Процессы и потоки

Эти два понятия очень важны, и вы должны постараться их хорошенько осмыслить. Процессом (process) называется экземпляр вашей программы, загруженной в память. Этот экземпляр может создавать потоки (thread), которые представляют собой последовательность инструкций на выполнение. Важно понимать, что выполняются не процессы, а именно потоки. Причем любой процесс имеет хотя бы один поток. Этот поток называется главным (основным) потоком приложения.

Так как практически всегда потоков гораздо больше, чем физических процессоров для их выполнения, то потоки на самом деле выполняются не одновременно, а по очереди. (Заметьте, что распределение процессорного времени происходит именно между потоками, так как, еще раз повторю, процессы не выполняются.) Но переключение между ними происходит так часто, что кажется будто они выполняются параллельно.

В зависимости от ситуации потоки могут находиться в трех состояниях. Давайте посмотрим, что это за состояния. Во-первых, поток может выполняться, когда ему выделено процессорное время, т.е. он может находиться в состоянии активности. Во-вторых, он может быть неактивным и ожидать выделения процессора, т.е. быть в состоянии готовности. И есть еще третье, тоже очень важное состояние - состояние блокировки. Когда поток заблокирован, ему вообще не выделяется время. Обычно блокировка ставится на время ожидания какого-либо события. При возникновении этого события поток автоматически переводится из состояния блокировки в состояние готовности. Например, если один поток выполняет вычисления, а другой должен ждать результатов, чтобы сохранить их на диск. Второй мог бы использовать цикл типа "while(!isCalcFinished) continue;", но легко убедиться на практике, что во время выполнения этого цикла процессор занят на 100% (это называется активным ожиданием). Таких вот циклов следует по возможности избегать, в чем нам оказывает неоценимую помощь механизм блокировки. Второй поток может заблокировать себя до тех пор, пока первый не установит событие, сигнализирующее о том, что чтение окончено.

В системе выделяются два вида потоков - интерактивные, крутящие свой цикл обработки сообщений (такие, как главный поток приложения), и рабочие, представляющие собой простую функцию. Во втором случае поток завершается по мере завершения выполнения этой функции.

Заслуживающим внимания моментом является также способ организации очередности потоков. Можно было бы, конечно, обрабатывать все потоки по очереди, но такой способ далеко не самый эффективный. Гораздо разумнее оказалось ранжировать все потоки по приоритетам. Приоритет потока обозначается числом от 0 до 31, и определяется исходя из приоритета процесса, породившего поток, и относительного приоритета самого потока. Таким образом, достигается наибольшая гибкость, и каждый поток в идеале получает столько времени, сколько ему необходимо.

Иногда приоритет потока может изменяться динамически. Так интерактивные потоки, имеющие обычно класс приоритета Normal, система обрабатывает несколько иначе и несколько повышает фактический приоритет таких потоков, когда процесс, их породивший, находится на переднем плане (foreground). Это сделано для того, чтобы приложение, с которым в данный момент работает пользователь, быстрее реагировало на его действия.

Синхронизация потоков

Необходимость синхронизации

Итак, в Windows выполняются не процессы, а потоки. При создании процесса автоматически создается его основной поток. Этот поток в процессе выполнения может создавать новые потоки, которые, в свою очередь, тоже могут создавать потоки и т.д. Процессорное время распределяется именно между потоками, и получается, что каждый поток работает независимо.

Все потоки, принадлежащие одному процессу, разделяют некоторые общие ресурсы - такие, как адресное пространство оперативной памяти или открытые файлы. Эти ресурсы принадлежат всему процессу, а значит, и каждому его потоку. Следовательно, каждый поток может работать с этими ресурсами без каких-либо ограничений. Но так ли это в действительности? Вспомним, что в Windows реализована вытесняющая многозадачность - это значит, что в любой момент система может прервать выполнение одного потока и передать управление другому. (Раньше использовался способ организации, называемый кооперативной многозадачностью. Система ждала, пока поток сам не соизволит передать ей управление. Именно поэтому в случае глухого зависания одного приложения приходилось перезагружать компьютер. Так была организована, например, Windows 3.1). Что произойдет, если один поток еще не закончил работать с каким-либо общим ресурсом, а система переключилась на другой поток, использующий тот же ресурс? Произойдет штука очень неприятная, я вам это могу с уверенностью сказать, и результат работы этих потоков может чрезвычайно сильно отличаться от задуманного. Такие конфликты могут возникнуть и между потоками, принадлежащими различным процессам. Всегда, когда два или более потоков используют какой-либо общий ресурс, возникает эта проблема.

Именно поэтому необходим механизм, позволяющий потокам согласовывать свою работу с общими ресурсами. Этот механизм получил название механизма синхронизации потоков (thread synchronization).

Структура механизма синхронизации

Что же представляет собой этот механизм? Это набор объектов операционной системы, которые создаются и управляются программно, являются общими для всех потоков в системе (некоторые - для потоков, принадлежащих одному процессу) и используются для координирования доступа к ресурсам. В качестве ресурсов может выступать все, что может быть общим для двух и более потоков - файл на диске, порт, запись в базе данных, объект GDI, и даже глобальная переменная программы (которая может быть доступна из потоков, принадлежащих одному процессу).

Объектов синхронизации существует несколько, самые важные из них - это взаимоисключение (mutex), критическая секция (critical section), событие (event) и семафор (semaphore). Каждый из этих объектов реализует свой способ синхронизации. Какой из них следует использовать в каждом конкретном случае вы поймете, подробно познакомившись с каждым из этих объектов. Также в качестве объектов синхронизации могут использоваться сами процессы и потоки (когда один поток ждет завершения другого потока или процесса); а также файлы, коммуникационные устройства, консольный ввод и уведомления об изменении (к сожалению, освещение этих объектов синхронизации выходит за рамки данной статьи).

В чем смысл объектов синхронизации? Каждый из них может находиться в так называемом сигнальном состоянии. Для каждого типа объектов это состояние имеет различный смысл. Потоки могут проверять текущее состояние объекта и/или ждать изменения этого состояния и таким образом согласовывать свои действия. Что еще очень важно - гарантируется, что когда поток работает с объектами синхронизации (создает их, изменяет состояние) система не прервет его выполнения, пока он не завершит это действие. Таким образом, все конечные операции с объектами синхронизации являются атомарными (неделимыми), как бы выполняющимися за один такт.

ПРИМЕЧАНИЕ

Важно понимать, что никакой реальной связи между объектами синхронизации и ресурсами нет. Они не смогут предотвратить нежелательный доступ к ресурсу, они лишь подсказывают потокам, когда можно работать с ресурсом, а когда нужно подождать. Можно провести грубую аналогию со светофорами - они показывают, когда можно ехать, но ведь в принципе водитель может и не обратить внимания на красный свет (правда, потом он об этом скорее всего пожалеет ;)

Работа с объектами синхронизации

Чтобы создать тот или иной объект синхронизации, производится вызов специальной функции WinAPI типа Create... (напр. CreateMutex). Этот вызов возвращает дескриптор объекта (HANDLE), который может использоваться всеми потоками, принадлежащими данному процессу. Есть возможность получить доступ к объекту синхронизации из другого процесса - либо унаследовав дескриптор этого объекта, либо, что предпочтительнее, воспользовавшись вызовом функции открытия объекта (Open...). После этого вызова процесс получит дескриптор, который в дальнейшем можно использовать для работы с объектом. Объекту, если только он не предназначен для использования внутри одного процесса, обязательно присваивается имя. Имена всех объектов должны быть различны (даже если они разного типа). Нельзя, например, создать событие и семафор с одним и тем же именем.

По имеющемуся дескриптору объекта можно определить его текущее состояние. Это делается с помощью т.н. ожидающих функций. Чаще всего используется функция WaitForSingleObject. Эта функция принимает два параметра, первый из которых - дескриптор объекта, второй - время ожидания в мсек. Функция возвращает WAIT_OBJECT_0, если объект находится в сигнальном состоянии, WAIT_TIMEOUT - если истекло время ожидания, и WAIT_ABANDONED, если объект-взаимоисключение не был освобожден до того, как владеющий им поток завершился. Если время ожидания указано равным нулю, функция возвращает результат немедленно, в противном случае она ждет в течение указанного промежутка времени. В случае, если состояние объекта станет сигнальным до истечения этого времени, функция вернет WAIT_OBJECT_0, в противном случае функция вернет WAIT_TIMEOUT.

Если в качестве времени указана символическая константа INFINITE, то функция будет ждать неограниченно долго, пока состояние объекта не станет сигнальным.

Если необходимо узнавать о состоянии сразу нескольких объектов, следует воспользоваться функцией WaitForMultipleObjects.

Чтобы закончить работу с объектом и освободить дескриптор вызывается функция CloseHandle.

Очень важен тот факт, что обращение к ожидающей функции блокирует текущий поток, т.е. пока поток находится в состоянии ожидания, ему не выделяется процессорного времени.

Виды объектов синхронизации

Теперь давайте рассмотрим каждый тип объектов синхронизации в отдельности.

Взаимоисключения

Объекты-взаимоисключения (мьютексы, mutex - от MUTual EXclusion) позволяют координировать взаимное исключение доступа к разделяемому ресурсу. Сигнальное состояние объекта (т.е. состояние "установлен") соответствует моменту времени, когда объект не принадлежит ни одному потоку и его можно "захватить". И наоборот, состояние "сброшен" (не сигнальное) соответствует моменту, когда какой-либо поток уже владеет этим объектом. Доступ к объекту разрешается, когда поток, владеющий объектом, освободит его.

Для того, чтобы объявить взаимоисключение принадлежащим текущему потоку, надо вызвать одну из ожидающих функций. Поток, которому принадлежит объект, может его "захватывать" повторно сколько угодно раз (это не приведет к самоблокировке), но столько же раз он должен будет его освобождать с помощью функции ReleaseMutex.

События

Объекты-события используются для уведомления ожидающих потоков о наступлении какого-либо события. Различают два вида событий - с ручным и автоматическим сбросом. Ручной сброс осуществляется функцией ResetEvent. События с ручным сбросом используются для уведомления сразу нескольких потоков. При использовании события с автосбросом уведомление получит и продолжит свое выполнение только один ожидающий поток, остальные будут ожидать дальше.

Функция CreateEvent создает объект-событие, SetEvent - устанавливает событие в сигнальное состояние, ResetEvent-сбрасывает событие. Функция PulseEvent устанавливает событие, а после возобновления ожидающих это событие потоков (всех при ручном сбросе и только одного при автоматическом), сбрасывает его. Если ожидающих потоков нет, PulseEvent просто сбрасывает событие.

Семафоры

Объект-семафор - это фактически объект-взаимоисключение со счетчиком. Данный объект позволяет "захватить" себя определенному количеству потоков. После этого "захват" будет невозможен, пока один из ранее "захвативших" семафор потоков не освободит его. Семафоры применяются для ограничения количества потоков, одновременно работающих с ресурсом. Объекту при инициализации передается максимальное число потоков, после каждого "захвата" счетчик семафора уменьшается. Сигнальному состоянию соответствует значение счетчика больше нуля. Когда счетчик равен нулю, семафор считается не установленным (сброшенным).

Критические секции

Объект-критическая секция помогает программисту выделить участок кода, где поток получает доступ к разделяемому ресурсу, и предотвратить одновременное использование ресурса. Перед использованием ресурса поток входит в критическую секцию (вызывает функцию EnterCriticalSection). Если после этого какой-либо другой поток попытается войти в ту же самую критическую секцию, его выполнение приостановится, пока первый поток не покинет секцию с помощью вызова LeaveCriticalSection. Похоже на взаимоисключение, но используется только для потоков одного процесса.

Существует также функция TryEnterCriticalSection, которая проверяет, занята ли критическая секция в данный момент. С ее помощью поток в процессе ожидания доступа к ресурсу может не блокироваться, а выполнять какие-то полезные действия.

Защищенный доступ к переменным

Существует ряд функций, позволяющих работать с глобальными переменными из всех потоков не заботясь о синхронизации, т.к. эти функции сами за ней следят. Это функции InterlockedIncrement/InterlockedDecrement, InterlockedExchange, InterlockedExchangeAdd и InterlockedCompareExchange. Например, функция InterlockedIncrement увеличивает значение 32-битной переменной на единицу - удобно использовать для различных счетчиков. Более подробно об этих функциях см. в документации.

Пример

Как известно, теория лучше всего познается на примерах. Давайте рассмотрим небольшой пример работы с объектом-взаимоисключением. Для простоты я использовал консольное Win32 приложение, но как вы понимаете, это совершенно не обязательно.

```
#include <windows.h>
#include <iostream.h>

void main()
{
    DWORD res;

    // создаем объект-взаимоисключение
    HANDLE mutex = CreateMutex(NULL, FALSE, "APPNAME-MTX01");
    // если он уже существует, CreateMutex вернет дескриптор существующего
    // объекта, а GetLastError вернет ERROR_ALREADY_EXISTS

    // в течение 20 секунд пытаемся захватить объект
    cout<<"Trying to get mutex...\n"; cout.flush();
    res = WaitForSingleObject(mutex,20000);

    if (res == WAIT_OBJECT_0) // если захват удался
    {
        // ждем 10 секунд
        cout<<"Got it! Waiting for 10 secs...\n"; cout.flush();
        Sleep(10000);

        // освобождаем объект
        cout<<"Now releasing the object.\n"; cout.flush();
        ReleaseMutex(mutex);
    }

    // закрываем дескриптор
```

```
    CloseHandle(mutex);
}
```

Для проверки работы мьютекса запустите сразу два экземпляра этого приложения. Первый экземпляр сразу захватит объект и освободит его только через 10 секунд. Только после этого второму экземпляру удастся захватить объект. В данном примере объект используется для синхронизации между процессами, поэтому он обятельно должен иметь имя.

Синхронизация в MFC

Библиотека MFC содержит специальные классы для синхронизации потоков (CMutex, CEvent, CCriticalSection и CSemaphore). Эти классы соответствуют объектам синхронизации WinAPI и являются производными от класса CSyncObject. Чтобы понять, как их использовать, достаточно просто взглянуть на конструкторы и методы этих классов - Lock и Unlock. Фактически эти классы - всего лишь обертки для объектов синхронизации.

Есть еще один способ использования этих классов - написание так называемых потоково-безопасных классов (thread-safe classes). Потоково-безопасный класс - это класс, представляющий какой либо ресурс в вашей программе. Вся работа с ресурсом осуществляется только через этот класс, который содержит все необходимые для этого методы. Причем класс спроектирован таким образом, что его методы сами заботятся о синхронизации, так что в приложении он используется как обычный класс. Объект синхронизации MFC добавляется в этот класс в качестве закрытого члена класса, и все функции этого класса, осуществляющие доступ к ресурсу, согласуют с ним свою работу.

С классами синхронизации MFC можно работать как напрямую, используя методы Lock и Unlock, так и через промежуточные классы CSingleLock и CMultiLock (хотя на мой взгляд, работать через промежуточные классы несколько неудобно. Но использование класса CMultiLock необходимо, если вы хотите следить за состоянием сразу нескольких объектов).

Заключение

Игнорируя возможности многозадачности, которые предоставляет Windows, вы игнорируете преимущества этой операционной системы. Это как раз то, чего не может себе позволить ни один уважающий себя программист. А как вы могли убедиться, многозадачность - это вовсе не так сложно, как кажется на первый взгляд.

Интересующимся данной темой и владеющим английским могу порекомендовать следующие статьи и разделы MSDN:

1. Platform SDK / Windows Base Services / Executables / Processes and Threads
2. Platform SDK / Windows Base Services / Interprocess Communication / Synchronization

Автор: Алекс Jenter