят. Если вызов вернул значение FALSE, поток может выполнить какую-либо другую работу, а затем вновь обратиться к **TryEnterCriticalSection**.

Вызов **TryEnterCricalSecion** будет продолжать возвращать FALSE, а вызов **EnterCnticalSelecton** будет продолжать блокировать выполнение потоков до тех пор, пока поток, выполняющий критическую секцию, не обратится к **LeaveCriticalSecion**. Таким образом, чаще всего функция, содержащая критичес­кий участок кода, выглядит следующим образом:

void one\_at\_a\_time\_please()

( EnterCritica1Section(&section): // инициализация выполняется

// в другом месте

// здесь расположен критический участок кода

LeaveCritica1Section(&section):

}

Другой вариант аналогичной функции:

BOOL only\_one\_at\_a\_time()

{ if(!TryEnterCriticalSection(&section)) // инициализация выполняется

// в другом месте

return FALSE: // функция занята другим потоком

// здесь расположен критический участок кода

LeaveCnticalSection(&section);

return TRUE: // мы сделали это!

}

Если вы больше не нуждаетесь в критической секции, вы обратитесь к **DeleteCnticalSection**, чтобы уничтожить ее. При этом система освободит ресур­сы, связанные с критической секцией.

В пакете добавлений NT 4.0 Service Pack 3 впервые появился еще один вы­зов, предназначенный для инициализации критической секции. Этот вызов под названием **InitializeCriticalSectionAndSpinCount** работает в точности так же, как и **InitializeCriticalSection**, если речь идет об однопроцессорной системе. Но если компьютер оснащен несколькими процессорами, новый системный вызов обла­дает некоторыми преимуществами. Обнаружив, что критическая секция уже заня­та другим потоком, ваш поток ждет некоторое небольшое количество времени, определяемое специальным счетчиком *(spin count).* Если после этого критичес­кая секция все еще остается занятой, поток переходит в другой режим ожидания, напоминающий **MaitForSingleObject**. Значение специального счетчика можно установить для каждой критической секции при помощи вызова **SetCriticalSectionSpinCount**. He забывайте, что на однопроцессорных компьютерах эти вызовы не будут иметь эффекта.

Вызовы, связанные с критической секцией Таблица 5

|  |  |
| --- | --- |
| Вызов | Назначение |
| InitializeCriticalSection InitializeCriticalSectionAndSpinCount  EnterCriticalSection  TryEnterCriticalSection  LeaveCriticalSection  SetCriticalSectionSpinCount  DeleteCriticalSection | Инициализирует переменную типа CRITICALSECTION Инициализирует переменную типа CRITICALSECTION и счетчик ее ожидания (см. текст)  Входит в критическую секцию, блокирует остальные потоки, пытающиеся войти в критическую секцию  Пытается войти в критическую секцию, возвращает ошибку в случае, если критическая секция уже исполняется каким-либо потоком  Покидает критическую секцию, разрешая другим потокам войти в нее  Устанавливает значение счетчика ожидания критической секции (spin count)  Уничтожает переменную CRITICALSECTION |

Другие объекты синхронизации

Помимо упомянутых существуют также другие типы объектов, за состоянием которых можно следить при помощи вызова **MaitForSingleObject** и аналогичных функций. На самом деле объекты, о которых пойдет речь, не являются объекта­ми синхронизации в чистом виде, однако не упомянуть о них в рамках этой кни­ги нельзя. На мой взгляд, лучше всего рассказать об этих объектах именно в этой главе. Вспомним, например, о вызовах, которые использовались в лабораторной работе про ПРОЦЕССЫ, ПОТОКИ и НИТИ для определения момента завершения процесса или потока. В отношении этих вызовов можно сказать, что в момент завершения процесса или потока дескриптор этого процесса или потока переходит в сигнальное состояние. Дескриптор процесса или потока не является объектом синхронизации, однако за его состоя­нием можно следить точно так же, как вы следите за состоянием любого объекта синхронизации.

Оповещение об изменениях

*Оповещение об изменении (change notification) —* это специальный объект, который переходит в сигнальное состояние в случае, если содержимое дискового каталога изменяется. Чтобы создать оповещение об изменении, необходимо обратиться к функции FindFirstChangeNotification. В качестве параметров функция принимает имя каталога и два флага. Первый флаг указывает на то, что поиск изменений будет осуществляться не только в указанном каталоге, но и во всех его подката­логах. Второй флаг определяет набор событий, которым будет уделять внимание оповещение об изменениях. Например, можно следить за любым изменением файла, за изменением размера или времени записи любого файла и т. п.

Если происходит любое из указанных изменений, дескриптор переходит в сигнальное состояние. Переход дескриптора в сигнальное состояние можно об­наружить при помощи **MaitForSingleObject** или любого другого аналогичного вы­зова. Функция **FindNextChangeNotification** выводит дескриптор из сигнального состояния таким образом, чтобы можно было продолжить слежение за измене­ниями в каталоге. Если осуществлять слежение за изменениями больше не тре­буется, следует воспользоваться функцией **FindCloseChangeNotification** для того, чтобы уничтожить соответствующий дескриптор.

В листинге 4 приведен пример программы, которая осуществляет слежение за изменениями в каталоге C:\SPOOLMSG. Если вы запустите программу, а затем скопируете в этот каталог файл (или уничтожите или переименуете один из со­держащихся в этом каталоге файлов), программа выведет на экран окно сообще­ния, сообщающее об изменениях. Конечно, более сложная программа может оп­ределить имя файла (при помощи вызова FindFirst и других связанных с этим вызовов), отобразить его содержимое, после чего уничтожить его.

**Листинг 4.** Использование оповещения об изменениях

#include <windows.h>

void main()

{

HANDLE spool=FindFirstChangeNotification("c:\\spoolmsg", FALSE,

FILE\_NOTIFY\_CHANGE\_FILE\_NAME);

while(1) {

WaitForSingleObject(spool,INFINITE);

MessageBox(NULL,"Контрольный каталог был изменен",

"SPOOLMSG",MB\_OK);

FindNextChangeNotification(spool);

}

}

Дискриптор консоли

Дискриптор консоли переходит в сигнальное состояние в случае, если в стандартном потоке ввода находятся данные, которые необходимо прочитать. Стандартный дискриптор ввода можно получить несколькими способами. Например, можно обратиться к функции **CreateFile** и передать ей в качестве имени файла ключевое слово CONIN$ или вызвать функцию **GetStdHandlе**.

Поначалу слежение за состоянием стандартного дескриптора ввода может показаться бессмысленным. В конце концов, действия, связанные с ожиданием данных, поступающих через стандартный поток ввода, обычно выполняются стан­дартными функциями C++. Не спешите делать выводы. Слежение за состояни­ем потока ввода может потребоваться вам при организации перекрывающегося ввода/вывода, о котором речь в следующей лабораторной работе. Мы рассмотрим, каким образом можно ожидать поступления данных через, например, последовательный порт и при этом выполнять какие-либо другие действия.

Например, используя подобную технологию, можно организовать работу вашей программы таким образом, чтобы она реагировала на поступление управляющих команд как из последовательного порта, так и от консоли. Подобный подход лег­ко реализуется при помощи уже знакомой нам функции **WaitForMultipleObjects**. Конечно же, такую программу можно написать и по-другому, однако подобный метод является весьма эффективным и достаточно удобным.

Заключение

Если вы не используете преимуществ многопоточности и многозадачности, зна­чит, вы не используете преимуществ Windows. Освоить методы работы в много­поточной среде относительно несложно, а преимущества многопоточного подхо­да сложно переоценить. К тому же Windows предлагает обширный набор удобных механизмов, облегчающих взаимодействие потоков и процессов и существенно упрощающих разработку многопоточных приложений.

*Выбор метода синхронизации*

Зачастую реализовать тот или иной метод синхронизации значительно проще, чем определить, какой из механизмов синхронизации лучшим образом подходит для применения в Вашей программе. Основными типами объектов синхрониза­ции являются *события (events), мьютексы (mutexes)* и *семафоры (semaphores).* Каждый из этих объектов обладает своими особенностями, делающими его при­менение удобным в той или иной ситуации.

Событие напоминает общий для нескольких процессов флаг, однако действу­ет несколько сложнее, чем обычная переменная общего доступа. События удоб­но применять в случае, если один поток хочет оповестить другой поток о том, что он завершил ту или иную операцию и готов к дальнейшим действиям. *Мануалыюе событие* (manual event) остается в установленном состоянии до тех пор, пока ваша программа явно сбросит его. *Единичное событие* (single event) автома­тически сбрасывается в момент, когда хотя бы один из потоков обнаружит, что оно установлено. Вызов PulseEvent устанавливает событие на время, необходимое для того, чтобы об изменении состояния события узнали все ожидающие его по­токи, а затем автоматически сбрасывает событие.

Мьюотекс сигнализирует о том, что поток эксклюзивно владеет тем или иным ресурсом. В любой момент времени мьютексом владеет не более одного потока. Если ноток является хозяином мьютекса, он может повторно завладеть им лю­бое количество раз. Поток сохраняет за собой права на владение мьютексом до тех пор, пока он не обратится к **ReleaseMutex**. Если перед этим поток завладел мьютексом несколько раз, чтобы освободить мьютекс, он должен обратиться к **ReleaseMutex** такое же количество раз. Система будет отказывать всем остальным потокам, пытающимся завладеть мьютексом, до тех пор, пока поток-владелец мьютекса не иынолнит завершающее обращение к ReleaseMutex.

Если вы хотите использовать технологию наподобие мьютекса в рамках од­ного процесс;», вы можете использовать *критическую секцию.* Критическая сек­ция обеспечивает большую эффективность, чем мьютекс. Для своих функций критическая секция использует специальную переменную вашей программы (тип переменной — CRITICAL\_SECTION), которая выполняет функции дорожного полицей­ского, управляющего движением на узком участке дороги. Когда поток обраща­ется к **EnterCriticalSection**, он получает право выполнить критический участок кода. Если после этого другой поток попытается выполнить **EnterCriticalSection**, его выполнение будет блокировано до тех пор, пока первый поток не обратится к **LeaveCriticalSection**. Если поток, желающий выполнить критический участок кода, не желает, чтобы его выполнение было блокировано в случае, если этот участок уже занят другим потоком, он может вместо **EnterCritica1Secion** исполь­зовать вызов **TryCriticalSection**.

Помимо мьютексов и критических секций для ограничения доступа к ресур­сам можно использовать семафоры. В отличие от мьютексов и критических сек­ций, семафор разрешает одновременное использование ресурса для нескольких потоков. Однако при этом максимальное количество потоков, одновременно об­ладающих доступом к ресурсу, ограничивается, Каждый семафор обладает свя­занным с ним счетчиком. Каждый раз, когда поток получает доступ к семафору, значение счетчика уменьшается на единицу. Когда значение счетчика семафора становится равным нулю, семафор становится недоступным для остальных по­токов. Чтобы воспользоваться семафором, остальные потоки обязаны ждать до тех пор, пока один из потоков, обладающих доступом к семафору, не обратится к вызову **ReleaseSemaphore**. В результате этого вызова значение счетчика увеличит­ся на единицу, и семафором сможет воспользоваться какой-либо другой поток.

Когда следует использовать тот или иной метод синхронизации? Это зависит от того, какие цели преследует ваша программа. Если необходимо оповестить несколько потоков о том, что подготовка исходных данных завершена и потоки могут приступить к их обработке, скорее всего, удобнее воспользоваться собы­тием. Чтобы предотвратить одновременный доступ к файлу нескольких потоков, удобнее использовать мьютекс (или критическую секцию в случае, если речь идет о потоках, принадлежащих одному процессу). Чтобы ограничить количество се­тевых запросов, которые имеет право инициировать одна рабочая станция, удоб­нее использовать семафор.

За исключением критической секции, все объекты синхронизации работают очень похоже. Чтобы создать тот или иной объект синхронизации, следует обра­титься к соответствующему вызову создания объекта (например, **CreateMutex** или **CreateSemaphore**). Этот вызов либо создает новый объект синхронизации, либо открывает уже существующий, который обладает указанным вами именем. Если вы намерены открыть уже существующий объект синхронизации, вы можете ис­пользовать для этого специальный вызов (например, **OpenMutex** или **OpenSemaphore**). В любом случае вы получаете дескриптор объекта синхронизации. Если вы рас­считываете, что создаваемый вами объект будет использоваться только текущим процессом и его дочерними процессами, вы можете не указывать имя объекта при его создании. Дочерние процессы могут наследовать дескрипторы объектов син­хронизации, созданные родительским процессом.

В качестве первого атрибута вызову Create объекта синхронизации передает­ся указатель на структуру SECURITY\_ATTRIBUTE, в которой содержится информация о доступе к новому дескриптору. Структура содержит флаг, определяющий, могут ли дочерние процессы наследовать этот дескриптор, а также указатель на деск­риптор безопасности, который можно не указывать. Если вы хотите использовать дескриптор безопасности по умолчанию, вы можете в качестве указателя на него подставить в структуру значение NULL. Если вас также не волнует значение флага, определяющего режим наследования, в качестве первого аргумента вы можете передать функции создания объекта синхронизации значение NULL. В этом случае будет использоваться значение флага по умолчанию. Чтобы присвоить значение дескриптору безопасности, используйте функцию **ImtializeSecurityDescriptor** совместно с другими вызовами, предназначенными для управления уровнем доступа.

Если вы хотите, чтобы доступом к объекту синхронизации обладали взаимо­независимые процессы, вы должны присвоить этому объекту имя. Имя — это последовательность символов с учетом регистра. В состав имени не может вхо­дить символ обратной косой (\). Имена всех объектов синхронизации принадле­жат к единому пространству имен, поэтому нельзя присвоить одинаковое имя, например, мьютексу и семафору.

Следить за состоянием объектов синхронизации можно при помощи специ­альных вызовов, самый простой из которых — **WaitForSingleObject**. Эта функция проверяет, находится ли объект в *сигнальном состоянии.* Как только это произош­ло, функция немедленно возвращает управление вызвавшей программе и сооб­щает ей об изменении состояния объекта. Сигнальное состояние для разных объек­тов синхронизации обозначает разное. Событие переходит в сигнальное состояние в случае, если оно установлено. Мьютекс переходит в сигнальное состояние в случае, если он освободился. А семафор — в случае, если значение его счетчика отлично от нуля. Функция **WaitForSingleObject** может ожидать изменения состо­яния интересующего вас объекта сколь угодно долгое время, однако эту же функ­цию можно использовать для того, чтобы только проверить состояние объекта и немедленно продолжить работу основной программы.

Помимо **WaitForSingleObject** существуют также другие вызовы, которые мож­но использовать для слежения за состоянием объектов синхронизации. Напри­мер, вызов WaitForMu1t1p1eObjects позволяет следить за состоянием сразу несколь­ких объектов синхронизации. В зависимости от переданных ей аргументов эта функция вернет управление вызвавшей программе либо в момент, когда свое со­стояние изменят все интересующие вас объекты, либо когда в сигнальное со­стояние перейдет хотя бы один из них. Существуют также некоторые другие вызовы, которые работают подобно **WaitForSingleObject** и **WaitForMu1tip1eObjects** с некоторыми отличиями.

Изменить состояние события можно в любой момент при помощи специаль­ных вызовов. Мьютексы и семафоры изменяют состояние в зависимости от того, владеют ли ими какие-либо потоки. Освободить мьютекс можно при помощи вызова ReleaseMutex. Семафор освобождается при помощи вызова **ReleaseSemaphore**.

Предотвращение взаимного блокирования потоков

Разрабатывая приложения для многозадачной или многопоточной среды, вы мо­жете столкнуться с возможностью возникновения взаимного блокирования по­токов (по-английски это понятие обозначается терминами *deadlock —* тупик или *deadly embrace —* смертельные объятия). Проще говоря, взаимное блокирование потоков — это состояние, когда два потока не могут продолжить выполнение, так как каждый из них ожидает, пока второй освободит занятый им ресурс.

Предположим, например, что каждый из двух потоков для продолжения работы должен завладеть двумя мьютексами (МТХА и МТХВ). Предположим, что в опреде­ленный момент времени первый поток завладел мьютексом МТХА и собирается завладеть мьютексом МТХВ. Но в этот же момент второй поток завладел мьютек­сом МТХВ и собирается завладеть мьютексом МТХА. Таким образом, каждый из по­токов завладел только частью ресурсов, необходимых для продолжения его ра­боты, и ожидает, пока вторая часть ресурсов освободится. Очевидно, что каждый из потоков не освободит первую часть ресурсов, пока не получит доступ ко вто­рой части, однако это никогда не произойдет. В результате оба потока взаимно блокируют друг друга и не могут продолжить работу.

Приведенный пример — один из самых простых и легко предсказуемых слу­чаев. Предотвратить взаимное блокирование двух потоков относительно легко. К сожалению, вы можете столкнуться со значительно более сложными сценари­ями. Представьте, например, что поток Х блокируется потоком Y, поток Y, в свою очередь, блокируется потоком Z, а поток Z блокируется потоком X.

Рекомендации по поводу предотвращения взаимного блокирования потоков можно обнаружить фактически в любой хорошей книге, посвященной програм­мированию в многозадачной среде. Самым лучшим оружием против взаимного блокирования является здравый смысл разработчика программы. Постарайтесь избегать ситуаций, в которых ваша программа ожидает освобождения ресурса ничем не ограниченное время. Ограничьте время ожидания некоторым конечным значением. Если в течение указанного вами периода времени ресурс все еще ос­тается занятым, освободите все ресурсы, которыми владеет ваша программа, и спустя некоторое время повторите попытку завладеть полным комплектом ресур­сов, необходимых для продолжения работы. В случае, аналогичном рассмотрен­ному ранее, где речь идет о двух мьютексах МТХА и МТХВ, вы можете попытаться организовать работу обоих потоков таким образом, чтобы оба они пытались зав­ладеть сначала мьютексом МТХА, а затем мьютексом МТХВ. В этом случае взаимно­го блокирования также удастся избежать.

Использование событий

Если событие будет использоваться потоками одного процесса, вы можете не присваивать этому событию имя. Чтобы создать событие, обратитесь к вызову **CreateEvent** и передайте ему четыре аргумента. Первый аргумент — атрибуты без­опасности. Если вы намерены использовать атрибуты по умолчанию, присвойте этому аргументу значение NULL. Второй аргумент определяет тип события *(мануалъное — manual* или *единичное — single),* а третий аргумент соответствует на­чальному состоянию события. Наконец, последний аргумент соответствует име­ни события. Если вы хотите создать безымянное событие, передайте в качестве этого аргумента значение NULL.

Если вы хотите, чтобы к событию смогли обратиться другие процессы, вы должны присвоить этому событию уникальное имя. Имя не может содержать символов обратной косой. При именовании событий необходимо учитывать ре­гистр символов. Вызов **CreateEvent** создает событие с указанным именем в слу­чае, если такого события еще не создано в системе. Если событие с указанным именем уже существует, вызов **CreateEvent** просто открывает существующее со­бытие. В любом случае этот вызов возвращает вызвавшему процессу дескриптор события.

Чтобы определить состояние события, вы можете обратиться к любой из функ­ций слежения за состоянием объектов синхронизации (например, WaitForSingleObject). Более подробно об этом рассказано несколько ранее, в раз­деле «Выбор метода синхронизации».

Чтобы перевести событие в сигнальное состояние, необходимо использовать вызовы **SetEvent** или **PulseEvent**. Вызов **PulseEvent** переводит событие в сигналь­ное состояние на период времени, пока все ожидающие это событие потоки не прореагируют на него. После этого событие автоматически сбрасывается. Единич­ное событие остается в сигнальном состоянии до того момента, пока на него не прореагирует один из ожидающих потоков. После этого единичное событие ав­томатически сбрасывается. Мануальное событие остается в сигнальном состоя­нии до тех пор, пока не произойдет обращение к функции **ResetEvent**. Чтобы закрыть событие, в котором вы больше не нуждаетесь, необходимо обратиться к функции **CloseHandle**. Простой пример использования событий приведен в лис­тинге 1.

Использование мьютексов

Вызов **CreateMutex** предназначен для создания нового мьютекса, а вызов **OpenMutex** — для открытия уже существующего мьютекса. Как и в случае с другими объекта­ми синхронизации, если вы хотите, чтобы мьютекс использовался несколькими разными процессами, вы должны присвоить мьютексу имя. Вызов **CreateMutex** принимает в качестве аргументов атрибуты безопасности (обычно NULL), флаг, определяющий, будет ли мьютекс изначально принадлежать потоку, который его создает, и имя мьютекса, которое можно не указывать.

Если поток желает завладеть мьютексом, он должен обратиться к одной из функций ожидания объекта синхронизации (например, **WaitForSingleObject**). Если вызов функции завершается успехом, вызвавший ее поток становится обладате­лем мьютекса. Поток, владеющий мьютексом, может завладеть им повторно. Од­нако другие потоки, пытающиеся завладеть мьютексом, будут вынуждены ожи­дать до тех пор, пока поток-владелец не освободит мьютекс.

Чтобы освободить мьютекс, поток-владелец должен обратиться к функции **ReleaseMutex**. Если ранее поток завладел мьютексом несколько раз (то есть, напри­мер, этот поток несколько раз обращался к функции **WaitForSingleObject** в отно­шении одного и того же мьютекса), чтобы освободить мьютекс, поток должен обратиться к **ReleaseMutex** точно такое же количество раз. Если мьютекс больше не нужен, программа должна обратиться к вызову **CloseHandle** для того, чтобы закрыть мьютекс. Пример программы, использующей мьютекс, приведен в лис­тинге 2.

Использование критических секций

Критические секции работают приблизительно так же, как мыотексы, однако их нельзя использовать для синхронизации работы нескольких процессов. Крити­ческая секция ограничивает доступ к некоторому участку кода для нескольких потоков, принадлежащих одному и тому же процессу. Конечно, это ограничивает область использования критических секций, однако критические секции работа­ют быстрее, чем мыотексы.

Чтобы использовать критическую секцию, необходимо создать в глобальной памяти переменную CRITICAL\_SECTION. После этого программа должна один раз обратиться к функции **InitializeCriticalSection**. В начале участка кода, доступ к которому требуется ограничить, необходимо разместить обращение к функции **EnterCriticalSection**. В конце критического участка кода следует разместить об­ращение к функции **LeaveCriticalSection**. После того как один из потоков обра­тится к функции **EnterCriticalSection**, выполнение остальных потоков, обращаю­щихся к этой функции, будет блокировано. Как только поток, выполняющий критический участок кода, обратится к вызову **LeaveCriticalSection**, один из по­токов, блокированных вызовом **EnterCriticalSection**, сможет продолжить работу. Если в момент обращения потока к **LeaveCriticalSection** ни один другой поток не ожидает освобождения критической секции, следующий поток, обратившийся к

**EnterCnticalSection**, сможет успешно приступить к выполнению критического участка кода,

Состояние критической секции можно проверить при помощи вызова **TryCriticalSection**. Если вызов вернул значение «истина», значит, критический уча­сток свободен, и поток может приступить к его выполнению. Если вызов **TryCriticalSection** вернул значение «ложь», критическая секция уже занята, и поток должен подождать некоторое время, а потом еще раз проверить, можно ли ей завладеть. Ожидая освобождения критической секции, поток может выполнять ка­кую-либо другую полезную работу.

Использование семафоров

Так же как и мьютексы, семафоры служат для ограничения одновременного дос­тупа к ресурсу нескольких потоков. В любой момент времени мьютекс разреша­ет доступ к ресурсу только для одного потока. Используя семафор, вы можете организовать работу программы таким образом, что к ресурсу одновременно смо­гут получить доступ несколько потоков, однако количество этих потоков будет ограничено. Создавая семафор, вы указываете максимальное количество пото­ков, которые одновременно смогут работать с ресурсом. Если в некоторый момент времени с ресурсом работает меньшее количество потоков, семафор ос­тается в сигнальном состоянии. Как только количество потоков, желающих ра­ботать с ресурсом, становится больше допустимого, семафор блокирует доступ к семафору.

Для работы с семафором служат вызовы **CreateSemaphore** и **OpenSemaphore**. Как и в других случаях, вызов **CreateSemaphore** либо создает новый семафор с указан­ным именем или открывает уже существующий семафор. Вызов **OpenSemaphore** срабатывает только в случае, если семафор с указанным именем уже существует.

При создании семафора необходимо указать атрибуты безопасности, началь­ное значение счетчика, максимальное допустимое значение счетчика и имя (ко­торое можно не указывать). Обращение к семафору со стороны какого-либо по­тока уменьшает значение счетчика на единицу. Как только значение счетчика становится равным нулю, семафор блокирует доступ к ресурсу. Доступ к ресур­су будет заблокирован до тех пор, пока один из потоков, работающих с ресур­сом, не вызовет **ReleaseSemaphore**.

Вызов **ReleaseSemaphore** добавляет к значению счетчика семафора указанное в качестве параметра значение (обычно 1). При помощи этого же вызова можно определить значение счетчика до того, как оно было увеличено. Пример исполь­зования семафора приведен в листинге 3.

Ожидание нескольких объектов

Вызов WaitForSingleObject позволяет следить за состоянием одного объекта синх­ронизации, однако, если ваша программа должна реагировать на изменение состо­яния нескольких объектов, вы можете использовать вызов Wa1tForMu1t1p1eObjects, В качестве аргумента этой функции необходимо передать массии дескрипторов объектов синхронизации и флаг, указывающий, будет ли программа реагировать на изменение состояния любого из объектов или для продолжения се функциони­рования необходимо изменение состояния абсолютно всех объектов.

Есть несколько других версий вызовов, позволяющих следить за состоя­нием объектов синхронизации, однако вызовы WaitForSingleObject и WaitForMultipleObjects являются базовыми.

Использование блокированных переменных

Windows позволяет организовать работу программы таким образом, чтобы к не­которым переменным в любой момент времени мог обращаться только один по­ток. Для этого служат так называемые блокированные вызовы. Например, вызов **InterlockedIncrement** добавляет 1 к 32-битному значению и возвращает преды­дущее значение. Используя **InterlockedIncrement**, вы можете быть уверенными, что ни один другой поток не сможет обратиться к переменной во время модифи­кации ее значения.

Конечно же, если ваша программа использует только собственные перемен­ные, изменять их значения могут только потоки, принадлежащие основному про­цессу вашей программы. Однако, если вы используете общую для нескольких процессов область памяти, возможно, вам потребуется предотвратить одновремен­ный доступ к этой памяти со стороны нескольких потоков, принадлежащих раз­ным процессам.

К вызовам, обеспечивающим корректный доступ к общим переменным со сто­роны нескольких потоков, относятся **InterlockedCompareExchange, InterockedDecrement, InterlockedExchange, InterlockedExchangeAdd** и **InterlockedIncrement**. Чаще всего вместо этих вызовов программисты предпочитают использовать более высокоуровневые механизмы синхронизации, такие как, например, события, мьютексы или се­мафоры.