А.В. Макаров, С.Ю. Скоробогатов, А.М. Чеповский

Учебный курс “CIL и системное программирование в Microsoft .NET”

Лекция 21.

Применение потоков и волокон

21. Применение потоков и волокон

- **Процесс (process)**

  Определяет адресное пространство, в котором выполняется код приложения, и стандартный контекст безопасности
- **Поток (thread, нить)**

  Объект планирования процессорного времени. Планировщик потоков принадлежит ядру операционной системы.
- **Волокно (fiber, нить)**

  Альтернативный механизм планирования процессорного времени. Планировщик принадлежит процессу и предоставляется разработчиком.
- **Обработка сообщений**

  Еще одна альтернатива планировщика уровня процесса, основанная на использовании цикла обработки сообщений. (GUI, Single Thread Apartment)
  (*здесь не рассматривается*)

21.1. Потоко-безопасные и небезопасные функции

char* strtok(char *string, char *delimiters);

```
unsigned __stdcall ThreadProc( void *param )
{
    char *token = strtok( "one,two,three", "," );
    while( token ) {
        …
        token = strtok( NULL, seps );
    }
}
```

Windows

Функция strtok запоминает во внутренней статической переменной место, где находился предыдущий разделитель.

char* strtok_r(char *string, char *delimiters, char **ptrptr);

```
unsigned __stdcall ThreadProc( void *param )
{
    char *temp;
    char *token = strtok_r( "one,two,three", ",", &temp );
    while( token ) {
        …
        token = strtok_r( NULL, ",", &temp );
    }
}
```

Linux

21.2. Создание потоков. Win32 API и библиотека времени исполнения

Функции Win32 API

```
HANDLE CreateThread(
   LPSECURITY_ATTRIBUTES lpAttr, SIZE_T dwStackSize,
   LPTHREAD_START_ROUTINE lpProcedure,
   LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId
);
void ExitThread( DWORD dwExitCode );
```

Функции библиотеки времени исполнения

```
uintptr_t _beginthread(
   void (__cdecl *start_address)(void *), unsigned stack_size,
   void *arglist
);
void _endthread( void );
uintptr_t _beginthreadex(
   void *security, unsigned stack_size,
   unsigned (__stdcall *start_address)(void *),
   void *arglist, unsigned initflag, unsigned *thrdaddr
);
void _endthreadex( unsigned retval );
```

21.2. Управление потоками. Win32 API

Функции Win32 API

```
DWORD SuspendThread( hThread );
DWORD ResumeThread( hThread );
BOOL  TerminateThread( HANDLE hThread, DWORD dwExitCode );
BOOL  GetExitCodeThread( HANDLE hThread, LPDWORD lpdwExitCode );
BOOL  SwitchToThread( void );
VOID  Sleep( DWORD dwMilliseconds );
```

Типы таймеров

Таймер низкого разрешения

```
DWORD GetTickCount( void );
```

Основан на планировщике потоков, разрешение порядка 10-15 мс.

«Мультимедийный» таймер

```
DWORD timeGetTime( void );
MMRESULT timeBeginPeriod( UINT uResilution );
```

Разрешение от 1 мс, изменение разрешения мультимедийного таймера влияет на работу планировщика ядра и на все процессы системы.

Таймер высокого разрешения

```
BOOL QueryPerformanceCounter( LARGE_INTEGER *lpCounter );
BOOL QueryPerformanceFrequency( LARGE_INTEGER *lpCounter );
```

Использует счетчик тактов процессора, разрешение лучше 1 мкс.

21.2. Потоки, пример реализации

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

unsigned __stdcall ThreadProc( void *param ) {
    Sleep( 1000 );
    delete[] (int*)param;
    return 0;
}

int main( void ) {
   int      i;
   HANDLE   hThread[ 10 ];
   DWORD    dwThreadId;

   for ( i = 0; i < 10; i++ )
      hThread[i] = (HANDLE)_beginthreadex(
         NULL, 0, ThreadProc, new int [128], 0, &dwThreadId
      );
   /* здесь код выполняется одновременно с потоками */

   WaitForMultipleObjects( 10, hThread, TRUE, INFINITE );
   for ( i = 0; i < 10; i++ ) CloseHandle( hThread[i] );

   return 0;
}
```

Процедура потока выполняется «параллельно» с остальными потоками и, в том числе, с первичным потоком процесса.

Дожидаемся завершения всех созданных потоков

21.3. Волокна

```
#define _WIN32_WINNT 0x0400
#include <windows.h>
```

Инициализация внутренних данных для работы с волокнами

```
LPVOID ConvertThreadToFiber( LPVOID lpParameter );
```

Создание нового волокна

```
LPVOID CreateFiber(
    SIZE_T dwStackSize, LPFIBER_START_ROUTINE lpStartAddress,
    LPVOID lpParameter );
LPVOID CreateFiberEx(
    SIZE_T dwStackCommitSize, SIZE_T dwStackReserveSize,
    DWORD dwFlags, LPFIBER_START_ROUTINE lpStartAddress,
    LPVOID lpParameter );
```

Переключение на волокно

```
VOID SwitchToFiber( LPVOID lpFiber );
```

Удаление волокна

```
VOID DeleteFiber( LPVOID lpFiber );
```

Освобождение внутренних структур данных, выделенных для волокон

```
BOOL ConvertFiberToThread(void);
```

21.3. Волокна, пример реализации

```
#define FIBERS    2
static LPVOID     fiberEnd = NULL, fiberCtl, fiber[ FIBERS ];
int main( void ) {
    int    i;
    fiberCtl = ConvertThreadToFiber( NULL );
    for ( i = 0; i < FIBERS; i++ )
        fiber[i] = CreateFiber( 10000, FiberProc, NULL );
    for ( i = 0; i < FIBERS; ) {
        SwitchToFiber( fiber[i] );
        if ( fiberEnd ) {
            DeleteFiber( fiberEnd );
            for ( i = 0; i < FIBERS; i++ )
                if ( fiber[i] == fiberEnd ) fiber[i] = NULL;
            fiberEnd = NULL;
        }
        for ( i = 0; i < FIBERS; i++ ) if ( fiber[i] ) break;
    }
    ConvertFiberToThread();
    return 0;
}
```

Начинаем работу с волокнами

Сюда мы попадаем при завершении волокна

Завершение работы с волокнами

21.3. Волокна, пример реализации

```
static void shedule( BOOL fDontEnd )
{
    int         n, current;

    if ( !fDontEnd ) {
        fiberEnd = GetCurrentFiber();
        SwitchToFiber( fiberCtl );
    }

    for ( n = 0; n < FIBERS; n++ )
        if ( fiber[n] && fiber[n] != GetCurrentFiber() ) break;

    if ( n >= FIBERS ) return;  /* других рабочих волокон нет */

    SwitchToFiber( fiber[n] );
}
```

Волокно заканчивает работу

переключение в main, где волокно будет уничтожено

переключение на следующее рабочее волокно

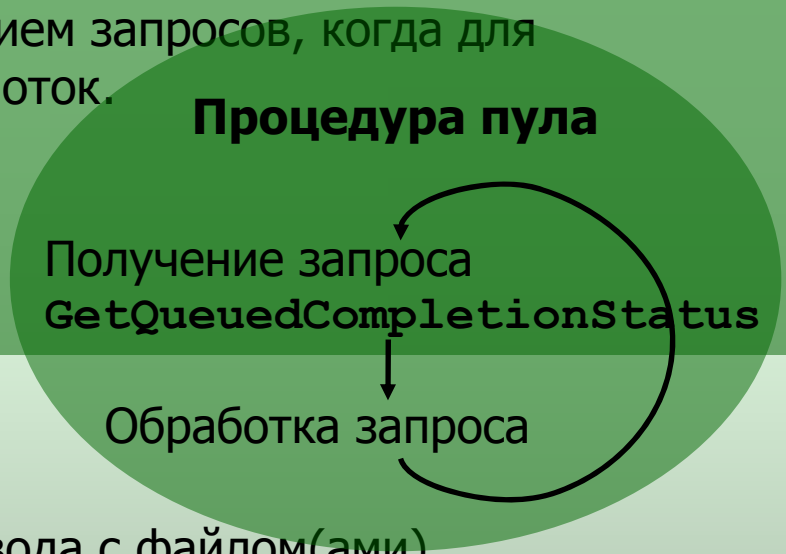
```
VOID CALLBACK FiberProc( PVOID lpParameter )
{
    int    i;

    for ( i=0; i<100; i++ ) { Sleep( 1000 ); shedule( TRUE ); }

    shedule( FALSE );
}
```

Собственно рабочая часть волокна

21.4. Пул потоков порт завершения ввода-вывода

Порт завершения ввода-вывода – эффективное средство управления многопоточными приложениями с обслуживанием запросов, когда для обработки каждого запроса выделяется свой поток.

- Малые затраты на управление пулом
- Высокая утилизация процессора
- Экономия системных ресурсов



1. Создание порта завершения ввода-вывода
`CreateIoCompletionPort`

2. Ассоциирование порта завершения ввода-вывода с файлом(ами)
`CreateIoCompletionPort` (необязательно)

3. Создание пула потоков, выполняющих процедуру пула
`_beginthreadex, _beginthread`

4. Распределение поступающих запросов по потокам
`PostQueuedCompletionStatus`

21.4. Пул потоков
порт завершения ввода-вывода

```
HANDLE CreateIoCompletionPort(
    HANDLE FileHandle, HANDLE ExistingCompletionPort,
    ULONG_PTR CompletionKey, DWORD NumberOfConcurrentThreads );
BOOL PostQueuedCompletionStatus(
    HANDLE CompletionPort, DWORD dwNumberOfBytesTransferred,
    ULONG_PTR CompletionKey, LPOVERLAPPED lpOverlapped );
BOOL GetQueuedCompletionStatus(
   HANDLE CompletionPort, LPDWORD lpNumberOfBytes,
   PULONG_PTR lpCompletionKey, LPOVERLAPPED* lpOverlapped,
   DWORD dwMilliseconds );
```

Создание порта завершения ввода-вывода

```
#define CONCURRENTS    4
HANDLE hIoCP;
hIoCP = CreateIoCompletionPort(
    INVALID_HANDLE_VALUE, NULL, NULL, CONCURRENTS
);
```

Максимальное число одновременно активных потоков

Ассоциирование порта завершения ввода-вывода с файлом(ами)

```
#define SOME_NUMBER    123
CreateIoCompletionPort( hFile, hIoCP, SOME_NUMBER, 0 );
```

Произвольный номер, передаваемый в процедуру пула

21.4. Пул потоков
порт завершения ввода-вывода, пример

```
#include <process.h>
#define _WIN32_WINNT 0x0500
#include <windows.h>
#define MAXQUERIES   15
#define CONCURENTS   3
#define POOLSIZE     5
unsigned __stdcall PoolProc( void *arg )
{
    DWORD          size;
    ULONG_PTR      key;
    LPOVERLAPPED   lpov;
    while (
        GetQueuedCompletionStatus(
            (HANDLE)arg, &size, &key, &lpov, INFINITE
        )
    ) {
      if ( !size && key == (ULONG_PTR)-1 ) break;
      Sleep( 300 );   /* обработка запроса */
   }
   return 0L;
}
```

Произвольный номер, переданный в процедуру пула

21.4. Пул потоков
порт завершения ввода-вывода, пример

```
int main( void )
{
    int      i;
    HANDLE   hCPort, hThread[ POOLSIZE ];
    DWORD    temp;

    hCPort = CreateIoCompletionPort(
        INVALID_HANDLE_VALUE, NULL, NULL, CONCURENTS
    );
    for ( i = 0; i < POOLSIZE; i++ )
        hThread[i] = (HANDLE)_beginthreadex(
            NULL, 0, PoolProc, (void*)hCPort, 0, (unsigned*)&temp
        );

    for ( i = 0; i < MAXQUERIES; i++ )
        PostQueuedCompletionStatus( hCPort, 1, i, NULL );

    /* завершаем работу всех потоков, разослав уведомления */
    for ( i = 0; i < POOLSIZE; i++ )
        PostQueuedCompletionStatus(hCPort,0,(ULONG_PTR)-1,NULL);

    WaitForMultipleObjects( POOLSIZE, hThread, TRUE, INFINITE );

    for ( i = 0; i < POOLSIZE; i++ ) CloseHandle( hThread[i] );
    CloseHandle( hCPort );

    return 0;
}
```

Создаем порт и пул потоков

Распределяем запросы по потокам пула

Ждем завершения потоков и закрываем описатели

21.4. Пул потоков, управляемый системой

При этом система берет на себя:

- Создание порта завершения ввода-вывода (ассоциирование с файлами не предусмотрено).
- Создание пула потоков (число потоков зависит от числа процессоров)
- Реализацию процедуры пула (разработчик предоставляет процедуру только для обработки запроса, которая будет вызвана стандартной процедурой пула).
- Создание дополнительных потоков пула при необходимости.

Для работы со стандартным пулом предназначена процедура:

```
BOOL QueueUserWorkItem(
     LPTHREAD_START_ROUTINE QueryFunction, PVOID pContext,
     ULONG Flags
);
```

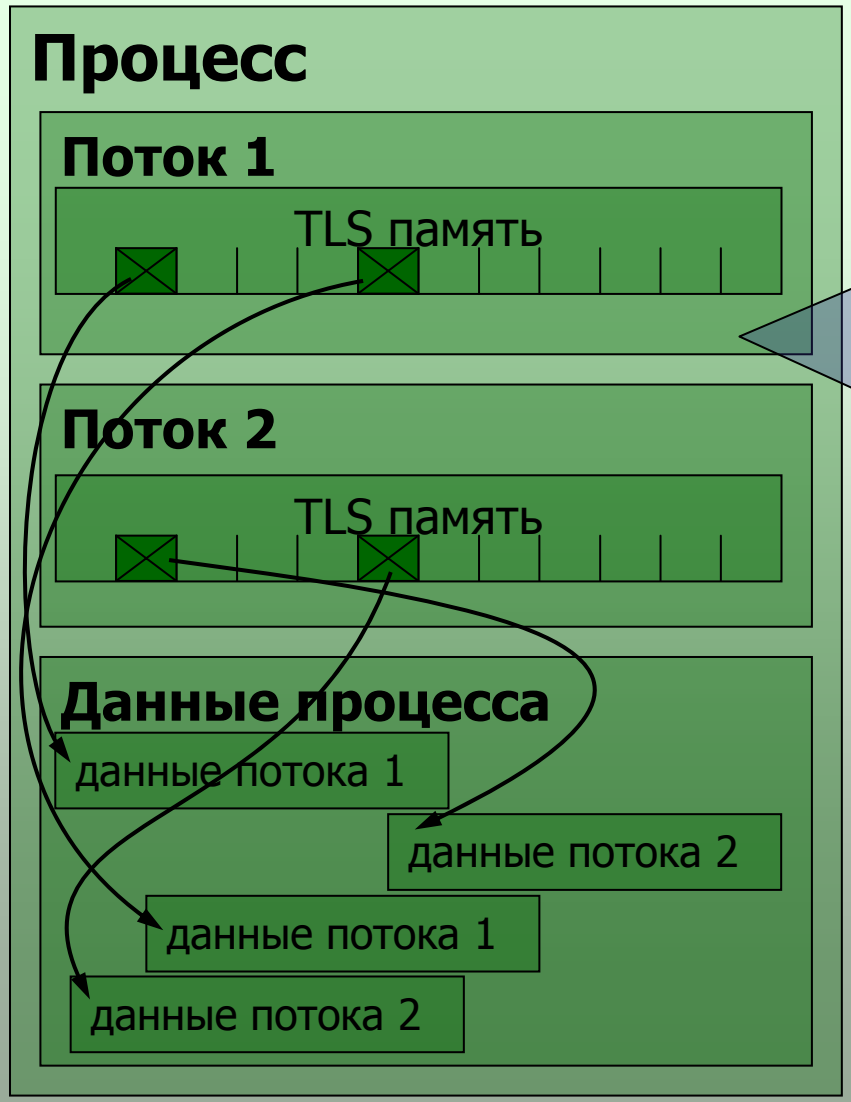
которая при первом обращении создает порт и пул и ставит в очередь порта запрос, а в последующем просто добавляет запросы в очередь. Последний параметр «**Flags**» позволяет влиять на способ назначения процедуре потока, например **WT_EXECUTELONGFUNCTION** заставит систему создать новый поток.

21.4. Пул потоков, управляемый системой пример реализации

```
#include <process.h>
#define _WIN32_WINNT 0x0500
#include <windows.h>

#define MAXQUERIES  15
#define POOLSIZE    3

static LONG    cnt;
static HANDLE  hEv;

DWORD WINAPI QProc( LPVOID lpData );

int main( void )
{
   int    i;

   hEv = CreateEvent( NULL, TRUE, FALSE, 0 );

   for ( i = 0; i < POOLSIZE; i++ )
      QueueUserWorkItem( QProc, NULL, WT_EXECUTELONGFUNCTION );

   for ( ; i < MAXQUERIES; i++ )
      QueueUserWorkItem( QProc, NULL, WT_EXECUTEDEFAULT );

   WaitForSingleObject( hEv, INFINITE );
   CloseHandle( hEv );

   return 0;
}
```

```
DWORD WINAPI QProc( LPVOID lpData )
{
   int r = InterlockedIncrement( &cnt );

   Sleep( 300 );
   if ( r >= MAXQUERIES ) SetEvent(hEv);

   return 0L;
}
```

21.5. Память, локальная для потоков и волокон



Каждый поток обладает вектором TLS (Thread Local Storage) памяти. При обращении к элементам этого вектора каждый поток получает доступ к своей копии вектора и не имеет возможности прочитать данные из вектора другого потока.

Обычно в ячейках TLS вектора хранятся указатели на данные, размещаемые в общем адресном пространстве. При этом любой поток потенциально может получить доступ к данным другого потока, но только владелец точно знает, где они находятся.

Аналогичный механизм FLS (Fiber Local Storage) предусмотрен для волокон

21.5. Память, локальная для потоков и волокон функции Win32 API

```
DWORD TlsAlloc( void );
```

Зарезервировать ячейку в TLS векторе.
Возвращается индекс ячейки, по которому любой поток получит доступ к ячейке в собственном векторе.

```
void TlsFree( DWORD dwIndex );
```

Освободить ячейку в TLS векторе.
Все потоки **теряют(!)** данные в своих ячейках.

```
LPVOID TlsGetValue( DWORD dwIndex );
BOOL TlsSetValue( DWORD dwIndex, LPVOID lpValue );
```

Прочитать или изменить значение в собственной ячейке TLS вектора.

```
DWORD FlsAlloc( PFLS_CALLBACK_FUNCTION lpCallback );
VOID WINAPI FlsCallback( PVOID lpFlsData )
{
    ...
}
```

Предоставляется разработчиком для очистки ресурсов, на которые ссылается освобождаемая ячейка в FLS памяти потока

```
void FlsFree( DWORD dwIndex );

PVOID FlsGetValue( DWORD dwIndex );
BOOL FlsSetValue( DWORD dwIndex, PVOID lpValue );
```

21.5. Память, локальная для потоков
Пример реализации

```
#include <process.h>
#include <windows.h>
static DWORD dwTlsData;}
unsigned __stdcall ThreadProc( void *param ) {
   TlsSetValue( dwTlsData, (LPVOID)new int array[100] );
   Proc( (int)param );
   delete (int[])TlsGetValue( dwTlsData );
   return 0;
}
int main( void ) {
   HANDLE   hThread;   unsigned dwThread;
   dwTlsData = TlsAlloc();
   hThread = (HANDLE) _beginthreadex(
      NULL, 0, ThreadProc, (void*)i, 0, &dwThread
   );
   WaitForSingleObject( hThread, INFINITE );
   CloseHandle( hThread );
   TlsFree( dwTlsData );
   return 0;
}
```

```
void Proc( int x ) {
   int   *iptr;
   iptr=(int*)TlsGetValue(dwTlsData);
   iptr[99] = x;
```

Функции, исполняемые в контексте разных потоков будут использовать собственные данные.

Индекс сохраняется в глобальной переменной и все потоки обращаются к ячейкам с одинаковым индексом в своих TLS векторах

21.5. Память, локальная для потоков декларативный подход

Рассмотренный императивный подход с явным использованием функций управления TLS памятью имеет достаточно высокую трудоемкость и некоторые ограничения, связанные с размерами TLS вектора.

Альтернативно возможен декларативный подход, когда работу по управлению TLS памятью берет на себя компилятор совместно с библиотекой времени исполнения.

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

__declspec(thread) int iTls[ 10000 ];

unsigned __stdcall ThreadProc( void *param )
{
   int   i;

   for ( i = 0; i < 10000; i++ ) iTls[i] = i*i;

   …

   return 0;
}
```

Компилятор разместит переменную в специальном сегменте (`_TLS`), который средствами run-time библиотеки создается для каждого потока

Декларативный подход для описания FLS пока не предусмотрен.

21.6. Поддержка NUMA

NUMA системы отличаются от обычных SMP тем, что доступ разных процессоров к разным адресам общего адресного пространства занимает различное время и сопровождается различными накладными расходами.

В Windows введены понятия **идеального процессора** (т.е. процессора, на котором первый раз запущено приложение) и **привязки процесса или потока** (affinity mask) к процессору, которые активно используются планировщиком для эффективного использования аппаратуры NUMA системы.

Привязка процесса или потока к процессору задается целым числом, содержащим битовую маску. Самый младший бит соответствует первому процессору.

Поддержка NUMA сводится к возможности назначить маску привязки к процессору, изменить номер идеального процессора и узнать топологию NUMA системы (объем памяти и число процессоров на каждом узле) и определить принадлежность конкретного процессора узлу.

21.6. Поддержка NUMA, функции Win32 API

```
BOOL GetProcessAffinityMask(
  HANDLE hProcess, PDWORD_PTR lpProcessAM, PDWORD_PTR lpSystemAM
);

BOOL SetProcessAffinityMask(
  HANDLE hProcess, DWORD_PTR dwProcessAffinityMask
);

DWORD_PTR SetThreadAffinityMask(
  HANDLE hThread, DWORD_PTR dwThreadAffinityMask
);

DWORD SetThreadIdealProcessor(
  HANDLE hThread, DWORD dwIdealProcessor
);

BOOL GetNumaHighestNodeNumber( PULONG HighestNodeNumber );

BOOL GetNumaProcessorNode( UCHAR Processor, PUCHAR NodeNumber );

BOOL GetNumaNodeProcessorMask(
  UCHAR Node, PULONGLONG ProcessorMask
);

BOOL GetNumaAvailableMemoryNode(
  UCHAR Node, PULONGLONG AvailMem
);
```