**Синхронизация процессов при работе с Windows**

**Анатолий Тенцер**

**Содержание**

[Синхронизация процессов при работе с Windows 3](#_Toc40611446)

[Функции синхронизации 3](#_Toc40611447)

[Функции, ожидающие единственного объекта 3](#_Toc40611448)

[Функции, ожидающие нескольких объектов 4](#_Toc40611449)

[Прерывание ожидания по запросу на завершение операции ввода-вывода или APC 7](#_Toc40611450)

[Объекты синхронизации 8](#_Toc40611451)

[Event (событие) 8](#_Toc40611452)

[Mutex (Mutually Exclusive) 11](#_Toc40611453)

[Semaphore (семафор) 13](#_Toc40611454)

[Waitable timer (таймер ожидания) 15](#_Toc40611455)

[lpDueTime 16](#_Toc40611456)

[lPeriod 16](#_Toc40611457)

[pfnCompletionRoutine 16](#_Toc40611458)

[lpArgToCompletionRoutine 16](#_Toc40611459)

[fResume 16](#_Toc40611460)

[Дополнительные объекты синхронизации 18](#_Toc40611461)

[Сообщение об изменении папки (change notification) 18](#_Toc40611462)

[Устройство стандартного ввода с консоли (console input) 21](#_Toc40611463)

[Задание (Job) 21](#_Toc40611464)

[Процесс (Process) 21](#_Toc40611465)

[Поток (thread) 22](#_Toc40611466)

[Дополнительные механизмы синхронизации 22](#_Toc40611467)

[Критические секции 22](#_Toc40611468)

[Защищенный доступ к переменным (Interlocked Variable Access) 24](#_Toc40611469)

[Резюме 25](#_Toc40611470)

# Синхронизация процессов при работе с Windows

Задача синхронизации встает при одновременном доступе нескольких процессов (или нескольких потоков одного процесса) к какому-либо ресурсу. Поскольку поток в Win32 может быть остановлен в любой, заранее ему неизвестный момент времени возможна ситуация, когда один из потоков не успел завершить модификацию ресурса (например, отображенной на файл области памяти), но был остановлен, а другой поток попытался обратиться к этому же ресурсу. В этот момент ресурс находится в несогласованном состоянии, и последствия обращения к нему могут быть самыми неожиданными – от порчи данных, до нарушения защиты памяти.

Главной идеей, положенной в основу синхронизации потоков в Win32 является использование объектов синхронизации и функций ожидания. Объекты могут находиться в одном из двух состояний – Signaled или Not Signaled. Функции ожидания блокируют выполнение потока до тех пор, пока заданный объект находится в состоянии Not Signaled. Таким образом, поток, которому необходим эксклюзивный доступ к ресурсу, должен выставить какой-либо объект синхронизации в несигнальное состояние, а по окончании – сбросить его в сигнальное. Остальные потоки должны перед доступом к этому ресурсу вызвать функцию ожидания, которая позволит им дождаться освобождения ресурса.

Рассмотрим, какие объекты и функции синхронизации предоставляет нам Win32 API.

## Функции синхронизации

Функции синхронизации делятся на две основные категории – это функции, ожидающие единственного объекта и функции, ожидающие одного из нескольких объектов

### Функции, ожидающие единственного объекта

Простейшей функцией ожидания является

**function** WaitForSingleObject(  
 hHandle: THandle; // идентификатор объекта  
 dwMilliseconds: DWORD // период ожидания  
): DWORD; **stdcall**;

Функция ожидает перехода объекта hHandle в сигнальное состояние в течении dwMilliseconds миллисекунд. Если в качестве параметра dwMilliseconds передать значение INFINITE, функция будет ждать в течение неограниченного времени. Если dwMilliseconds равен 0, то функция проверяет состояние объекта и немедленно возвращает управление.

Функция возвращает одно из следующих значений:

|  |  |
| --- | --- |
| WAIT\_ABANDONED | Поток, владевший объектом, завершился, не переведя объект в сигнальное состояние. |
| WAIT\_OBJECT\_0 | Объект перешел в сигнальное состояние |
| WAIT\_TIMEOUT | Истек срок ожидания. Обычно в этом случае генерируется ошибка, либо функция вызывается в цикле до получения другого результата |
| WAIT\_FAILED | Произошла ошибка, например неверное значение hHandle. Более подробную информацию можно получить, вызвав GetLastError |

Следующий фрагмент кода запрещает Action1 до перехода объекта ObjectHandle в сигнальное состояние. Например, таким образом можно дожидаться завершения процесса, предав в качестве ObjectHandle его идентификатор, полученный функцией CreateProcess.

**var**  
 Reason: DWORD;  
 ErrorCode: DWORD;  
  
Action1.Enabled := FALSE;  
**try**  
 **repeat**  
 Application.ProcessMessages;  
 Reason := WailForSingleObject(ObjectHandle, 10);  
 **if** Reason = WAIT\_FAILED **then** **begin**  
 ErrorCode := GetLastError;  
 **raise** Exception.CreateFmt(  
 ‘Wait for object failed with error: %d’, [ErrorCode]);  
 **end**;  
 **until** Reason <> WAIT\_TIMEOUT;  
**finally**  
 Actionl.Enabled := TRUE;  
**end**;

В случае, когда требуется одновременно с ожиданием объекта, перевести в сигнальное состояние другой объект может использоваться функция:

**function** SignalObjectAndWait(  
 hObjectToSignal: THandle; // объект, который будет переведен в  
 // сигнальное состояние   
 hObjectToWaitOn: THandle; // объект, которого ожидает функция  
 dwMilliseconds: DWORD; // период ожидания  
 bAlertable: BOOL // задает, должна ли функция возвращать  
 // управление в случае запроса на   
 // завершение операции ввода-вывода  
): DWORD; **stdcall**;

Возвращаемые значения аналогичны функции WaitForSingleObject.

**!** В модуле Windows.pas эта функция ошибочно объявлена, как возвращающая значение BOOL. Если Вы намерены её использовать – объявите её корректно или используйте приведение типа возвращенного значения к DWORD

Объект hObjectToSignal может быть семафором, событием (event), либо мутексом. Параметр bAlertable определяет, будет ли прерываться ожидание объекта, в случае, если операционная система запросит у потока окончание операции асинхронного ввода-вывода, либо асинхронный вызов процедуры. Более подробно это обсуждается ниже.

### Функции, ожидающие нескольких объектов

Иногда требуется задержать выполнение потока до срабатывания одного или всех сразу из группы объектов. Для решения подобной задачи служат следующие функции:

**type**  
 TWOHandleArray = **array**[0..MAXIMUM\_WAIT\_OBJECTS - 1] **of** THandle;  
 PWOHandleArray = ^TWOHandleArray;  
  
**function** WaitForMultipleObjects(  
 nCount: DWORD; // Задает количество объектов  
 lpHandles: PWOHandleArray; // Адрес массива объектов  
 bWaitAll: BOOL; // Задает, требуется ожидание всех  
 // объектов или любого  
 dwMilliseconds: DWORD // Период ожидания  
): DWORD; **stdcall**;

Функция возвращает одно из следующих значений:

|  |  |
| --- | --- |
| Число в диапазоне от WAIT\_OBJECT\_0 до WAIT\_OBJECT\_0 + nCount – 1 | Если bWaitAll равно TRUE, то это число означает, что все объекты перешли в сигнальное состояние.  Если FALSE – то, вычтя из возвращенного значения WAIT\_OBJECT\_0, мы получим индекс объекта в массиве lpHandles. |
| Число в диапазоне от WAIT\_ABANDONED\_0 до WAIT\_ABANDONED\_0 + nCount – 1 | Если bWaitAll равно TRUE – это означает, что все перешли в сигнальное состояние, но хотя бы один из владевших ими потоков завершился, не сделав объект сигнальным.  Если FALSE – то, вычтя из возвращенного значения WAIT\_ABANDONED\_0, мы получим индекс объекта в массиве lpHandles, поток, владевший которым, завершился, не сделав его сигнальным. |
| WAIT\_TIMEOUT | Истек период ожидания |
| WAIT\_FAILED | Произошла ошибка |

Например, в следующем фрагменте кода программа пытается модифицировать два различных ресурса, разделяемых между потоками.

**var**  
 Handles: **array**[0..1] **of** THandle;  
 Reason: DWORD;  
 RestIndex: Integer;  
  
...  
  
Handles[0] := OpenMutex(SYNCHRONIZE, FALSE, ‘FirstResource’);  
Handles[1] := OpenMutex(SYNCHRONIZE, FALSE, ‘SecondResource’);  
// Ждем первого из объектов  
Reason := WaitForMultipleObjects(2, @Handles, FALSE, INFINITE);  
**case** Reason **of**  
 WAIT\_FAILED: RaiseLastWin32Error;  
 WAIT\_OBJECT\_0, WAIT\_ABANDONED\_0:  
 **begin**  
 ModifyFirstResource;  
 RestIndex := 1;  
 **end**;  
 WAIT\_OBJECT\_0 + 1, WAIT\_ABANDONED\_0 + 1:  
 **begin**  
 ModifySecondResource;  
 RestIndex := 0;  
 **end**;  
 // WAIT\_TIMEOUT возникнуть не может  
**end**;  
// Теперь ожидаем освобождения следующего объекта  
**if** WailForSingleObject(Handles[RestIndex],   
 INFINITE) = WAIT\_FAILED **then**  
 RaiseLastWin32Error;  
// Дождались, модифицируем оставшийся ресурс.  
**if** RestIndex = 0 **then**  
 ModifyFirstResource  
**else**  
 ModifySecondResource;

Описанную выше технику можно применять, если Вы точно знаете, что задержка ожидания объекта окажется небольшой. В противном случае Ваша программа окажется “замороженной” и не сможет даже перерисовать своё окно. Если период задержки может оказаться значительным, то необходимо дать программе возможность реагировать на сообщения Windows. Выходом может служить использование функций с ограниченным периодом ожидания (и повторный вызов, в случае возврата WAIT\_TIMEOUT), либо использование функции:

**function** MsgWaitForMultipleObjects(  
 nCount: DWORD; // количество объектов синхронизации  
 **var** pHandles; // адрес массива объектов  
 fWaitAll: BOOL; // Задает, требуется ожидание всех  
 // объектов или любого  
 dwMilliseconds, // Период ожидания  
 dwWakeMask: DWORD // Тип события, прерывающего ожидание  
): DWORD; **stdcall**;

Главное отличие этой функции от предыдущей – параметр dwWakeMask, который является комбинацией битовых флагов QS\_XXX и задает типы сообщений, которые прерывают ожидание функции, независимо от состояния ожидаемых объектов. Например, маска QS\_KEY позволяет прервать ожидание при появлении в очереди сообщений WM\_KEYUP, WM\_KEYDOWN, WM\_SYSKEYUP или WM\_SYSKEYDOWN, а маска QS\_PAINT - сообщения WM\_PAINT. Полный список значений, допустимых для dwWakeMask имеется в документации по Windows SDK. При появлении в очереди потока, вызвавшего функцию, сообщений, соответствующих заданной маске функция возвращает значение WAIT\_OBJECT\_0 + nCount. Получив это значение, Ваша программа может обработать его и снова вызвать функцию ожидания. Рассмотрим пример с запуском внешнего приложения. Необходимо, чтобы на время его работы вызывающая программа не реагировала на ввод пользователя, однако её окно должно продолжать перерисовываться.

**procedure** TForm1.Button1Click(Sender: TObject);  
**var**  
 PI: TProcessInformation;  
 SI: TStartupInfo;  
 Reason: DWORD;  
 Msg: TMsg;  
**begin**  
 // Инициализируем структуру TStartupInfo  
 FillChar(SI, SizeOf(SI), 0);  
 SI.cb := SizeOf(SI);  
 // Запускаем внешнюю программу  
 Win32Check(CreateProcess(**NIL**, 'COMMAND.COM', **NIL**,  
 **NIL**, FALSE, 0, **NIL**, **NIL**, SI, PI));  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
// Попробуйте заменить нижеприведенный код на строку  
// WaitForSingleObject(PI.hProcess, INFINITE);  
// и посмотреть, как будет реагировать программа на  
// перемещение других окон над её окном  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 **repeat**  
 // Ожидаем завершения дочернего процесса или сообщения   
 // перерисовки WM\_PAINT  
 Reason := MsgWaitForMultipleObjects(1, PI.hProcess, FALSE,  
 INFINITE, QS\_PAINT);  
 **if** Reason = WAIT\_OBJECT\_0 + 1 **then** **begin**  
 // В очереди сообщений появился WM\_PAINT – Windows  
 // требует обновить окно программы.  
 // Удаляем сообщение из очереди  
 PeekMessage(Msg, 0, WM\_PAINT, WM\_PAINT, PM\_REMOVE);  
 // И перерисовываем наше окно  
 Update;  
 **end**;  
 // Повторяем цикл, пока не завершится дочерний процесс  
 **until** Reason = WAIT\_OBJECT\_0;  
 // Удаляем из очереди накопившиеся там сообщения  
 **while** PeekMessage(Msg, 0, 0, 0, PM\_REMOVE) **do**;  
 CloseHandle(PI.hProcess);  
 CloseHandle(PI.hThread)  
**end**;

Если в потоке, вызывающем функции ожидания явно (функцией CreateWindow) или неявно (используя TForm, DDE, COM) создаются окна Windows – поток должен обрабатывать сообщения. Поскольку широковещательные сообщения посылаются всем окнам в системе – поток, не обрабатывающий сообщения может вызвать взаимоблокировку, (система ждет, когда поток обработает сообщение, поток – когда система или другие потоки освободят объект) и привести к зависанию Windows. Если в Вашей программе имеются подобные фрагменты – необходимо использовать MsgWaitForMultipleObjects или MsgWaitForMultipleObjectsEx и позволять прервать ожидание для обработки сообщений. Алгоритм аналогичен вышеприведенному примеру.

### Прерывание ожидания по запросу на завершение операции ввода-вывода или APC

Windows поддерживает асинхронные вызовы процедур. При создании каждого потока (thread) с ним ассоциируется очередь асинхронных вызовов процедур (APC queue). Операционная система (или приложение пользователя, при помощи функции QueueUserAPC) может помещать в неё запросы на выполнение функций в контексте этого потока. Эти функции не могут быть выполнены немедленно, поскольку поток может быть занят. Поэтому, операционная система вызывает их, когда поток вызывает одну из следующих функций ожидания:

**function** SleepEx(  
 dwMilliseconds: DWORD; // Период ожидания  
 bAlertable: BOOL // задает, длжна ли функция возвращать  
 // управление в случае запроса на   
 // асинхронный вызов процедуры  
): DWORD; **stdcall**;

**function** WaitForSingleObjectEx(  
 hHandle: THandle; // Идентификатор объекта  
 dwMilliseconds: DWORD; // Период ожидания  
 bAlertable: BOOL // задает, длжна ли функция возвращать  
 // управление в случае запроса на   
 // асинхронный вызов процедуры  
): DWORD; **stdcall**;

**function** WaitForMultipleObjectsEx(  
 nCount: DWORD; // количество объектов  
 lpHandles: PWOHandleArray;// адрес массива идентификаторов объектов  
 bWaitAll: BOOL; // Задает, требуется ожидание всех  
 // объектов или любого  
 dwMilliseconds: DWORD; // Период ожидания  
 bAlertable: BOOL // задает, должна ли функция возвращать  
 // управление в случае запроса на   
 // асинхронный вызов процедуры  
): DWORD; **stdcall**;

**function** SignalObjectAndWait(  
 hObjectToSignal: THandle; // объект, который будет переведен в  
 // сигнальное состояние   
 hObjectToWaitOn: THandle; // объект, которого ожидает функция  
 dwMilliseconds: DWORD; // период ожидания  
 bAlertable: BOOL // задает, должна ли функция возвращать  
 // управление в случае запроса на   
 // асинхронный вызов процедуры  
): DWORD; **stdcall**;

**function** MsgWaitForMultipleObjectsEx(  
nCount: DWORD; // количество объектов синхронизации  
 **var** pHandles; // адрес массива объектов  
 fWaitAll: BOOL; // Задает, требуется ожидание всех  
 // объектов или любого  
 dwMilliseconds, // Период ожидания  
 dwWakeMask: DWORD // Тип события, прерывающего ожидание  
 dwFlags: DWORD // Дополнительные флаги  
): DWORD; **stdcall**;

Если параметр bAlertable равен TRUE (либо dwFlags в функции MsgWaitForMultipleObjectsEx содержит MWMO\_ALERTABLE) , то при появлении в очереди APC запроса на асинхронный вызов процедуры операционная система выполняет вызовы всех имеющихся в очереди процедур, после чего функция возвращает значение WAIT\_IO\_COMPLETION.

Такой механизм позволяет реализовать, например, асинхронный ввод-вывод. Поток может инициировать фоновое выполнение одной или нескольких операций ввода-вывода функциями ReadFileEx или WriteFileEx, передав им адреса функций-обработчиков завершения операции. По завершении вызовы этих функций будут поставлены в очередь асинхронного вызова процедур. В свою очередь, инициировавший операции поток, когда он будет готов обработать результаты, может, используя одну из вышеприведенных функций ожидания, позволить операционной системе вызвать функции-обработчики. Поскольку очередь APC реализована на уровне ядра ОС, она более эффективна, чем очередь сообщений и позволяет реализовать гораздо более эффективный ввод-вывод.

### Объекты синхронизации

Объектами синхронизации называются объекты Windows, идентификаторы которых могут использоваться в функциях синхронизации. Они делятся на две группы – объекты, использующиеся только для синхронизации и объекты, которые используются в других целях, но могут вызывать срабатывание функций ожидания. К первой группе относятся:

#### Event (событие)

Позволяет известить один или несколько ожидающих потоков о наступлении события. Event бывает

|  |  |
| --- | --- |
| Отключаемый вручную | Будучи установленным в сигнальное состояние, остается в нем до тех пор, пока не будет переключен явным вызовом функции ResetEvent |
| Автоматически отключаемый | Автоматически переключается в несигнальное состояние операционной системой, когда один из ожидающих его потоков завершается. |

Для создания объекта используется функция:

**function** CreateEvent(  
 lpEventAttributes: PSecurityAttributes; // Адрес структуры  
 // TSecurityAttributes  
 bManualReset, // Задает, будет Event переключаемым   
 // вручную (TRUE) или автоматически (FALSE)  
 bInitialState: BOOL; // Задает начальное состояние. Если TRUE -   
 // объект в сигнальном состоянии  
 lpName: PChar // Имя или **NIL**, если имя не требуется  
): THandle; **stdcall**; // Возвращает идентификатор созданного  
 // объекта

Структура TSecurityAttributes описана, как:

TSecurityAttributes = record  
 nLength: DWORD; // Размер структуры, должен   
 // инициализироваться как   
 // SizeOf(TSecurityAttributes)  
 lpSecurityDescriptor: Pointer; // Адрес дескриптора защиты. В   
 // Windows 95 и 98 игнорируется  
 // Обычно можно указывать **NIL**  
 bInheritHandle: BOOL; // Задает, могут ли дочерние   
 // процессы наследовать объект  
**end**;

Если не требуется задание особых прав доступа под Windows NT или возможности наследования объекта дочерними процессами, в качестве параметра lpEventAttributes можно передавать NIL. В этом случае объект не может наследоваться дочерними процессами и ему задается дескриптор защиты «по умолчанию».

Параметр lpName позволяет разделять объекты между процессами. Если lpName совпадает с именем уже существующего объекта типа Event, созданного текущим или любым другим процессом, функция не создает нового объекта, а возвращает идентификатор уже существующего. При этом игнорируются параметры bManualReset, bInitialState и lpSecurityDescriptor. Проверить, был объект создан, или используется уже существующий можно следующим образом:

hEvent := CreateEvent(**NIL**, TRUE, FALSE, ‘EventName’);  
**if** hEvent = 0 **then**  
 RaiseLastWin32Error;  
**if** GetLastError = ERROR\_ALREADY\_EXISTS **then** **begin**  
 // Используем ранее созданный объект  
**end**;

Если объект используется для синхронизации внутри одного процесса, его можно объявить как глобальную переменную и создавать без имени.

Имя объекта не должно совпадать с именем любого из существующих объектов типов Semaphore, Mutex, Job, Waitable Timer или FileMapping. В случае совпадения имен, функция возвращает ошибку.

Если известно, что Event уже создан, для получения доступа к нему можно вместо CreateEvent воспользоваться функцией:

**function** OpenEvent(  
 dwDesiredAccess: DWORD; // Задает права доступа к объекту  
 bInheritHandle: BOOL; // Задает, может ли объект наследоваться   
 // дочерними процессами  
 lpName: PChar // Имя объекта  
): THandle; **stdcall**;

Функция возвращает идентификатор объекта, либо 0, в случае ошибки. Параметр dwDesiredAccess может принимать одно из следующих значений:

|  |  |
| --- | --- |
| EVENT\_ALL\_ACCESS | Приложение получает полный доступ к объекту |
| EVENT\_MODIFY\_STATE | Приложение может изменять состояние объекта функциями SetEvent и ResetEvent |
| SYNCHRONIZE | Только для Windows NT – приложение может использовать объект только в функциях ожидания |

После получения идентификатора можно приступать к его использованию. Для этого имеются следующие функции:

**function** SetEvent(hEvent: THandle): BOOL; **stdcall**;

Устанавливает объект в сигнальное состояние

**function** ResetEvent(hEvent: THandle): BOOL; **stdcall**;

Сбрасывает объект, устанавливая его в несигнальное состояние

**function** PulseEvent(hEvent: THandle): BOOL; **stdcall**

Устанавливает объект в сигнальное состояние, дает отработать всем функциям ожидания, ожидающим этот объект, а затем снова сбрасывает его.

В WinAPI события используются, для выполнения операций асинхронного ввода-вывода. Следующий пример показывает, как приложение инициирует запись одновременно в два файла, а затем ожидает завершения записи перед продолжением работы. Такой подход может обеспечить более высокую производительность при высокой интенсивности ввода-вывода, чем последовательная запись.

**var**  
 Events: **array**[0..1] **of** THandle; // Массив объектов синхронизации  
 Overlapped: **array**[0..1] **of** TOverlapped;   
  
...  
  
// Создаем объекты синхронизации  
Events[0] := CreateEvent(**NIL**, TRUE, FALSE, **NIL**);  
Events[1] := CreateEvent(**NIL**, TRUE, FALSE, **NIL**);  
  
// Инициализируем структуры TOverlapped  
FillChar(Overlapped, SizeOf(Overlapped), 0);  
Overlapped[0].hEvent := Events[0];  
Overlapped[1].hEvent := Events[1];  
  
// Начинаем асинхронную запись в файлы  
WriteFile(hFirstFile, FirstBuffer, SizeOf(FirstBuffer),  
 FirstFileWritten, @Overlapped[0]);  
WriteFile(hSecondFile, SecondBuffer, SizeOf(SecondBuffer),  
 SecondFileWritten, @Overlapped[1]);  
  
// Ожидаем завершения записи в оба файла  
WaitForMultipleObjects(2, @Events, TRUE, INFINITE);  
  
// Уничтожаем объекты синхронизации  
CloseHandle(Events[0]);  
CloseHandle(Events[1]);

По завершении работы с объектом, он должен быть уничтожен функцией CloseHandle.

Delphi предоставляет класс TEvent, инкапсулирующий функциональность объекта Event. Класс расположен в модуле SyncObjs.pas и объявлен следующим образом:

**type**  
 TWaitResult = (wrSignaled, wrTimeout, wrAbandoned, wrError);  
  
 TEvent = **class**(THandleObject)  
 **public**  
 **constructor** Create(EventAttributes: PSecurityAttributes;  
 ManualReset, InitialState: Boolean; const Name: string);  
 **function** WaitFor(Timeout: DWORD): TWaitResult;  
 **procedure** SetEvent;  
 **procedure** ResetEvent;  
 **end**;

Назначение методов очевидно из их названий. Использование этого класса позволяет не вдаваться в тонкости реализации вызываемых функций Windows API. Для простейших случаев объявлен еще один класс с упрощенным конструктором.

**type**  
 TSimpleEvent = **class**(TEvent)  
 **public**  
 **constructor** Create;  
 **end**;  
  
…  
  
**constructor** TSimpleEvent.Create;  
**begin**  
 FHandle := CreateEvent(**nil**, True, False, **nil**);  
**end**;

#### Mutex (Mutually Exclusive)

Мутекс – это объект синхронизации, который находится в сигнальном состоянии только тогда, когда он не принадлежит ни одному из процессов. Как только хотя бы один процесс запрашивает владение мутексом, он переходит в несигнальное состояние и остается в нем до тех пор, пока не будет освобожден владельцем. Такое поведение позволяет использовать мутексы для синхронизации совместного доступа нескольких процессов к разделяемому ресурсу. Для создания мутекса используется функция:

**function** CreateMutex(  
 lpMutexAttributes: PSecurityAttributes; // Адрес структуры   
 // TSecurityAttributes  
 bInitialOwner: BOOL; // Задает, будет ли процесс владеть  
 // мутексом сразу после создания  
 lpName: PChar // Имя мутекса  
): THandle; **stdcall**;

Функция возвращает идентификатор созданного объекта, либо 0. Если мутекс с заданным именем уже был создан, возвращается его идентификатор. В этом случае функция GetLastError вернет код ошибки ERROR\_ALREDY\_EXISTS. Имя не должно совпадать с именем уже существующего объекта типов Semaphore, Event, Job, Waitable Timer или FileMapping

Если неизвестно, существует ли уже мутекс с таким именем, программа не должна запрашивать владение объектом при создании (т.е. должна передать в качестве bInitialOwner значение FALSE).

Если мутекс уже существует, приложение может получить его идентификатор функцией

**function** OpenMutex(  
 dwDesiredAccess: DWORD; // Задает права доступа к объекту  
 bInheritHandle: BOOL; // Задает, может ли объект наследоваться   
 // дочерними процессами  
 lpName: PChar // Имя объекта  
): THandle; **stdcall**;

Параметр dwDesiredAccess может принимать одно из следующих значений

|  |  |
| --- | --- |
| MUTEX\_ALL\_ACCESS | Приложение получает полный доступ к объекту |
| SYNCHRONIZE | Только для Windows NT – приложение может использовать объект только в функциях ожидания и функции ReleaseMutex |

Функция возвращает идентификатор открытого мутекса, либо 0, в случае ошибки. Мутекс переходит в сигнальное состояние после срабатывания функции ожидания, в которую был передан его идентификатор. Для возврата в несигнальное состояние служит функция

**function** ReleaseMutex(hMutex: THandle): BOOL; **stdcall**;

Если несколько процессов обмениваются данными, например, через файл, отображенный на память, каждый из них должен содержать следующий код для обеспечения корректного доступа к общему ресурсу:

**var**  
 Mutex: THandle;  
  
// При инициализации программы  
Mutex := CreateMutex(**NIL**, FALSE, ‘UniqueMutexName’);  
**if** Mutex = 0 **then**  
 RaiseLastWin32Error;  
  
...  
// Доступ к ресурсу  
WaitForSingleObject(Mutex, INFINITE);  
**try**  
 // Доступ к ресурсу, захват мутекса гарантирует,  
 // что остальные процессы пытающиеся получить доступ  
 // будут остановлены на функции WaitForSingleObject  
 ...  
**finally**  
 // Работа с ресурсом окончена, освобождаем его  
 // для остальных процессов  
 ReleaseMutex(Mutex);  
**end**;  
  
...  
// При завершении программы  
CloseHandle(Mutex);

Подобный код удобно инкапсулировать в класс, который создает защищенный ресурс, мутекс, имеет свойства и методы для оперирования ресурсом, защищая их при помощи функций синхронизации.

Разумеется, если работа с ресурсом может потребовать значительного времени, то необходимо либо использовать функцию MsgWaitForSingleObject, либо вызывать WaitForSingleObject в цикле с нулевым периодом ожидания, проверяя код возврата. В противном случае Ваше приложение окажется замороженным. Всегда защищайте захват-освобождение объекта синхронизации при помощи блока try ... finally, иначе ошибка во время работы с ресурсом приведет к блокированию работы всех процессов, ожидающих его освобождения.

#### Semaphore (семафор)

Семафор представляет собой счетчик, содержащий целое число в диапазоне от 0 до заданной при его создании максимальной величины. Счетчик уменьшается каждый раз, когда поток успешно завершает функцию ожидания, использующую семафор и увеличивается вызовом функции ReleaseSemaphore. При достижении семафором значения 0 он переходит в несигнальное состояние, при любых других значениях счетчика – его состояние сигнальное. Такое поведение позволяет использовать семафор в качестве ограничителя доступа к ресурсу, поддерживающему заранее заданное количество подключений.

Для создания семафора служит функция:

**function** CreateSemaphore(  
 lpSemaphoreAttributes: PSecurityAttributes; // Адрес структуры   
 // TSecurityAttributes  
 lInitialCount, // Начальное значение счетчика  
 lMaximumCount: Longint; // Максимальное значение счетчика  
 lpName: PChar // Имя объекта  
): THandle; **stdcall**;

Функция возвращает идентификатор созданного семафора, либо 0, если создать объект не удалось.

Параметр lMaximumCount задает максимальное значение счетчика семафора, lInitialCount задает начальное значение счетчика и должен быть в диапазоне от 0 до lMaximumCount. lpName задает имя семафора. Если в системе уже есть семафор с таким именем, то новый не создается, а возвращается идентификатор существующего семафора. В случае если семафор используется внутри одного процесса, можно создать его без имени, передав в качестве lpName значение NIL. Имя семафора не должно совпадать с именем уже существующего объекта типов event, mutex, waitable timer, job, или file-mapping.

Идентификатор ранее созданного семафора может быть, также, получен функцией:

**function** OpenSemaphore(  
 dwDesiredAccess: DWORD; // Задает права доступа к объекту  
 bInheritHandle: BOOL; // Задает, может ли объект наследоваться   
 // дочерними процессами  
 lpName: PChar // Имя объекта  
): THandle; **stdcall**;

Параметр dwDesiredAccess может принимать одно из следующих значений:

|  |  |
| --- | --- |
| SEMAPHORE\_ALL\_ACCESS | Поток получает все права на семафор |
| SEMAPHORE\_MODIFY\_STATE | Поток может увеличивать счетчик семафора функцией ReleaseSemaphore |
| SYNCHRONIZE | Только Windows NT – поток может использовать семафор в функциях ожидания |

Для увеличения счетчика семафора используется функция:

**function** ReleaseSemaphore(  
 hSemaphore: THandle; // Идентификатор семафора  
 lReleaseCount: Longint; // Счетчик будет увеличен на эту величину  
 lpPreviousCount: Pointer // Адрес 32-битной переменной,   
 // принимающей предыдущее значение  
 // счетчика  
): BOOL; **stdcall**;

Если значение счетчика после выполнения функции превысит заданный для него функцией CreateSemaphore максимум, то ReleaseSemaphore возвращает FALSE и значение семафора не изменяется. В качестве параметра lpPreviousCount можно передать NIL, если это значение нам не нужно.

Рассмотрим пример приложения, запускающего на выполнение несколько заданий в отдельных потоках (например, программа для фоновой загрузки файлов из Internet). Если количество одновременно выполняющихся заданий будет слишком велико, то это приведет к неоправданной загрузке канала. Поэтому реализуем потоки, в которых будет выполняться задание таким образом, чтобы при превышении их количества заранее заданной величины поток останавливался и ожидал завершения работы ранее запущенных заданий.

**unit** LimitedThread;  
  
**interface**  
  
**uses** Classes;  
  
**type**  
 TLimitedThread = **class**(TThread)  
 **procedure** Execute; override;  
 **end**;  
   
**implementation**  
  
**uses** Windows;  
  
**const**  
 MAX\_THREAD\_COUNT = 10;  
  
**var**  
 Semaphore: THandle;  
  
**procedure** TLimitedThread.Execute;  
**begin**  
 // Уменьшаем счетчик семафора. Если к этому моменту уже запущено  
 // MAX\_THREAD\_COUNT потоков – счетчик равен 0 и семафор в   
 // несигнальном состоянии. Поток будет заморожен до завершения   
 // одного из запущенных ранее.  
 WaitForSingleObject(Semaphore, INFINITE);  
   
 // Здесь располагается код, отвечающий за функциональность потока,  
 // например загрузка файла  
 ...  
  
 // Поток завершил работу, увеличиваем счетчик семафора и позволяем  
 // начать обработку другим потокам.  
 ReleaseSemaphore(Semaphore, 1, **NIL**);  
**end**;  
  
**initialization** // Создаем семафор при старте программы  
 Semaphore := CreateSemaphore(**NIL**, MAX\_THREAD\_COUNT,   
 MAX\_THREAD\_COUNT, **NIL**);  
  
**finalization**  
 // Уничтожаем семафор по завершении программы  
 CloseHandle(Semaphore);  
**end**;

#### Waitable timer (таймер ожидания)

Таймер ожидания отсутствует в Windows 95 и для его использования необходима Windows 98 или Windows NT 4.0 и выше.

Таймер ожидания переходит в сигнальное состояние по завершении заданного интервала времени. Для его создания используется функция:

**function** CreateWaitableTimer(  
 lpTimerAttributes: PSecurityAttributes; // Адрес структуры   
 // TSecurityAttributes  
 bManualReset: BOOL; // Задает, будет ли таймер переходить в   
 // сигнальное состояние по завершении функции  
 // ожидания  
 lpTimerName: PChar // Имя объекта  
): THandle; **stdcall**;

Если параметр bManualReset равен TRUE, то таймер после срабатывания функции ожидания остается в сигнальном состоянии до явного вызова SetWaitableTimer, если FALSE - таймер автоматически переходит в несигнальное состояние.

Если lpTimerName совпадает с именем уже существующего в системе таймера – функция возвращает его идентификатор, позволяя использовать объект для синхронизации между процессами. Имя таймера не должно совпадать с именем уже существующих объектов типов event, semaphore, mutex, job или file-mapping.

Идентификатор уже существующего таймера можно получить функцией:

**function** OpenWaitableTimer(  
 dwDesiredAccess: DWORD; // Задает права доступа к объекту  
 bInheritHandle: BOOL; // Задает, может ли объект наследоваться   
 // дочерними процессами  
 lpTimerName: PChar // Имя объекта  
): THandle; **stdcall**;

Параметр dwDesiredAccess может принимать следующие значения:

|  |  |
| --- | --- |
| TIMER\_ALL\_ACCESS | Разрешает полный доступ к объекту |
| TIMER\_MODIFY\_STATE | Разрешает изменять состояние таймера функциями SetWaitableTimer и CancelWaitableTimer |
| SYNCHRONIZE | Только Windows NT – разрешает использовать таймер в функциях ожидания |

После получения идентификатора таймера, поток может задать время его срабатывания функцией

**function** SetWaitableTimer(  
 hTimer: THandle; // Идентификатор таймера  
 const lpDueTime: TLargeInteger; // Время срабатывания  
 lPeriod: Longint; // Период повторения срабатывания  
 pfnCompletionRoutine: TFNTimerAPCRoutine; // Процедура-обработчик  
 lpArgToCompletionRoutine: Pointer;// Параметр процедуры-обработчика  
 fResume: BOOL // Задает, будет ли операционная   
 // система «пробуждаться»  
): BOOL; **stdcall**;

Рассмотрим параметры подробнее.

##### lpDueTime

Задает время срабатывания таймера. Время задается в формате TFileTime и базируется на coordinated universal time (UTC), т.е. должно указываться по Гринвичу. Для преобразования системного времени в TFileTime используется функция SystemTimeToFileTime. Если время имеет положительный знак, оно трактуется как абсолютное, если отрицательный – как относительное от момента запуска таймера.

##### lPeriod

Задает срок между повторными срабатываниями таймера. Если lPeriod равен 0 – таймер сработает один раз.

##### pfnCompletionRoutine

Адрес функции, объявленной как:

**procedure** TimerAPCProc(  
 lpArgToCompletionRoutine: Pointer; // данные  
 dwTimerLowValue: DWORD; // младшие 32 разряда значения таймера  
 dwTimerHighValue: DWORD; // старшие 32 разряда значения таймера  
); **stdcall**;

Эта функция вызывается, когда срабатывает таймер, если поток, ожидающий его срабатывания, использует функцию ожидания, поддерживающую асинхронный вызов процедур. В функцию передаются 3 параметра:

* lpArgToCompletionRoutine – значение, переданное в качестве одноименного параметра в функцию SetWaitableTimer. Приложение может использовать его для передачи в процедуру обработки адреса блока данных, необходимых для её работы
* dwTimerLowValue и dwTimerHighValue – соответственно члены dwLowDateTime и dwHighDateTime структуры TFileTime. Они описывают время срабатывания таймера. Время задается в UTC формате (по Гринвичу).

Если дополнительная функция обработки не нужна, в качестве этого параметра можно передать NIL.

##### lpArgToCompletionRoutine

Это значение передается в функцию pfnCompletionRoutine при её вызове.

##### fResume

Определяет необходимость “пробуждения” системы, если на момент срабатывания таймера она находится в режиме экономии электроэнергии (suspended). Если операционная система не поддерживает пробуждение и fResume равно TRUE, функция SetWaitableTimer выполнится успешно, однако последующий вызов GetLastError вернет результат ERROR\_NOT\_SUPPORTED.

Если необходимо перевести таймер в неактивное состояние, это можно сделать функцией:

**function** CancelWaitableTimer(hTimer: THandle): BOOL; **stdcall**;

Эта функция не изменяет состояния таймера и не приводит к срабатыванию функций ожидания и вызову процедур-обработчиков.

По завершении работы объект должен быть уничтожен функцией CloseHandle

Создадим класс, который ожидает в отдельном потоке наступления заданного времени, а затем вызывает процедуру главного потока приложения. Такой класс может использоваться, например, в планировщике заданий. Поскольку таймер ожидания позволяет задавать время срабатывания в абсолютных величинах, отпадает необходимость постоянно анализировать текущее время, используя обычный таймер Windows.

**unit** WaitThread;  
  
**interface**  
  
**uses** Classes, Windows;  
  
**type**  
 TWaitThread = **class**(TThread)  
 WaitUntil: TDateTime;  
 procedure Execute; **override**;  
 **end**;  
  
**implementation  
  
uses** SysUtils;  
  
**procedure** TWaitThread.Execute;  
**var** Timer: THandle;  
 SystemTime: TSystemTime;  
 FileTime, LocalFileTime: TFileTime;  
**begin** Timer := CreateWaitableTimer(**NIL**, FALSE, **NIL**);  
 **try** DateTimeToSystemTime(WaitUntil, SystemTime);  
 SystemTimeToFileTime(SystemTime, LocalFileTime);  
 LocalFileTimeToFileTime(LocalFileTime, FileTime);  
 SetWaitableTimer(Timer, TLargeInteger(FileTime), 0,   
 **NIL**, **NIL**, FALSE);  
 WaitForSingleObject(Timer, INFINITE);  
 **finally** CloseHandle(Timer);  
 **end**;  
**end**;  
  
**end**.

Использовать этот класс можно, например, следующим образом:

**type** TForm1 = **class**(TForm)  
 Button1: TButton;  
 **procedure** Button1Click(Sender: TObject);  
 **private** **procedure** TimerFired(Sender: TObject);  
 **end**;  
  
...

**implementation  
  
uses** WaitThread;  
  
**procedure** TForm1.Button1Click(Sender: TObject);  
**var** T: TDateTime;  
**begin** **with** TWaitThread.Create(TRUE) **do  
 begin** OnTerminate := TimerFired;  
 FreeOnTerminate := TRUE;  
 // Срок ожидания закончится через 5 секунд  
 WaitUntil := Now + 1 / 24 / 60 / 60 \* 5;  
 Resume;  
 **end**;  
**end**;  
  
**procedure** TForm1.TimerFired(Sender: TObject);  
**begin** ShowMessage('Timer fired !');  
**end**;

### Дополнительные объекты синхронизации

Некоторые объекты Win32 API не предназначены исключительно для целей синхронизации, однако могут использоваться с функциями синхронизации. Такими объектами являются:

#### Сообщение об изменении папки (change notification)

Windows позволяет организовать слежение за изменениями объектов файловой системы. Для этого служит функция

**function** FindFirstChangeNotification(  
 lpPathName: PChar; // Путь к папке, изменения в которой нас   
 // интересуют  
 bWatchSubtree: BOOL; // Задает необходимость слежения за   
 // изменениями во вложенных папках  
 dwNotifyFilter: DWORD // Фильтр событий  
): THandle; **stdcall**;

Параметр dwNotifyFilter – это битовая маска из одного или нескольких следующих значений:

|  |  |
| --- | --- |
| FILE\_NOTIFY\_CHANGE\_FILE\_NAME | |
|  | Слежение ведется за любым изменением имени файла, в т.ч. созданием и удалением файлов |
| FILE\_NOTIFY\_CHANGE\_DIR\_NAME | |
|  | Слежение ведется за любым изменением имени папки, в т.ч. созданием и удалением папок |
| FILE\_NOTIFY\_CHANGE\_ATTRIBUTES | |
|  | Слежение ведется за любым изменением аттрибутов |
| FILE\_NOTIFY\_CHANGE\_SIZE | |
|  | Слежение ведется за изменением размера файлов. Изменение размера происходит при записи в файл. Функция ожидания срабатывает только после успешного сброса дискового кэша |
| FILE\_NOTIFY\_CHANGE\_LAST\_WRITE | |
|  | Слежение за изменением времени последней записи в файл, т.е., фактически, за любой записью в файл. Функция ожидания срабатывает только после успешного сброса дискового кэша. |
| FILE\_NOTIFY\_CHANGE\_SECURITY | |
|  | Слежение за любыми изменениями дескрипторов защиты |

Идентификатор, возвращенный этой функцией, может использоваться в любой функции ожидания. Он переходит в сигнальное состояние, когда в папке происходят запрошенные для слежения изменения. Продолжить слежение можно, используя функцию:

**function** FindNextChangeNotification(  
 hChangeHandle: THandle  
): BOOL; **stdcall**;

По завершении работы, идентификатор должен быть закрыт при помощи функции:

**function** FindCloseChangeNotification(  
 hChangeHandle: THandle  
): BOOL; **stdcall**;

Чтобы не блокировать исполнение основного потока программы функцией ожидания, удобно реализовать ожидание изменений в отдельном потоке. Реализуем поток на базе класса TThread. Для того чтобы можно было прервать исполнение потока методом Terminate необходимо, чтобы функция ожидания, реализованная в методе Execute, также прерывалась при вызове Terminate. Для этого будем использовать вместо WaitForSingleObject функцию WaitForMultipleObjects, и прерывать ожидание по событию (event), устанавливаемому в Terminate.

**type**  
 TCheckFolder = **class**(TThread)  
 **private** FOnChange: TNotifyEvent;  
 Handles: array[0..1] of THandle; // Идентификаторы объектов  
 // синхронизации  
 **procedure** DoOnChange;  
 **protected**  
 **procedure** Execute; **override**;  
 **public**  
 **constructor** Create(CreateSuspended: Boolean;  
 PathToMonitor: String; WaitSubTree: Boolean;  
 OnChange: TNotifyEvent; NotifyFilter: DWORD);  
 **destructor** Destroy; **override**;  
 **procedure** Terminate;  
 **end**;

**procedure** TCheckFolder.DoOnChange;  
// Эта процедура вызывается в контексте главного потока приложения  
// В ней можно использовать вызовы VCL, изменять состояние формы,  
// например перечитать содержимое TListBox, отображающего файлы  
**begin**  
 **if** Assigned(FOnChange) **then**  
 FOnChange(Self);  
**end**;

**procedure** TCheckFolder.Terminate;  
**begin**  
 **inherited**; // Вызываем TThread.Terminate, устанавливаем   
 // Terminated = TRUE  
 SetEvent(Handles[1]); // Сигнализируем о необходимости  
 // прервать ожидание  
**end**;

**constructor** TCheckFolder.Create(CreateSuspended: Boolean;  
 PathToMonitor: String; WaitSubTree: Boolean;  
 OnChange: TNotifyEvent; NotifyFilter: DWORD);  
**var**  
 BoolForWin95: Integer;  
**begin** // Создаемпоток остановленным **inherited** Create(TRUE);  
 // Windows 95 содержит не очень корректную реализацию функции  
 // FindFirstChangeNotification. Для корректной работы, необходимо,  
 // чтобы:  
 // - lpPathName - не содержал завершающего слэша "\" для  
 // некорневого каталога  
 // - bWatchSubtree - TRUE должен передаваться как BOOL(1)  
 **if** WaitSubTree **then**  
 BoolForWin95 := 1  
 **else**  
 BoolForWin95 := 0;  
 **if** (Length(PathToMonitor) > 1) **and**  
 (PathToMonitor[Length(PathToMonitor)] = ‘\’) **and**  
 (PathToMonitor[Length(PathToMonitor)-1] <> ‘:’) **then**  
 Delete(PathToMonitor, Length(PathToMonitor), 1);  
 Handles[0] := FindFirstChangeNotification(  
 PChar(PathToMonitor), BOOL(BoolForWin95), NotifyFilter);  
 Handles[1] := CreateEvent(**NIL**, TRUE, FALSE, **NIL**);  
 FOnChange := OnChange;  
 // И, при необходимости, запускаем  
 **if not** CreateSuspended **then**  
 Resume;  
**end**;

**destructor** TCheckFolder.Destroy;  
**begin**  
 FindCloseChangeNotification(Handles[0]);  
 CloseHandle(Handles[1]);   
 **inherited**;  
**end**;

**procedure** TCheckFolder.Execute;  
**var**  Reason: Integer;  
 Dummy: Integer;  
**begin  
 repeat** // Ожидаем изменения в папке, либо сигнала о завершении   
 // потока  
 Reason := WaitForMultipleObjects(2, @Handles, FALSE, INFINITE);  
 **if** Reason = WAIT\_OBJECT\_0 **then begin** // Изменилась папка, вызываем обработчик в контексте  
 // главного потока приложения  
 Synchronize(DoOnChange);  
 // И продолжаем поиск  
 FindNextChangeNotification(Handles[0]);  
 **end**;  
 **until** Terminated;  
**end**;

Поскольку метод TThread.Terminate не виртуальный, этот класс нельзя использовать с переменной типа TThread, т.к. в этом случае будет вызываться Terminate от TThread, который не может прервать ожидания, и поток будет выполняться до изменения в папке, за которой ведется слежение.

#### Устройство стандартного ввода с консоли (console input)

Идентификатор, стандартного устройства ввода с консоли, полученный при помощи вызова функции GetStdHandle(STD\_INPUT\_HANDLE), можно использовать в функциях ожидания. Он находится в сигнальном состоянии, если очередь ввода консоли непустая и в несигнальном, если пустая. Это позволяет организовать ожидание ввода символов, либо, при помощи функции WaitForMultipleObjects совместить его с ожиданием каких-то других событий.

#### Задание (Job)

Job – это новый механизм Windows 2000, позволяющий объединить группу процессов в одно задание и манипулировать ими одновременно. Идентификатор задания находится в сигнальном состоянии, если все процессы, ассоциированные с ним завершились по причине истечения лимита времени на выполнение задания.

#### Процесс (Process)

Идентификатор процесса, полученный при помощи функции CreateProcess, переходит в сигнальное состояние по завершении процесса. Это позволяет организовать ожидание завершения процесса, например, при запуске из приложения внешней программы.

**var**  
 PI: TProcessInformation;  
 SI: TStartupInfo;  
  
...  
  
 FillChar(SI, SizeOf(SI), 0);  
 SI.cb := SizeOf(SI);  
 Win32Check(CreateProcess(**NIL**, 'COMMAND.COM', **NIL**,  
 **NIL**, FALSE, 0, **NIL**, **NIL**, SI, PI));  
 // Задерживаем исполнение программы до завершения процесса  
 WaitForSingleObject(PI.hProcess, INFINITE);  
 CloseHandle(PI.hProcess);  
 CloseHandle(PI.hThread);

Следует понимать, что в этом случае вызывающий процесс будет заморожен полностью и не сможет обрабатывать сообщения. Поэтому, если дочерний процесс может выполняться в течение длительного времени, лучше использовать более корректный вариант ожидания, описанный в разделе, посвященном функции MsgWaitForMultipleObjects.

#### Поток (thread)

Идентификатор потока находится в несигнальном состоянии до тех пор, пока поток выполняется. По его завершении идентификатор переходит в сигнальное состояние. Это позволяет легко узнать, завершился ли поток, либо при помощи функции, ожидающей нескольких объектов, организовать ожидание завершения одного из, либо всех интересующих потоков.

### Дополнительные механизмы синхронизации

#### Критические секции

Критические секции – это механизм, предназначенный для синхронизации потоков внутри одного процесса. Как и мутекс, критическая секция может в один момент времени принадлежать только одному потоку, однако, она предоставляет более быстрый и эффективный механизм, чем мутексы. Перед использованием критической секции необходимо инициализировать её функцией:

**procedure** InitializeCriticalSection(  
 **var** lpCriticalSection: TRTLCriticalSection  
); **stdcall**;

После создания объекта поток, перед доступом к защищаемому ресурсу должен вызвать функцию:

**procedure** EnterCriticalSection(  
 **var** lpCriticalSection: TRTLCriticalSection  
); **stdcall**;

Если в этот момент ни один из потоков в процессе не владеет объектом, то поток становится владельцем критической секции и продолжает выполнение. Если секция уже захвачена другим потоком то выполнение потока, вызвавшего функцию приостанавливается до её освобождения.

Поток, владеющий критической секцией, может повторно вызывать функцию EnterCriticalSection без блокирования своего исполнения. По завершению работы с защищаемым ресурсом поток должен вызвать функцию:

**procedure** LeaveCriticalSection(  
 **var** lpCriticalSection: TRTLCriticalSection  
); **stdcall**;

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

Имеется возможность попытаться захватить объект без замораживания потока. Для этого служит функция:

**function** TryEnterCriticalSection(  
 **var** lpCriticalSection: TRTLCriticalSection  
): BOOL; **stdcall**;

Она проверяет, захвачена секция ли в момент её вызова. Если да – функция возвращает FALSE, в противном случае – захватывает секцию и возвращает TRUE.

По завершении работы с критической секцией, она должна быть уничтожена вызовом функции:

**procedure** DeleteCriticalSection(  
 **var** lpCriticalSection: TRTLCriticalSection  
); **stdcall**;

Рассмотрим пример приложения, осуществляющего в нескольких потоках загрузку данных по сети. Глобальные переменные BytesSummary и TimeSummary хранят общее количество загруженных байт и время загрузки. Эти переменные каждый поток обновляет по мере считывания данных. Для предотвращения конфликтов приложение должно защитить общий ресурс при помощи критической секции.

**var**  
 // Глобальные переменные  
 CriticalSection: TRTLCriticalSection;  
 BytesSummary: Cardinal;  
 TimeSummary: TDateTime;  
 AverageSpeed: Float;  
  
...  
  
// При инициализации приложения  
InitializeCriticalSection(CriticalSection);  
BytesSummary := 0;  
TimeSummary := 0;  
AverageSpeed := 0;  
  
  
//В методе Execute потока, загружающего данные.  
**repeat**  
 BytesRead := ReadDataBlockFromNetwork;  
 EnterCriticalSection(CriticalSection);  
 **try** BytesSummary := BytesSummary + BytesRead;  
 TimeSummary := TimeSummary + (Now - ThreadStartTime);  
 **if** TimeSummary > 0 **then**  
 AverageSpeed := BytesSummary / (TimeSummary/24/60/60);  
 **finally**  
 LeaveCriticalSection(CriticalSection)  
 **end**;  
**until** LoadComplete;  
  
// При завершении приложения  
DeleteCriticalSection(CriticalSection);

Delphi предоставляет класс, инкапсулирующий функциональность критической секции. Класс объявлен в модуле SyncObjs.pas

**type**  
 TCriticalSection = **class**(TSynchroObject)  
 **public**  
 **constructor** Create;  
 **destructor** Destroy; **override**;  
 **procedure** Acquire; **override**;  
 **procedure** Release; **override**;  
 **procedure** Enter;  
 **procedure** Leave;  
 **end**;

Методы Enter и Leave являются синонимами методов Acquire и Release соответственно и добавлены для лучшей читаемости исходного кода.

**procedure** TCriticalSection.Enter;  
**begin**  
 Acquire;  
**end**;

**procedure** TCriticalSection.Leave;  
**begin**  
 Release;  
**end**;

#### Защищенный доступ к переменным (Interlocked Variable Access)

Часто возникает необходимость совершения операций над разделяемыми между потоками 32-разрядными переменными. Для упрощения решения этой задачи WinAPI предоставляет функции для защищенного доступа к ним, не требующие использования дополнительных (и более сложных) механизмов синхронизации. Переменные, используемые в этих функциях, должны быть выровнены на границу 32-разрядного слова. Применительно к Delphi это означает, что если переменная объявлена внутри записи (record), то эта запись не должна быть упакованной (packed) и при её объявлении должна быть активна директива компилятора {$A+}. Несоблюдение этого требования может привести к возникновению ошибок на многопроцессорных конфигурациях.

**type**  
 TPackedRecord = **packed record**  
 A: Byte;  
 B: Integer;  
 **end**;   
// TPackedRecord.B нельзя использовать в функциях InterlockedXXX  
  
 TNotPackedRecord = **record**  
 A: Byte;  
 B: Integer;  
 **end**;

{$A-}  
**var** A1: TNotPackedRecord;  
// A1.B нельзя использовать в функциях InterlockedXXX  
 I: Integer  
// I можно использовать в функциях InterlockedXXX, т.к. переменные в  
// Delphi всегда выравниваются на границу слова безотносительно  
// к состоянию директивы компилятора $A

{$A+}  
**var** A2: TNotPackedRecord;  
// A2.B можно использовать в функциях InterlockedXXX

**function** InterlockedIncrement(  
 **var** Addend: Integer  
): Integer; **stdcall**;

Функция увеличивает переменную Addend на 1. Возвращаемое значение зависит от операционной системы:

|  |  |
| --- | --- |
| Windows 98, Windows NT 4.0 и старше | |
|  | Возвращается новое значение переменной Addend |
| Windows 95, Windows NT 3.51 | |
|  | Если после изменения Addend < 0 возвращается отрицательное число, не обязательно равное Addend  Если Addend = 0 – возвращается 0  Если после изменения Addend > 0 возвращается положительное число, не обязательно равное Addend. |

**function** InterlockedDecrement(  
 **var** Addend: Integer  
): Integer; **stdcall**;

Функция уменьшает переменную Addend на 1. Возвращаемое значение аналогично функции InterlockedIncrement.

**function** InterlockedExchange(  
 **var** Target: Integer;   
 Value: Integer  
): Integer; **stdcall**;

Функция записывает в переменную Target значение Value и возвращает предыдущее значение Target

Следующие функции для выполнения требуют Windows 98 или Windows NT 4.0 и старше.

**function** InterlockedCompareExchange(  
 **var** Destination: Pointer;  
 Exchange: Pointer;  
 Comperand: Pointer  
): Pointer; **stdcall**;

Функция сравнивает значения Destination и Comperand. Если они совпадают, значение Exchange записывается в Destination. Функция возвращает начальное значение Destination.

**function** InterlockedExchangeAdd(  
 Addend: PLongint;   
 Value: Longint  
): Longint; **stdcall**;

Функция добавляет к переменной, на которую указывает Addend значение Value и возвращает начальное значение Addend.

## Резюме

Многозадачная и многопоточная среда Win32 предоставляет широкие возможности для написания высокоэффективных приложений. Однако, написание приложений, использующих многопоточность и взаимодействующих друг с другом, при неаккуратном программировании может привести к их неверной работе, неоправданной загрузке и даже блокировке всей системы. Во избежание этого следуйте нижеприведенным рекомендациям:

* Если приложения или потоки одного процесса изменяют общий ресурс – защищайте доступ к нему при помощи критических секций или мутексов.
* Если доступ осуществляется только на чтение – защищать ресурс не обязательно.
* Критические секции более эффективны, но применимы только внутри одного процесса, мутексы могут использоваться для синхронизации между процессами.
* Используйте семафоры для ограничения количества обращений к одному ресурсу.
* Используйте события (event) для информирования потока о наступлении какого-либо события.
* Если разделяемый ресурс – 32-битная переменная – для синхронизации доступа к нему можно использовать функции, обеспечивающие разделяемый доступ к переменным.
* Многие объекты Win32 позволяют организовать эффективное слежение за своим состоянием при помощи функций ожидания. Это наиболее эффективный с точки зрения расхода системных ресурсов метод.
* Если Ваш поток создает (даже неявно, при помощи CoInitialize или функций DDE) окна – он должен обрабатывать сообщения. Не используйте в таком потоке функций не позволяющих прервать ожидание по приходу сообщения с большим или неограниченным периодом ожидания. Используйте функции MsgWaitForXXX