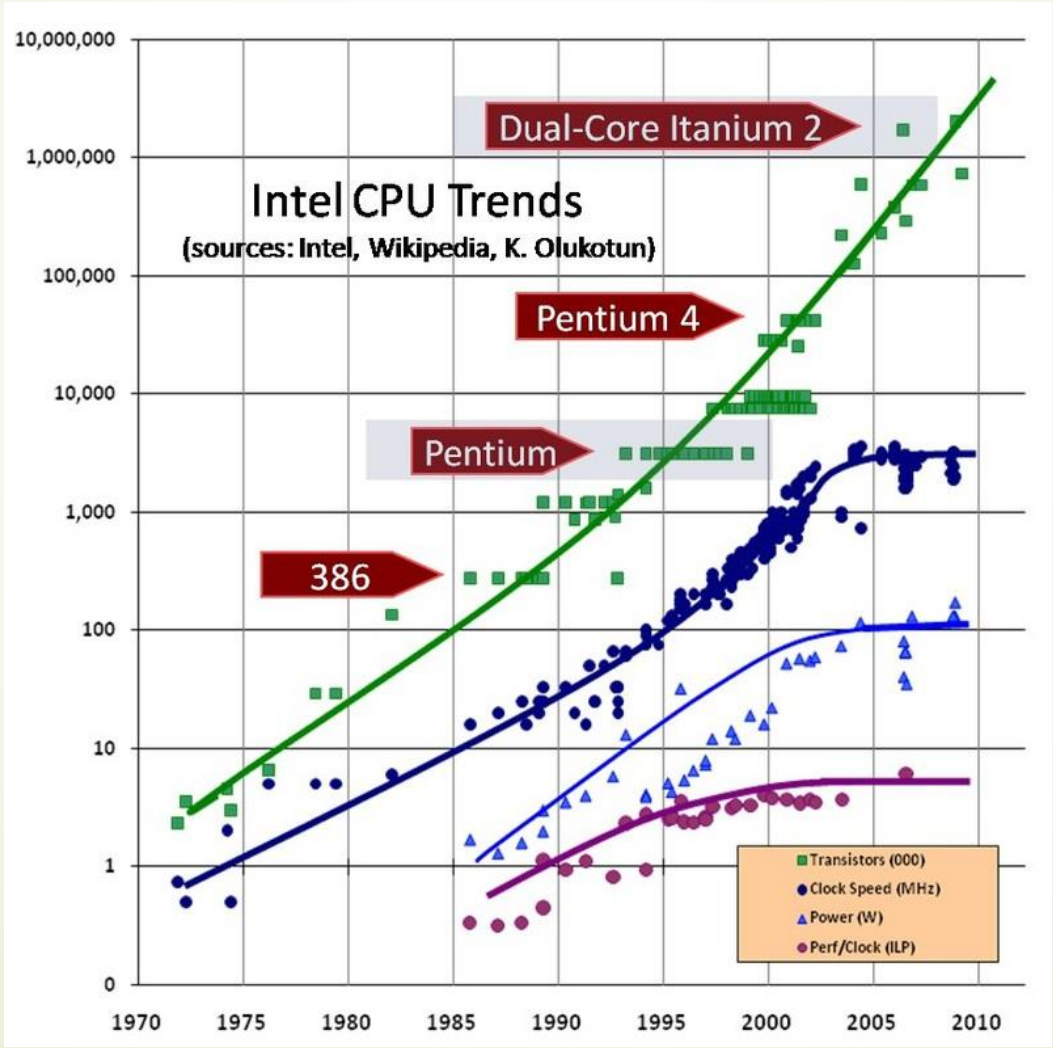
Лекция 7
Многопоточность

Процессы и потоки

- Процесс – это исполняемая копия приложения. Каждый процесс предоставляет ресурсы, необходимые для выполнения программы: процесс имеет виртуальное адресное пространство, открытые дескрипторы системных объектов, уникальный идентификатор процесса, класс приоритета, потоки выполнения. Каждый процесс запускается с одного потока, часто называемого основным потоком, но может создавать дополнительные потоки из любого из своих потоков
- Поток – последовательность инструкций, выполняемых устройством (CPU). Потоки существуют в рамках каких-либо процессов. У каждого потока есть собственные регистры и собственный стек, но они используют разделяемые ресурсы, т.е. другие потоки в данном процессе могут их использовать

Управление потоками на основе квантования процессорного времени

Рост технических характеристик процессоров семейства Intel



Преимущества многопоточности

- Асинхронное выполнение задач (например, работа с UI)
- Увеличение производительности в случае многоядерной системы
- Улучшенная структура программы

Работа с потоками в C# (CLR)

- Основной функционал для использования потоков в приложении сосредоточен в пространстве имен **System.Threading**. В нем определен класс, представляющий отдельный поток - класс **Thread**.
- Класс Thread определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. Основные свойства класса:

- Статическое свойство **CurrentContext** позволяет получить контекст, в котором выполняется поток
- Статическое свойство **CurrentThread** возвращает ссылку на выполняемый поток
- Свойство **IsAlive** указывает, работает ли поток в текущий момент
- Свойство **IsBackground** указывает, является ли поток фоновым
- Свойство **Name** содержит имя потока
- Свойство **Priority** хранит приоритет потока - значение перечисления ThreadPriority
- Свойство **ThreadState** возвращает состояние потока - одно из значений перечисления ThreadState

- Некоторые методы класса Thread:

- Статический метод **GetDomain** возвращает ссылку на домен приложения
- Статический метод **GetDomainId** возвращает id домена приложения, в котором выполняется текущий поток
- Статический метод **Sleep** останавливает поток на определенное количество миллисекунд
- Метод **Abort** уведомляет среду CLR о том, что надо прекратить поток, однако прекращение работы потока происходит не сразу, а только тогда, когда это становится возможно. Для проверки завершенности потока следует опрашивать его свойство ThreadState
- Метод **Interrupt** прерывает поток на некоторое время
- Метод **Join** блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
- Метод **Resume** возобновляет работу ранее приостановленного потока
- Метод **Start** запускает поток
- Метод **Suspend** приостанавливает поток

Статусы потоков

- Статусы потока содержатся в перечислении ThreadState:
- Aborted: поток остановлен, но пока еще окончательно не завершен
- AbortRequested: для потока вызван метод Abort, но остановка потока еще не произошла
- Background: поток выполняется в фоновом режиме
- Running: поток запущен и работает (не приостановлен)
- Stopped: поток завершен
- StopRequested: поток получил запрос на остановку
- Suspended: поток приостановлен
- SuspendRequested: поток получил запрос на приостановку
- Unstarted: поток еще не был запущен
- WaitSleepJoin: поток заблокирован в результате действия методов Sleep или Join

- В процессе работы потока его статус многократно может измениться под действием методов. Так, в самом начале еще до применения метода Start его статус имеет значение Unstarted. Запустив поток, мы изменим его статус на Running. Вызвав метод Sleep, статус изменится на WaitSleepJoin. А применяя метод Abort, мы тем самым переведем поток в состояние AbortRequested, а затем Aborted, после чего поток окончательно завершится.

Приоритеты потоков

- Приоритеты потоков располагаются в перечислении ThreadPriority:
- Lowest
- BelowNormal
- Normal
- AboveNormal
- Highest
- По умолчанию потоку задается значение Normal. Однако мы можем изменить приоритет в процессе работы программы. Например, повысить важность потока, установив приоритет Highest. Среда CLR будет считывать и анализировать значения приоритета и на их основании выделять данному потоку то или иное количество времени.

Создание потоков

```
public static void Count(){
        for (int i = 1; i < 9; i++){
            Console.WriteLine("Второй поток:");
            Console.WriteLine(i * i);
            Thread.Sleep(400);
        }
}
...
Thread myThread = new Thread(new ThreadStart(Count));
myThread.Start();
for (int i = 1; i < 9; i++){
     Console.WriteLine("Главный поток:");
      Console.WriteLine(i * i);
     Thread.Sleep(300);
}
```

Создание параметризованных потоков

```
public static void Count(object x){
        for (int i = 1; i < 9; i++){
            int n = (int)x;
            Console.WriteLine("Второй поток:");
            Console.WriteLine(i * n);
            Thread.Sleep(400);
        }
}
…
int number = 4;
var myThread = new Thread(new ParameterizedThreadStart(Count));
myThread.Start(number);
for (int i = 1; i < 9; i++){
            Console.WriteLine("Главный поток:");
            Console.WriteLine(i * i);
            Thread.Sleep(300);
}
```

Типобезопасный подход

```
public class Counter{
    private int _x;
    private int _y;
    public Counter(int x, int y){
          this._x = x;
          this._y = y;
    }
    public void Count(){
          for (int i = 1; i < 9; i++){
                Console.WriteLine("Второй поток:");
                Console.WriteLine(i * x * y);
                Thread.Sleep(400);
            }
      }
}
…
var counter = new Counter(5, 4);
var myThread = new Thread(new ThreadStart(counter.Count));
myThread.Start();
```

Синхронизация потоков

```
public static void Count(){
    x = 1;
    for (int i = 1; i < 9; i++){
        Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
        x++;
        Thread.Sleep(100);
    }
}
…
static int x = 0;
for (int i = 0; i < 5; i++){
    var myThread = new Thread(Count);
    myThread.Name = "Поток " + i.ToString();
    myThread.Start();
}
```

Средства синхронизации

- Critical section
- Mutex
- Semaphore
- Event
- Monitor

Locker

```
public static void Count(){
      lock (locker){
            x = 1;
            for (int i = 1; i < 9; i++){
                  Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
                  x++;
                  Thread.Sleep(100);
            }
      }
}
...
int x = 0;
static object locker = new object();
for (int i = 0; i < 5; i++){
      var myThread = new Thread(Count);
      myThread.Name = "Поток " + i.ToString();
      myThread.Start();
}
```

Monitor

```
public static void Count(){
      try{
            Monitor.Enter(locker);
            x = 1;
            for (int i = 1; i < 9; i++){
                  Console.WriteLine("{0}: {1}",
                  Thread.CurrentThread.Name, x);
                  x++;
                  Thread.Sleep(100);
            }
      }
      finally{
            Monitor.Exit(locker);
      }
}
…
```

- Кроме блокировки и разблокировки объекта класс **Monitor** имеет еще ряд методов, которые позволяют управлять синхронизацией потоков. Так, метод **Monitor.Wait** освобождает блокировку объекта и переводит поток в очередь ожидания объекта. Следующий поток в очереди готовности объекта блокирует данный объект. А все потоки, которые вызвали метод **Wait**, остаются в очереди ожидания, пока не получат сигнала от метода **Monitor.Pulse** или **Monitor.PulseAll**, посланного владельцем блокировки. Если метод **Monitor.Pulse** отправлен, поток, находящийся во главе очереди ожидания, получает сигнал и блокирует освободившийся объект. Если же метод **Monitor.PulseAll** отправлен, то все потоки, находящиеся в очереди ожидания, получают сигнал и переходят в очередь готовности, где им снова разрешается получать блокировку объекта.

AutoResetEvent

```
public static void Count(){
	waitHandler.WaitOne();
	x = 1;
	for (int i = 1; i < 9; i++){
		Console.WriteLine("{0}: {1}",
		Thread.CurrentThread.Name, x);
		x++;
		Thread.Sleep(100);
	}
	waitHandler.Set();
}
...
static AutoResetEvent waitHandler = new AutoResetEvent(true);
static int x = 0;
for (int i = 0; i < 5; i++){
	var myThread = new Thread(Count);
	myThread.Name = "Поток " + i.ToString();
	myThread.Start();
}
```

Если у нас в программе используются несколько объектов AutoResetEvent, то мы можем использовать для отслеживания состояния этих объектов методы WaitAll и WaitAny, которые в качестве параметра принимают массив объектов класса WaitHandle - базового класса для AutoResetEvent

Mutex

```
public static void Count(){
      mutexObj.WaitOne();
      x = 1;
      for (int i = 1; i < 9; i++){
            Console.WriteLine("{0}: {1}",
            Thread.CurrentThread.Name, x);
            x++;
            Thread.Sleep(100);
      }
      mutexObj.ReleaseMutex();
}
...
static Mutex mutexObj = new Mutex();
static int x = 0;
for (int i = 0; i < 5; i++){
      Thread myThread = new Thread(Count);
      myThread.Name = "Поток " + i.ToString();
      myThread.Start();
}
```

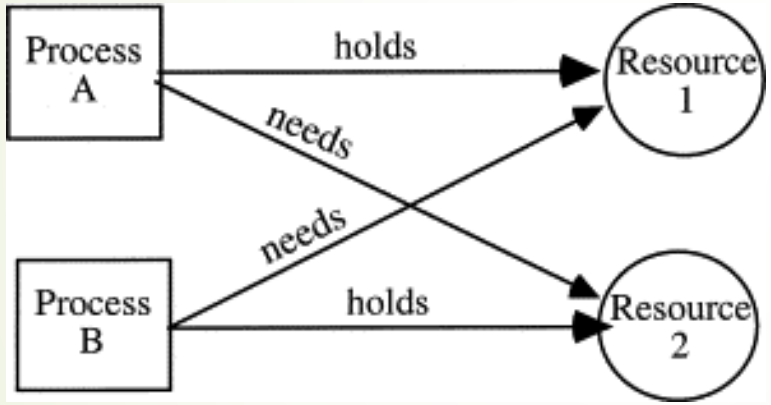
Использование мьютексов между процессами

```
static void Main(string[] args){
	bool existed;
	string guid = Marshal.GetTypeLibGuidForAssembly(Assembly.GetExecutingAssembly()).ToString();
	var mutexObj = new Mutex(true, guid, out existed);
	if (existed){
		Console.WriteLine("Приложение работает");
	}
	else{
		Console.WriteLine("Приложение уже было запущено!");
		return;
	}
}
```

Semaphore

```
class Reader{
        static Semaphore sem = new Semaphore(3, 3);
        Thread myThread;
        int count = 3;
        public Reader(int i){
                myThread = new Thread(Read);
                myThread.Name = "Читатель " + i.ToString();
                myThread.Start();
        }
        public void Read(){
                while (count > 0){
                        sem.WaitOne();
                        Console.WriteLine("{0} входит в библиотеку", Thread.CurrentThread.Name);
                        Console.WriteLine("{0} читает", Thread.CurrentThread.Name);
                        Thread.Sleep(1000);
                        Console.WriteLine("{0} покидает библиотеку", Thread.CurrentThread.Name);
                        sem.Release();
                        count--;
                        Thread.Sleep(1000);
        }
}
```

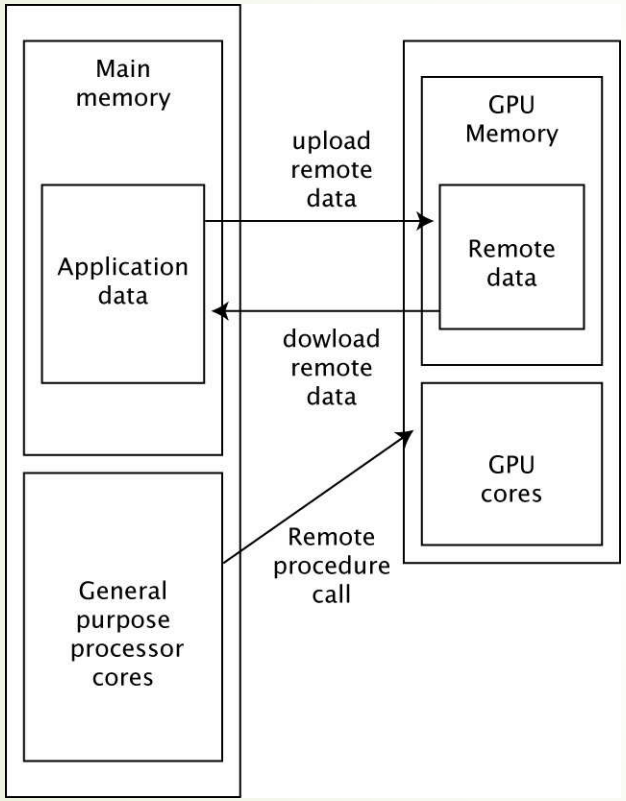
Deadlock (взаимная блокировка)

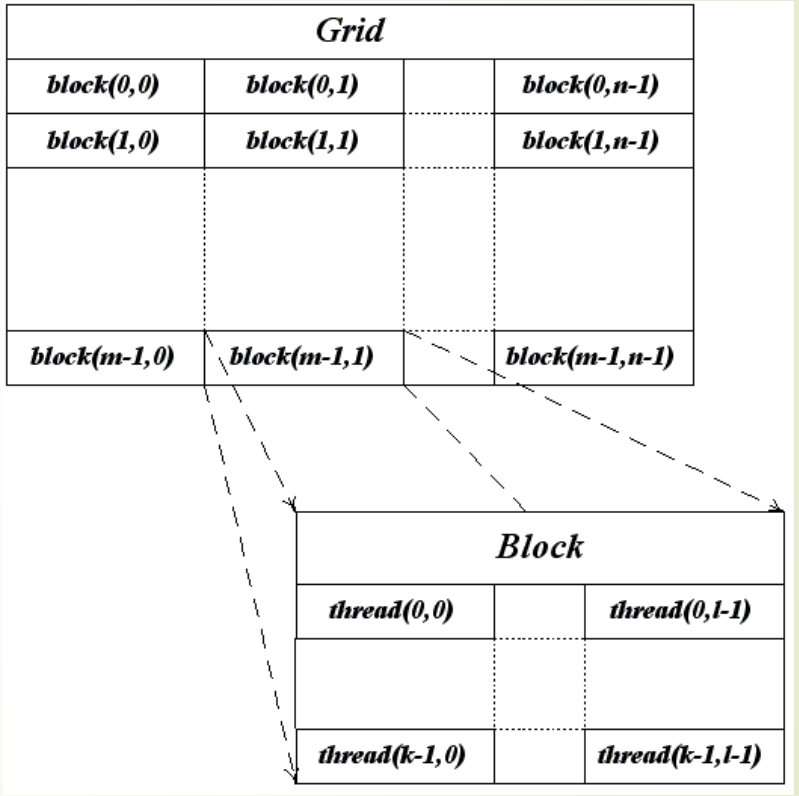


Таймеры

```
public static void Count(object obj){
    int x = (int)obj;
    for (int i = 1; i < 9; i++, x++){
        Console.WriteLine("{0}", x * i);
    }
}
...
int num = 0;
var tm = new TimerCallback(Count);
var timer = new Timer(tm, num, 0, 2000);
```

GP GPU


Main memory
Application data
General purpose processor cores
upload remote data
dowload remote data
Remote procedure call
GPU Memory
Remote data
GPU cores


Grid
block(0,0)
block(0,1)
block(0,n-1)
block(1,0)
block(1,1)
block(1,n-1)
block(m-1,0)
block(m-1,1)
block(m-1,n-1)
Block
thread(0,0)
thread(0,l-1)
thread(k-1,0)
thread(k-1,l-1)

Пример

```
#include <stdio.h>
#include <stdlib.h>

#define N 2

__global__ void MatAdd(int A[][N], int B[][N], int C[][N]){
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
```

```
int main(){
        int A[N][N] = {{1,2},{3,4}};
        int B[N][N] = {{5,6},{7,8}};
        int C[N][N] = {{0,0},{0,0}};
        int (*pA)[N], (*pB)[N], (*pC)[N];
        cudaMalloc((void**)&pA, (N*N)*sizeof(int));
        cudaMalloc((void**)&pB, (N*N)*sizeof(int));
        cudaMalloc((void**)&pC, (N*N)*sizeof(int));
        cudaMemcpy(pA, A, (N*N)*sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(pB, B, (N*N)*sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(pC, C, (N*N)*sizeof(int), cudaMemcpyHostToDevice);
        int numBlocks = 1;
        dim3 threadsPerBlock(N,N);
        MatAdd<<<numBlocks,threadsPerBlock>>>(pA, pB, pC);
        cudaMemcpy(C, pC, (N*N)*sizeof(int), cudaMemcpyDeviceToHost);
        …
        cudaFree(pA);
        cudaFree(pB);
        cudaFree(pC);
        return 0;
}
```