Синхронизация процессов и потоков. Критическая секция20

Процессом (process) называется экземпляр программы, загруженной в память. Этот экземпляр может создавать нити (thread), которые представляют собой последовательность инструкций на выполнение. Важно понимать, что выполняются не процессы, а именно нити.

Причем любой процесс имеет хотя бы одну нить. Эта нить называется главной (основной) нитью приложения.

Так как практически всегда нитей гораздо больше, чем физических процессоров для их выполнения, то нити на самом деле выполняются не одновременно, а по очереди (распределение процессорного времени происходит именно между нитями). Но переключение между ними происходит так часто, что кажется, будто они выполняются параллельно.

В зависимости от ситуации нити могут находиться в трех состояниях. Во-первых, нить может выполняться, когда ей выделено процессорное время, т.е. она может находиться в состоянии активности. Во-вторых, она может быть неактивной и ожидать выделения процессора, т.е. быть в состоянии готовности. И есть еще третье, тоже очень важное состояние - состояние блокировки. Когда нить заблокирована, ей вообще не выделяется время. Обычно блокировка ставится на время ожидания какого-либо события. При возникновении этого события нить автоматически переводится из состояния блокировки в состояние готовности. Например, если одна нить выполняет вычисления, а другая должна ждать результатов, чтобы сохранить их на диск. Вторая могла 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 оказывает неоценимую помощь механизм блокировки. Вторая нить может заблокировать себя до тех пор, пока первая не установит событие, сигнализирующее о том, что чтение окончено.

Синхронизация нитей в ОС Windows

В Windows реализована вытесняющая многозадачность - это значит, что в любой момент система может прервать выполнение одной нити и передать управление другой. Ранее, в Windows 3.1, использовался способ организации, называемый кооперативной многозадачностью: система ждала, пока нить сама не передаст ей управление и именно поэтому в случае зависания одного приложения приходилось перезагружать компьютер.

Все нити, принадлежащие одному процессу, разделяют некоторые общие ресурсы - такие, как адресное пространство оперативной памяти или открытые файлы. Эти ресурсы принадлежат всему процессу, а значит, и каждой его нити. Следовательно, каждая нить может работать с этими ресурсами без каких-либо ограничений. Но... Если одна нить еще не закончила работать с каким-либо общим ресурсом, а система переключилась на другую нить, использующую этот же ресурс, то результат работы этих нитей может чрезвычайно сильно отличаться от задуманного. Такие конфликты могут возникнуть и между нитями, принадлежащими различным процессам. Всегда, когда две или более нитей используют какой-либо общий ресурс, возникает эта проблема.

Пример. Несинхронизированная работа нитей: если временно приостановить выполнение нити вывода на экран (пауза), фоновая нить заполнения массива будет продолжать работать.

```
#include <windows.h>
#include <stdio.h>
int a[5];
HANDLE hThr;
unsigned long uThrID;
void Thread( void* pParams )
{
        int i, num = 0;
        while (1)
        {
                for (i=0; i<5; i++) a[i] = num;
                num++;
        }
}

int main( void )
{
        hThr=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread,NULL,0,&uThrID
);
        while(1)
        printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
        return 0;
}
```

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

Этот механизм представляет собой набор объектов операционной системы, которые создаются и управляются программно, являются общими для всех нитей в системе (некоторые - для нитей, принадлежащих одному процессу) и используются для координирования доступа к ресурсам. В качестве ресурсов может выступать все, что может быть общим для двух и более нитей - файл на диске, порт, запись в базе данных, объект GDI, и даже глобальная переменная программы (которая может быть доступна из нитей, 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. Также в качестве объектов синхронизации могут использоваться сами процессы и нити (когда одна нить ждет завершения другой нити или процесса); а также файлы, коммуникационные устройства, консольный ввод и уведомления об изменении.

Любой объект синхронизации может находиться в так называемом сигнальном состоянии. Для каждого типа объектов это состояние имеет различный смысл. Нити могут проверять текущее состояние объекта и/или ждать изменения этого состояния и таким образом согласовывать свои действия. При этом гарантируется, что когда нить работает с объектами синхронизации (создает их, изменяет состояние) система не прервет ее выполнения, пока она не завершит это действие. Таким образом, все конечные операции с объектами синхронизации являются атомарными (неделимыми.

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 нитями, 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 владеющая им нить завершилась. 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.

Очень важен тот факт, что обращение к ожидающей функции блокирует текущую нить, т.е. пока нить находится в состоянии ожидания, ей не выделяется процессорного времени.

Критические секции

Объект-критическая секция помогает программисту выделить участок кода, где нить получает доступ к разделяемому ресурсу, и предотвратить одновременное использование ресурса. Перед использованием ресурса нить входит в критическую секцию (вызывает функцию EnterCriticalSection). Если после этого какая-либо другая нить попытается войти в ту же самую критическую секцию, ее выполнение приостановится, пока первая нить не покинет секцию с помощью вызова LeaveCriticalSection. Используется только для нитей одного процесса. Порядок входа в критическую секцию не определен.

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

Пример. Синхронизация нитей с помощью критических секций.

```
#include <windows.h>
#include <stdio.h>
CRITICAL_SECTION cs;
int a[5];
HANDLE hThr;
unsigned long uThrID;

void Thread( void* pParams )
{
        int i, num = 0;
        while (1)
        {
```

```
                EnterCriticalSection( &cs );
                for (i=0; i<5; i++) a[i] = num;
                num++;
                LeaveCriticalSection( &cs );
        }
}

int main( void )
{
        InitializeCriticalSection( &cs );
        hThr=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread,NULL,0,&uThrID
);
        while(1)
        {
                EnterCriticalSection( &cs );
                printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
                LeaveCriticalSection( &cs );
        }
        return 0;
}
```