Лекция 8.Потоки и файлы

Поток выполнения — наименьшая единица обработки, исполнение которой может быть назначено операционной системой. Реализация потоков выполнения и процессов в разных операционных системах отличается друг от друга, но в большинстве случаев поток выполнения находится внутри процесса. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой момент времени).

Обычно число потоков в системе гораздо больше числа процессоров. Поэтому операционной системе приходится эмулировать многопоточность, заставляя процессор поочередно переключаться между потоками. Время работы процессора разбивается на небольшие интервалы, обычно называемые квантами. Когда квант времени заканчивается, выполнение текущего потока приостанавливается, содержимое регистров процессора сохраняется в специальной области памяти, и он переключается на обработку следующего потока. Когда очередь снова дойдет до этого потока, содержимое регистров будет полностью восстановлено и работа потока продолжится так, как будто она вовсе и не прерывалась. Таким образом, переключение потоков происходит совершенно незаметно для них самих.

Многопоточность

Многопоточность, как широко распространённая модель программирования и исполнения кода, позволяет нескольким потокам выполняться в рамках одного процесса. Эти потоки выполнения совместно используют ресурсы процесса, но могут работать и самостоятельно. Многопоточная модель программирования предоставляет разработчикам удобную абстракцию параллельного выполнения.

Кдостоинствам многопоточности в программировании можно отнести следующее:

•Упрощение программы в некоторых случаях, за счет использования общего адресного пространства.

•Меньшие относительно процесса временны¬е затраты на создание потока.

•Повышение производительности процесса за счет распараллеливания процессорных вычислений и операций ввода/вывода.

Несинхронизированные потоки

Первый пример иллюстрирует работу с несинхронизированными потоками. Основной цикл, который является основным потоком процесса, выводит на экран содержимое глобального массива целых чисел. Поток, названный "Thread", непрерывно
заполняет глобальный массив целых чисел.

```
#include<process.h> #include<stdio.h>inta[ 5 ];
void Thread( void* pParams ) { int i, num = 0;
while ( 1 )
{
for ( i = 0; i < 5; i++ ) a[ i ] = num; num++;
}
}
int main( void )
{
_beginthread( Thread, 0, NULL ); while( 1 )
printf("%d %d %d %d %d\n",

a[ 0 ], a[ 1 ], a[ 2 ], a[ 3 ], a[ 4 ] ); return 0;
}
```

Как видно из результата работы процесса, основной поток (сама программа) и поток Thread действительно работают параллельно (красным цветом обозначено состояние, когда основной поток выводит массив во время его заполнения потоком Thread):

81751652 81751652 81751651 81751651 81751651
81751652 81751652 81751651 81751651 81751651
83348630 83348630 83348630 83348629 83348629
83348630 83348630 83348630 83348629 83348629

83348630 83348630 83348630 83348629 83348629

Запустите программу, затем нажмите "Pause" для остановки вывода на дисплей (т. е. приостанавливаются операции ввода/вывода основного потока, но поток Thread продолжает свое выполнение в фоновом режиме) и любую другую клавишу для возобновления выполнения.

Критические секции

А что делать, если основной поток должен читать данные из массива после его обработки в параллельном процессе? Одно из решений этой проблемы - использование критических секций.

Критические секции обеспечивают синхронизацию подобно мьютексам (о мьютексах см. далее) за исключением того, что объекты, представляющие критические секции, доступны в пределах одного процесса. События, мьютексы и семафоры также можно использовать в "однопроцессном" приложении, однако критические секции обеспечивают более быстрый и более эффективный механизм взаимно-исключающейсинхронизации. Подобно мьютексам объект, представляющий критическую секцию, может использоваться только одним потоком в данный момент времени, что делает их крайне полезными при разграничении доступа к общим ресурсам. Трудно предположитьчто-нибудьо порядке, в котором потоки будут получать доступ к ресурсу, можно сказать лишь, что система будет
"справедлива" ковсемпотокам.

```
#include <windows.h> #include <process.h> #include <stdio.h>
CRITICAL_SECTION cs; int a[ 5 ];
void Thread( void* pParams )
{
int i, num = 0; while ( TRUE )
{
EnterCriticalSection(&cs );
for ( i = 0; i < 5; i++ ) a[ i ] = num; LeaveCriticalSection( &cs );
num++;
}
}
int main( void )
{
InitializeCriticalSection(&cs ); _beginthread( Thread, 0, NULL ); while(
TRUE )
{
EnterCriticalSection( &cs ); printf( "%d %d %d %d %d\n", a[ 0 ], a[ 1
], a[ 2 ],
a[ 3 ], a[ 4 ] ); LeaveCriticalSection( &cs );
}

return 0;
}
```

Мьютексы (взаимоисключения)

Мьютекс (взаимоисключение, mutex) - это объект синхронизации, который устанавливается в особое сигнальное состояние, когда не занят каким-либопотоком. Только один поток владеет этим объектом в любой момент времени, отсюда и название таких объектов — одновременный доступ к общему ресурсу исключается. Например, чтобы исключить запись двух потоков в общий участок памяти в одно и то же время, каждый поток ожидает, когда освободится мьютекс, становится его владельцем и только потом пишет чтолибо в этот участок памяти. После всех необходимых действий мьютекс освобождается, предоставляя другим потокам доступ к общему ресурсу. Два (или более) процесса могут создать мьютекс с одним и тем же именем, вызвав метод CreateMutex . Первый процесс действительно создает мьютекс, а следующие процессы получают хэндл уже существующего объекта. Это дает возможность нескольким процессам получить хэндл одного и того же мьютекса, освобождая программиста от необходимости заботиться о том, кто в действительности создает мьютекс. Если используется такой подход, желательно установить флаг bInitialOwner в FALSE, иначе возникнут определенные трудности при определении действительного создателя мьютекса.

Несколько процессов могут получить хэндл одного и того же мьютекса, что делает возможным взаимодействие между процессами. Вы можете использовать следующие механизмы такого подхода:

•Дочерний процесс, созданный при помощи функции CreateProcess может наследовать хэндл мьютекса в случае, если при его (мьютекса) создании функией CreateMutex был указан параметр lpMutexAttributes .

•Процесс может получить дубликат существующего мьютекса с помощью функции DuplicateHandle .

•Процесс может указать имя существующего мьютекса при вызове функций OpenMutex или CreateMutex .

Вообще говоря, если вы синхронизируете потоки одного процесса, более эффективным подходом является использование критических секций.

```
#include <windows.h> #include <process.h> #include <stdio.h> HANDLE
hMutex;
int a[ 5 ];
void Thread( void* pParams )
{
int i, num = 0; while ( TRUE )
{
WaitForSingleObject( hMutex, INFINITE ); for ( i = 0; i < 5; i++ ) a[ i
] = num; ReleaseMutex( hMutex );
num++;
}
}
int main( void )
{
hMutex = CreateMutex( NULL, FALSE, NULL ); _beginthread( Thread, 0,
NULL );
while( TRUE )
{
WaitForSingleObject( hMutex, INFINITE ); printf( "%d %d %d %d %d\n",
a[ 0 ], a[ 1 ], a[ 2 ], a[ 3 ], a[ 4 ] ); ReleaseMutex( hMutex );
}
```

Тип объекта	Описание
Событие с ручным сбросом	Это объект, сигнальное состояние которого сохраняется до ручного сброса функцией ResetEvent . Как только состояние объекта установлено в сигнальное, все находящиеся в цикле ожидания этого объекта потоки продолжают свое выполнение (освобождаются).
Событие с автоматическим сбросом	Объект, сигнальное состояние которого сохраняется до тех пор, пока не будет освобожден единственный поток, после чего система автоматически устанавливает несигнальное состояние события. Если нет потоков, ожидающих этого события, объект остается в сигнальном состоянии.

```
return 0;
}
```

События

А что, если мы хотим, чтобы в предыдущем примере второй поток запускался каждый раз после того, как основной поток закончит печать содержимого массива, т.е. значения двух последующих строк будут отличаться строго на 1?

Событие - это объект синхронизации, состояние которого может быть установлено в сигнальное путем вызова функций SetEvent или PulseEvent .
Существует два типа событий:

События полезны в тех случаях, когда необходимо послать сообщение потоку, сообщающее, что произошло определенное событие. Например, при асинхронных операциях ввода и вывода из одного устройства, система устанавливает событие в сигнальное состояние когда заканчивается какая-либоиз этих операций. Один поток может использовать несколько различных событий в нескольких перекрывающихся операциях, а затем ожидать прихода сигнала от любого из них.

Поток может использовать функцию CreateEvent для создания объекта события. Создающий событие поток устанавливает его начальное состояние. В создающем потоке можно указать имя события. Потоки других процессов могут получить доступ к этому событию по имени, указав его в функции OpenEvent .

Поток может использовать функцию PulseEvent для установки состояния события в сигнальное и затем сбросить состояние в несигнальное после освобождения соответствующего количества ожидающих потоков. В случае объектов с ручным сбросом освобождаются все ожидающие потоки. В случае объектов с автоматическим сбросом освобождается только единственный поток, даже если этого события ожидают несколько потоков. Если ожидающих потоков нет, PulseEvent просто устанавливает состояние события
в несигнальное.

```
#include <windows.h> #include <process.h> #include <stdio.h> HANDLE
hEvent1, hEvent2; int a[ 5 ];
void Thread( void* pParams )
{
int i, num = 0; while ( TRUE )
{
WaitForSingleObject( hEvent2, INFINITE ); for ( i = 0; i < 5; i++ ) a[
i ] = num; SetEvent( hEvent1 );
num++;
}

}
int main( void )
{
hEvent1 = CreateEvent( NULL, FALSE, TRUE, NULL ); hEvent2 = CreateEvent(
NULL, FALSE, FALSE, NULL ); _beginthread( Thread, 0, NULL );
while( TRUE )
{
WaitForSingleObject( hEvent1, INFINITE ); printf( "%d %d %d %d %d\n",
a[ 0 ], a[ 1 ], a[ 2 ], a[ 3 ], a[ 4 ] ); SetEvent( hEvent2 );
}
return 0;
}
```