1.1. Управление процессами в Windows

В отличие от «полуторазадачной» MS-DOS, которая оставляет прикладному программисту всю работу (и весь риск) организации параллельного функционирования процессов, многозадачные ОС предоставляют программисту более или менее удобный и богатый набор системных функций, позволяющих запустить несколько параллельных процессов и организовать их взаимодействие (синхронизацию процессов, обмен данными, взаимное исключение и т.п.). При этом ОС обязана гарантировать корректную и эффективную организацию переключения процессов, разделения между ними процессорного времени, памяти и других ресурсов.

Сложность проблемы организации взаимодействия параллельных процессов существенно разная для систем, использующих вытесняющую и невытесняющую диспетчеризацию процессов. При вытесняющей диспетчеризации процесс может быть прерван диспетчером практически в любой момент. Помимо задачи сохранения и последующего восстановления контекста процесса (см. п. **Ошибка! Источник ссылки не найден.**), которая должна решаться самой ОС, возникают еще и задачи обеспечения взаимного исключения при выполнении критических секций в многозадачных приложениях. Только разработчик программы может решить, какие части текста его программы являются критическими секциями и должны быть защищены семафорами.

В системе с невытесняющей диспетчеризацией программисту достаточно проверить, что критические секции не содержат вызовов блокирующих и вытесняющих функций. При этом можно гарантировать, что в ходе выполнения критической секции не произойдет переключения процессов.

Все версии Windows от 1.0 до 3.11 представляли собой достаточно мощные многозадачные системы с невытесняющей диспетчеризацией. Версии, начиная с Windows NT и Windows 95, используют вытесняющую диспетчеризацию.

1.1.1. Понятие объекта в Windows

В ОС Windows широко используется понятие системного ***объекта***. По сути, любой объект представляет собой некоторую структуру данных, расположенную в адресном пространстве системы. Поскольку приложения не могут иметь доступа к этой памяти, то для работы с объектом приложение должно получить ***хэндл*** объекта – некоторое условное число, которое будет представлять данный объект при обращении к API-функциям. Процесс получает хэндл, как правило, при вызове функции **CreateXxx** (здесь **Xxx** – название объекта), которая может либо создать новый объект, либо открыть существующий объект, созданный другим процессом. Функции вида **OpenXxx** позволяют только открыть существующий объект.

Объекты Windows делятся на объекты ядра (KERNEL), позволяющие управлять процессами, объекты USER, описывающие работу с окнами, и объекты GDI, задающие графические ресурсы Windows. В данном курсе рассматриваются только объекты ядра. Процессы, нити и открытые файлы являются примерами объектов ядра.

Одной из отличительных особенностей объектов ядра являются ***атрибуты защиты***, которые можно указать при создании объекта. Эти атрибуты определяют права доступа к объекту для различных пользователей и групп. Кроме того, при создании объекта ядра

можно задать его имя, которое используется для того, чтобы другие процессы могли открыть тот же объект, зная его имя.

Хэндл объекта может быть использован только тем процессом, который создал или открыл этот объект. Нельзя просто переслать значение хэндла другому процессу, оно не будет действовать в другом контексте. Имеется, однако, функция **DuplicateHandle**, которая создает корректную копию хэндла, требуя указать для этого, какой процесс создает копию, какого именно хэндла и для какого процесса предназначена копия.

Если хэндл объекта больше не нужен данному процессу, его следует закрыть с помощью функции **CloseHandle**, общей для разных типов объектов.

Объект существует до тех пор, пока не будут закрыты все хэндлы, указывающие на него.

1.1.2. Процессы и нити

Общее понятие процесса, рассмотренное выше в п. **Ошибка! Источник ссылки не найден.**, для ОС Windows как бы распадается на два понятия: собственно процесса и ***нити*** (thread; в некоторых книгах используется термин ***поток***). При этом нить является единицей работы, она участвует в конкуренции за процессорное время, изменяет свое состояние и приоритет, как было описано выше для процесса. Что же касается процесса в Windows, то он может состоять из нескольких нитей, использующих общую память, открытые файлы и другие ресурсы, принадлежащие процессу. В двух словах: процесс – *владеет* (памятью, файлами), нити – *работают*, при этом совместно используя ресурсы своего процесса. Правда, нить тоже кое-чем владеет: окнами, очередью сообщений, стеком.

Процесс создается при запуске программы (EXE-файла). Одновременно создается одна нить процесса (должен же кто-то работать!). Создание процесса выполняется с помощью API-функции **CreateProcess**. Основными параметрами при вызове этой функции являются следующие.

- Имя файла запускаемой программы.
- Командная строка, передаваемая процессу при запуске.
- Атрибуты защиты для создаваемых процесса и нити. И процесс, и нить являются объектами ядра Windows и в этом качестве могут быть защищены от несанкционированного доступа (например, от попыток других процессов вмешаться в работу данного процесса).
- Различные флаги, уточняющие режим создания процесса. Среди них следует отметить класс приоритета процесса, флаг отладочного режима (при этом система будет уведомлять процесс-родитель о действиях порожденного процесса), а также флаг создания приостановленного процесса, который не начнет работать, пока не будет вызвана функция возобновления работы.
- Блок среды процесса.
- Текущий каталог процесса.
- Параметры первого окна, которое будет открыто при запуске процесса.
- Адрес блока информации, через который функция возвращает родительскому процессу четыре числа: идентификатор созданного процесса, идентификатор нити, хэндл процесса и хэндл нити.

Если процесс успешно создан, функция **CreateProcess** возвращает ненулевое значение.

Класс приоритета процесса используется при определении приоритетов его нитей. Подробнее об этом в п. 1.1.3.

Хэндл объекта ядра Windows (в данном случае процесса или нити) позволяет выполнять различные операции с этим объектом. Подробнее о хэндлах и идентификаторах см. п. 1.1.4.

После создания процесса его единственная нить начинает выполнять программу процесса, работая параллельно с нитями других запущенных процессов. Если логика работы программы предполагает параллельное выполнение каких-либо действий в рамках одного процесса, то могут быть созданы дополнительные нити. Для этого используется функция **CreateThread**. Ее основные параметры следующие:

- атрибуты защиты для создаваемой нити;
- размер стека нити;
- стартовый адрес нити (обычно нить связывается с выполнением одной из функций, описанных в программе процесса, при этом в качестве стартового адреса указывается имя функции);
- параметр-указатель, позволяющий передать нити при запуске некоторое значение в качестве аргумента;
- флаг создания нити в приостановленном состоянии;
- указатель на переменную, в которой функция должна возвратить идентификатор созданной нити.

Возвращаемым значением функции **CreateThread** является хэндл созданной нити либо **NULL**, если создать нить не удалось.

Прекрасным примером многонитевой программы является Microsoft Word. В то время как основная нить обрабатывает ввод с клавиатуры, отдельная нить может динамически рассчитывать разбиение текста на страницы, еще одна нить может в это же время выполнять печать документа или его сохранение.

Для завершения работы нити используется вызов функции **ExitThread**. Для завершения работы всего процесса любая из его нитей может вызвать функцию **ExitProcess**. Единственным параметром каждой из этих функций является код завершения нити или процесса.

Завершение процесса приводит к освобождению всех ресурсов, которыми владел процесс: памяти, открытых файлов и т.п.

При завершении процесса завершаются все его нити. И наоборот, при завершении последней нити процесса завершается и сам процесс.

Не слишком широко известно, что нить не является самой мелкой единицей организации вычислений. На самом деле Windows позволяет создать внутри нити несколько ***волокон*** (fiber), которые в обычной терминологии могут быть описаны как сопрограммы или как задачи с невытесняющей диспетчеризацией, работающие в рамках одной и той же задачи с вытесняющей диспетчеризацией. Переключение волокон

выполняется только явно, с помощью функции **SwitchToFiber**. Об использовании сопрограмм см. /**Ошибка! Источник ссылки не найден.**/.

1.1.3. Планировщик Windows

Задачей планировщика является выбор очередной нити для выполнения. Планировщик вызывается в трех случаях:

- если истекает квант времени, выделенный текущей нити;
- если текущая нить вызвала блокирующую функцию (например, **WaitForMultipleObjects** или **ReadFile**) и перешла в состояние ожидания;
- если нить с более высоким приоритетом пробудилась от ожидания или была только что запущена.

Для выбора нити используется алгоритм приоритетной очереди. Для каждого уровня приоритета система ведет очередь активных нитей (т.е. нитей, находящихся в состоянии готовности). Для выполнения выбирается очередная нить из непустой очереди с самым высоким приоритетом.

Очереди активных нитей пополняются за счет нитей, проснувшихся после состояния ожидания и нитей, вытесненных планировщиком. Как правило, нить помещается в конец очереди. Исключение делается для нити, вытесненной до истечения ее кванта времени более приоритетной нитью. Такая «обиженная» нить ставится в голову очереди.

Значение кванта времени для серверных установок Windows равно обычно 120 мс, для рабочих станций – 20 мс.

Как вы думаете, почему для серверов квант времени больше?

Теперь подробнее о приоритетах. Хотя общая схема их назначения одинакова для всех версий Windows, детали могут разниться. Дальнейшее изложение ориентировано на Windows NT 4.0.

Все уровни приоритета нитей пронумерованы от 0 (самый низкий приоритет) до 31 (самый высокий). Уровни от 16 до 31 называются ***приоритетами реального времени***, они предназначены для выполнения критичных по времени системных операций. Только сама система или пользователь с правами администратора могут использовать приоритеты из этой группы. Уровни от 0 до 15 называются ***динамическими приоритетами***.

В Windows используется двухступенчатая схема назначения приоритетов. При создании процесса ему назначается (а впоследствии может быть изменен самой программой или пользователем) один из четырех ***классов приоритета***, с каждым из которых связано базовое значение приоритета:

- ***Realtime*** (базовый приоритет 24) – высший класс приоритета, допустимый только для системных процессов, занимающих процессор на очень короткое время;
- ***High*** (базовый приоритет 13) – класс высокоприоритетных процессов;
- ***Normal*** (базовый приоритет 8) – обычный класс приоритета, к которому относится большая часть запускаемых прикладных процессов;
- ***Idle*** (базовый приоритет 4) – низший (буквально – «холостой» или «простаивающий») класс приоритета, характерный для экранных заставок, мониторов производительности и других программ, которые не должны мешать жить более важным программам.

Собственно, приоритет связывается не с процессом, а с каждой его нитью. Приоритет нити определяется базовым приоритетом процесса, к которому прибавляется относительный приоритет нити – величина от –2 до +2. Относительный приоритет назначается нити при ее создании и может при необходимости изменяться. Имеется также возможность назначить нити критический приоритет (31 для процессов реального времени, 15 для остальных) или холостой приоритет (16 для процессов реального времени, 0 для остальных).

Для нитей процессов реального времени приоритеты являются статическими в том смысле, что система не пытается их изменять по своему усмотрению. Предполагается, что для этой группы процессов соотношение приоритетов должно быть именно таким, как задумал программист.

Для процессов трех низших классов их приоритеты не случайно называются динамическими. Планировщик может изменять приоритеты, а заодно и кванты времени для нитей в ходе их выполнения, чтобы достичь более справедливого распределения процессорного времени. Правила изменения динамических приоритетов следующие.

- Когда заблокированная нить дождалась нужного ей события, к приоритету нити прибавляется величина, зависящая от причины ожидания. Эта прибавка может достигать 6 единиц (но приоритет не должен превысить 15), если нить разблокирована вследствие нажатия клавиши или кнопки мыши. Таким способом система стремится уменьшить время реакции на действия пользователя. Всякий раз, когда нить полностью использует свой квант времени, прибавка уменьшается на 1, пока приоритет нити не вернется к своему заданному значению.
- Если нить владеет окном переднего плана (т.е. тем, с которым работает пользователь), то ради уменьшения времени реакции планировщик может увеличить квант времени для этой нити с 20 мс до 40 или 60 мс, в зависимости от настроек системы.
- Если планировщик обнаруживает, что некоторая нить пребывает в очереди более 3 с, то он повышает ее приоритет аж до 15 и удваивает ее квант. Но эта благотворительность разовая: когда Золушка-нить израсходует увеличенный квант или заблокируется, ее приоритет и квант возвращаются к прежним значениям. Смысл акции понятен: система пытается обеспечить хоть какое-то продвижение даже для низкоприоритетных нитей.

1.1.4. Процесс и нить как объекты

Подсистема управления процессами в Windows позволяет рассматривать процессы и нити как объекты, над которыми нити различных процессов могут выполнить целый ряд действий. Для этого требуется иметь хэндл процесса или нити. Как нам известно, такие хэндлы возвращаются как побочные результаты работы функций **CreateProcess** и **CreateThread**. Это позволяет процессу, запустившему другой процесс или нить, выполнять с ними необходимые действия. Однако в некоторых случаях желательно дать возможность управлять процессом не процессу-«родителю», а другому, совершенно постороннему процессу (например, программе администрирования системы). Проблема при этом в том, что, как уже отмечалось, хэндл в Windows имеет смысл только в пределах того процесса, который вызвал функцию **Create** и не может быть просто передан другому процессу.

Функция **OpenProcess** позволяет одному процессу получить хэндл любого другого процесса, указав для этого два параметра: идентификатор интересующего процесса и маску,

задающую набор запрашиваемых прав по отношению к этому процессу. Если подсистема безопасности Windows, сверив маркер доступа запрашивающего процесса с дескриптором безопасности процесса-объекта (см. п. **Ошибка! Источник ссылки не найден.**), сочтет запрос законным, то функция вернет требуемый хэндл.

А о каких, собственно, правах идет речь? Что именно один процесс может сделать с другим процессом? Основные права следующие:

- запускать новую нить процесса;
- запрашивать и изменять класс приоритета процесса;
- читать и записывать данные в памяти процесса;
- использовать процесс в качестве объекта синхронизации;
- принудительно прекращать выполнение процесса;
- запрашивать код завершения процесса.

Все перечисленные действия выполняются с помощью соответствующих API-функций (например, для прекращения процесса вызывается функция **TerminateProcess**), одним из параметров которых указывается хэндл открытого процесса-объекта.

1.1.5. Синхронизация нитей

1.1.5.1. Способы синхронизации

Windows предоставляет разнообразные возможности для синхронизации работы нитей, т.е., иначе говоря, для организации пассивного ожидания нитью некоторых событий, связанных с работой других нитей того же процесса или других процессов.

- Традиционной для Windows формой синхронизации является обмен ***сообщениями*** (messages). При этом, механизм сообщений предназначен не только для синхронизации, но и для обмена данными. Далее работа с сообщениями будет рассмотрена подробно.
- Разнообразные условия ожидания могут быть реализованы с помощью ***объектов синхронизации*** и ***функций ожидания***, которые позволяют заблокировать нить до момента перехода указанного объекта в сигнальное состояние.
- Использование переменных типа **CRITICAL_SECTION**, в отличие от предыдущих способов, возможно только для синхронизации нитей одного и того же процесса, но зато реализуется более эффективно по времени и памяти.

1.1.5.2. Объекты синхронизации и функции ожидания

Объекты синхронизации представляют собой частный случай объектов ядра Windows. Отличительной особенностью объектов синхронизации является возможность использовать в качестве аргументов функций ожидания.

Для создания объекта нужно вызвать функцию **Create***Xxx*, где вместо *Xxx* указывается название конкретного типа объектов, например, **CreateMutex** или **CreateEvent**.

Общей чертой всех объектов синхронизации является наличие двух состояний, одно из которых называется ***сигнальным*** (signaled), а другое, соответственно, несигнальным. Смысл сигнального состояния различен для разных типов объектов, общим же является то, что состояние ожидания для нити заканчивается тогда, когда объект переходит в сигнальное состояние.

Для конкретных типов объектов вместо термина «сигнальное состояние» может быть удобно использовать как синонимы «свободно», «включено», «имеется», «произошло» и т.п., а для несигнального, соответственно, «занято», «выключено», «нет» и т.п.

Поскольку объекты существуют в системной памяти, процессы могут получить к ним доступ только с помощью хэндлов. Это требует затрат времени на переключение контекста памяти при работе с объектами, но зато дает возможность синхронизации нитей, принадлежащих разным процессам. Для этого при создании объектов ядра одним из параметров функции **Create***Xxx* может указываться имя этого объекта. Если после этого другой процесс попытается создать объект того же типа и с тем же именем, то он получит хэндл уже существующего объекта. В результате оба процесса будут работать с одним и тем же объектом синхронизации.

К функциям ожидания относятся **WaitForSingleObject** (ожидание одного объекта) и **WaitForMultipleObjects** (ожидание нескольких объектов). Рассмотрим сначала первую из них.

Функция **WaitForSingleObject** имеет два аргумента:

- хэндл какого-либо объекта синхронизации;
- тайм-аут ожидания, т.е. максимальное время ожидания (в миллисекундах).

Эта функция – блокирующая: если в момент ее вызова указанный объект находится в несигнальном состоянии, то вызвавшая нить переводится в состояние ожидания (разумеется, пассивного!). Ожидание, как правило, завершается в одном из двух случаев: либо объект переходит в сигнальное состояние, либо истекает заданный тайм-аут. Третий, особый случай завершения связан с понятием «заброшенный мьютекс», которое будет пояснено ниже. Значение, возвращаемое функцией, позволяет узнать, какая из трех причин пробуждения нити имела место.

Если в момент вызова функции указанный объект уже находился в сигнальном состоянии, то ожидания не происходит, функция сразу же возвращает результат.

Если задана нулевая величина тайм-аута, то выполнение функции сводится к проверке, находится ли в данный момент объект в сигнальном состоянии.

Если значение тайм-аута равно константе **INFINITE**, то время ожидания не ограничено, т.е. нить может пробудиться только по сигнальному состоянию объекта.

Функция **WaitForMultipleObjects** отличается тем, что вместо единственного хэндла принимает в качестве аргументов адрес массива, содержащего несколько хэндлов, и количество этих хэндлов (размер массива). Как и в предыдущем случае, задается величина тайм-аута. Дополнительный, четвертый аргумент – булевский флаг режима ожидания. Значение **FALSE** означает ожидание *любого* объекта (достаточно, чтобы хоть один из них перешел в сигнальное состояние), **TRUE** – ожидание *всех* объектов (ожидание завершится, только когда все объекты окажутся в сигнальном состоянии).

Возвращаемое значение позволяет определить причину пробуждения:

- переход одного из заданных объектов в сигнальное состояние, и какого именно объекта;
- переход одного из объектов-мьютексов в «заброшенное» состояние (будет пояснено ниже), и какого именно объекта;
- истечение тайм-аута ожидания.

Кроме двух названных, имеется еще ряд функций ожидания. Функции **WaitForSingleObjectEx** и **WaitForMultipleObjectsEx** позволяют также дожидаться завершения асинхронных операций ввода/вывода для указанного файла. Функция **MsgWaitForMultipleObjects** позволяет, наряду с объектами синхронизации, использовать для пробуждения появление сообщений, ожидающих обработки. Функция **SignalObjectAndWait** выполняет редкостный трюк: она переводит один объект в сигнальное состояние и тут же начинает дожидаться другого объекта.

1.1.5.3. Типы объектов синхронизации

Список объектов синхронизации включает следующие типы объектов:

- процессы;
- нити;
- события (в узком смысле);
- мьютексы (двоичные семафоры);
- семафоры (недвоичные);
- таймеры;
- уведомления об изменениях в каталоге;
- консольный ввод.

Рассмотрение объектов синхронизации проще всего начать с процессов и нитей. Для тех и других сигнальным состоянием является только состояние завершения работы. Другими словами, можно заставить нить одного процесса ожидать завершения работы другой нити или процесса.

Простейшим из объектов, предназначенных только для синхронизации, является ***событие*** (event). Он создается при помощи функции **CreateEvent**, которая требует указать вид события (с автосбросом или с ручным сбросом), а также его начальное состояние – сигнальное или несигнальное. Переход в сигнальное состояние выполняется функцией **SetEvent**, а в несигнальное – **ResetEvent**. Функция **PulseEvent** как бы совмещает обе предыдущие: сигнальное состояние включается, функция ожидания, если она была вызвана ранее, завершается, но сигнальное состояние тут же сбрасывается.

Событие с ***автосбросом*** означает, что функция ожидания при завершении ожидания сбрасывает то событие (или события), которого (которых) она дождалась. Если несколько нитей ждут одного и того же события с автосбросом, то разблокирована будет только одна из них (Windows не гарантирует, какая именно), поскольку остальным «события не достанется». Напротив, событие ***с ручным сбросом*** остается в сигнальном состоянии и может быть сброшено только функцией **ResetEvent**, а до тех пор оно может «накормить» сколько угодно ожидающих нитей. Можно сказать, что событие с ручным сбросом больше похоже не на *событие*, а на *состояние*.

Следующий тип объектов – ***мьютекс*** (mutex). Это почти в чистом виде двоичный семафор в смысле Дейкстры. Мьютекс может быть либо свободен (это его сигнальное состояние), либо захвачен какой-либо нитью. При создании мьютекса в функции **CreateMutex** указывается его начальное состояние. Допускается многократный захват мьютекса *одной и той же* нитью, но после этого нить должна столько же раз освободить мьютекс, чтобы он вернулся в свободное состояние.

Для освобождения мьютекса нить, владеющая им, вызывает функцию **ReleaseMutex**, которая является аналогом дейкстровской V(S). Никакая другая нить, кроме владельца, не может освободить «чужой» мьютекс.

В специальной функции для захвата мьютекса нет необходимости, поскольку работу примитива P(S) с успехом выполняют функции ожидания.

Если нить, владеющая мьютексом, завершает свою работу, не освободив мьютекс, то он переходит в «заброшенное» состояние: теперь уже никто не может его освободить. Такую ситуацию следует считать признаком неаккуратного программирования, и она может быть признаком наличия в программе более серьезных ошибок синхронизации.

Объект типа ***семафор*** (semaphore) представляет собой недвоичный семафор, используемый обычно для управления распределением ограниченного числа единиц ресурса. В функции **CreateSemaphore** указываются два числа: максимальное значение счетчика, связанного с семафором (сколько всего имеется единиц ресурса), и начальное значение этого счетчика (сколько единиц свободно). Сигнальным является состояние семафора со счетчиком больше нуля. Функция ожидания уменьшает счетчик на 1, а функция **ReleaseSemaphore** увеличивает его на заданное количество единиц, но так, чтобы результат не превышал максимального значения.

В отличие от мьютекса, семафор не имеет владельца, т.е. возможна и вполне обычна ситуация, когда одна нить захватывает единицы ресурса, а другая освобождает их.

Таймер «для ожидания» (waitable timer) создается с помощью функции **CreateWaitableTimer**, после чего должен еще быть установлен функцией **SetWaitableTimer**. При создании таймера указывается его вид: с ручным сбросом или с автосбросом, аналогично объекту «событие». При вызове **SetWaitableTimer** указывается время до перехода в сигнальное состояние, в единицах по 100 нс. Время можно указать либо абсолютно (когда «зазвонить»), либо относительно текущего момента (как долго ждать). Можно также указать период (в миллисекундах), через который должен повторяться сигнал.

Таймер с ручным сбросом, перейдя в сигнальное состояние, остается в нем до вызова функции **CancelWaitableTimer** или повторного вызова **SetWaitableTimer**. Таймер с автосбросом может быть также сброшен функциями ожидания.

Полезно напомнить, что в Windows широко используется и другой тип таймеров «для сообщений», создаваемый функцией **SetTimer**, но эти таймеры не могут взаимодействовать с функциями ожидания и даже не являются объектами.

Объект типа ***уведомление об изменениях*** создается при помощи функции **FindFirstChangeNotification**. При этом указываются следующие аргументы:

- каталог файловой системы, изменения в котором должны отслеживаться;
- флаг, указывающий, должны ли отслеживаться изменения только в заданном каталоге или также во всех вложенных подкаталогах;
- фильтр отслеживаемых изменений.

Фильтр изменений представляет собой маску, в которой отдельные биты задают такие события, как изменение списка имен файлов и каталогов (это включает их создание, удаление и переименование), изменение атрибутов файла, размера, даты изменения и атрибутов защиты.

Функция возвращает хэндл объекта, который переходит в сигнальное состояние один раз, при первом появлении одного из включенных в фильтр изменений. Если требуется повторять отслеживание, следует вызывать функцию **FindNextChangeNotification**, передавая ей хэндл. Для удаления объекта нужно вызвать функцию с диким именем **FindCloseChangeNotification**.

Уведомления об изменениях позволяют избежать активного ожидания для программ, которые должны отслеживать и отображать состояние каталогов в динамике (как это делают, например, **Проводник** или **Far Manager**). При этом характер происшедшего изменения программа может выяснить, только перечитав каталог, но зато достоверно известно, что какое-то изменение было.

Хэндл объекта типа ***консольный ввод*** возвращается функцией **CreateFile**, у которой вместо имени открываемого файла указано устройство консольного ввода (**CONIN$**). Объект в сигнальном состоянии, когда в буфере ввода есть непрочитанные символы.

1.1.5.4. Критические секции

Еще одним средством синхронизации потоков служат переменные типа **CRITICAL_SECTION**. По назначению они совпадают с рассмотренными выше мьютексами, однако реализация двоичного семафора в этом случае совершенно иная. Если мьютексы – это один из типов объектов ядра, то критические секции – просто переменные. В чем здесь принципиальная разница? В том, что объекты существуют в памяти системы, поэтому прикладная программа не может напрямую обратиться к объекту, прочитать или записать его значение. Программа может только получить хэндл объекта, а затем она поручает системе выполнить то или иное действие с объектом, на который указывает данный хэндл. Такое решение позволяет использовать один объект для связи нескольких процессов, сохраняя при этом изоляцию памяти процесса от памяти системы и других процессов.

Недостаток использования объектов ядра в том, что обращение к ним всякий раз требует переключения контекста с пользовательского на системный и обратно, а это требует времени. Такова цена изоляции процессов.

Если требуется синхронизировать работу нитей *одного и того же* процесса, то более «дешевым» решением может быть использование переменных **CRITICAL_SECTION**. Программист должен описать переменную этого типа в программе процесса, при этом для всех нитей процесса будет доступен адрес этой переменной. Перед началом работы с критической секцией одна из нитей процесса должна вызвать функцию **InitializeCriticalSection**, передав ей адрес переменной. Затем он может использовать функцию **EnterCriticalSection**, чтобы занять двоичный семафор, и функцию **LeaveCriticalSection**, чтобы освободить его. Как и для мьютекса, один и тот же поток может несколько раз занять критическую секцию, но потом должен столько же раз освободить ее. Имеется еще удобная функция **TryEnterCriticalSection**, которая позволяет избежать блокировки и занимает критическую секцию, только если та была свободна в момент обращения. По истечении необходимости в синхронизации нитей следует вызвать функцию **DeleteCriticalSection**.

Вызов всех перечисленных функций выполняется системой гораздо быстрее, чем вызов функций, работающих с хэндлами объектов, поскольку вся работа выполняется в памяти вызывающего процесса, без переключения контекста памяти. Но зато невозможно

использовать критические секции для синхронизации нитей *разных* потоков, для этого нужны мьютексы.

1.1.6. Сообщения

Огромную роль в организации работы прикладных программ и самой системы Windows играют ***сообщения*** (messages). На обработке сообщений, посылаемых системой, основана вся работа с графическим интерфейсом программ. Процессы также могут обмениваться сообщениями с целью синхронизации действий и обмена данными.

Сообщение представляет собой структуру данных, содержащую тип сообщения, адресат сообщения (конкретное окно, все окна или нить процесса), время посылки сообщения, координаты курсора (для сообщений от мыши), а также два параметра, смысл которых зависит от типа сообщения.

Система посылает сообщения приложениям по самым разным поводам: при нажатии клавиш и кнопок мыши, создании и закрытии окон, изменении размеров и положения окна, выборе объекта в каком-либо списке, выборе команды в меню и т.п. Часто одно и то же действие пользователя (например, щелчок левой кнопкой мыши на пункте меню) вызывает посылку нескольких разных сообщений.

Большая часть посылаемых сообщений может обрабатываться самой системой по умолчанию. Прикладная программа может взять обработку некоторых сообщений на себя, чтобы обеспечить требуемую логикой работы реакцию на определенные события (например, при изменении размеров окна перераспределить его площадь между отдельными панелями; при попытке закрыть окно запросить подтверждение и т.п.). Программа должна также обрабатывать сообщения о посылаемых ей командах (при использовании меню, кнопок и т.п.) и сообщение о необходимости перерисовать содержимое окна.

При создании любого нового окна Windows требует задать ***оконную функцию***, которая должна обрабатывать сообщения, направленные этому окну. Нить, получившая сообщение, либо вызывает оконную функцию одного из своих окон (для выбора окна используется функция **DispatchMessage**), либо обрабатывает сообщение сама (если оно не связано с конкретным окном).

Имеется два принципиально разных способа посылки сообщения. Посылая сообщение ***синхронно***, отправитель дожидается окончания его обработки, прежде чем продолжить работу. ***Асинхронная*** посылка напоминает опускание письма в почтовый ящик, она не оказывает влияния на дальнейшую работу отправителя. Прикладная программа может посылать любые сообщения синхронным или асинхронным способом, как сочтет нужным разработчик. Для этого могут использоваться, соответственно, функция **SendMessage** (синхронная посылка) или функция **PostMessage** (асинхронная посылка).

Нить процесса получает сообщения, адресованные принадлежащим ей окнам или самой нити. Для выборки сообщений из очереди используется функция **GetMessage**. В случае отсутствия сообщений нить блокируется до их поступления. При желании программист может использовать функцию **PeekMessage**, которая проверяет наличие сообщения, не блокируя вызывающую нить.

Реализация асинхронного способа посылки достаточно проста: система просто помещает сообщение в очередь нити-получателя. Синхронная посылка в ранних,

невытесняющих версиях Windows также реализовалась очень просто: как непосредственный вызов оконной функции окна-получателя, минуя всякие очереди. После окончания обработки управление возвращалось отправителю. В современных версиях Windows все так и происходит, если нить посылает сообщение своему собственному окну. В остальных случаях такое решение невозможно. Во-первых, если сообщение пересылается между нитями разных процессов, то изоляция процессов в памяти не дает возможности одной нити вызвать другую как подпрограмму. Во-вторых, даже для нитей одного процесса такой вызов мог бы привести к непредсказуемым последствиям в случае переключения нитей по кванту времени.

В результате этого системе приходится действовать очень аккуратно. Нить, пославшая синхронное сообщение окну другой нити, блокируется. Сообщение помещается в отдельную очередь синхронных сообщений к нити-получателю. Эта нить продолжает свою нормальную работу до момента, когда она обратится за очередным сообщением, т.е. вызовет **GetMessage** или **PeekMessage**. Тогда система вызывает оконные функции нити для обработки каждого из ожидающих синхронных сообщений. Только когда все они обработаны, выполняется вызванная функция **GetMessage** или **PeekMessage**, проверяющая наличие асинхронных сообщений в очереди.

В процессе обработки синхронного сообщения оконная функция не должна вызывать функции, которые могут вызвать переключение системы на выполнение другой нити.