Разработка приложений на платформе .NET

Лекция 12

Многопоточное программирование

Процессы и потоки

- Процесс – выполняющаяся программа (экземпляр)
- Процесс может содержать один или несколько потоков
- Поток (нить, thread) – путь выполнения внутри исполняемого приложения
- При запуске приложения создается и запускается главный поток
- Любой поток может запускать дополнительные потоки
- Потоки выполняются параллельно и независимо
- Завершение процесса – завершение всех его потоков
- ~~Нет четкой корреляции между потоком операционной системы и управляемым потоком .NET~~
- Многопоточные приложения могут выполнятся и на однопроцессорном компьютере

Достоинства и недостатки

- При грамотном подходе может значительно ускорить работу приложения (только при многоядерной или много процессорной архитектуре)
- Позволяет повысить отзывчивость пользовательского интерфейса (даже при однопроцессорной архитектуре)
- Позволяет ускорить работу приложения за счет одновременного выполнения:
  - долгих удаленных операций (выполняющихся на других компьютерах)
    - Например, запрос к базе данных, к сервису или к интернет ресурсу
  - медленных, но мало затратных операций
    - Например, сохранение или чтение с диска

- Трудности разработки (дороговизна разработки)
  - Разбиение и оптимизация программы для многопоточной работы
  - Синхронизация потоков
  - Тестирование
- Трудности тестирования и отладки
  - Трудно обнаружимые ошибки
  - Невоспроизводимые ошибки
  - Непредсказуемые ошибки
- При неграмотном подходе может замедлить приложение
  - На создание и поддержание работы потоков тратятся ресурсы

Потоки в .NET

- Пространства имен
  - **System.Threading**
  - **System.Threading.Tasks**
  - **System.ComponentModel** (поток для **UI**, **BackgroundWorker**)
  - **System.Collections.Concurrent** (потокобезопасные коллекции)
- Класс **System.Threading.Thread**
  - Методы для работы с потоками
  - Статические члены для текущего потока
  - **static Thread Thread.CurrentThread** – текущий поток
- Единица кода для запуска в потоке – метод
  - В отдельном потоке всегда запускается какой-то метод

Запуск потока

- Необходимо создать метод, который будет выполнятся новым потоком
  - `public static void ThreadMethod() {…}`
- Создание экземпляра делегата на метод
  - **ThreadStart** – для запуска потока без параметров
  - **ParameterizedThreadStart** – для запуска потока с одним параметром (но параметр **object**)
- Создание потока и передача ему делегата на метод
  - `Thread thread= new Thread(new ThreadStart(threadMethod));`
- Запуск потока `thread.Start();`

Передача параметров потоку

- Использование делегата **ParameterizedThreadStart** вместо **ThreadStart**
- Передача только 1 параметра, но параметра типа **object**

```
public static void ThreadMethod(object o){..}
Thread thread =
        new Thread(new ParameterizedThreadStart(ThreadMethod));
thread.Start(obj);
```

- Использования замыкания и лямбда выражения

```
int i = 5;
Thread thread = new Thread( () => ThreadMethodWithInt( i ) ));
```

- Методы для выполнения в потоке ничего не возвращают

Демонстрации

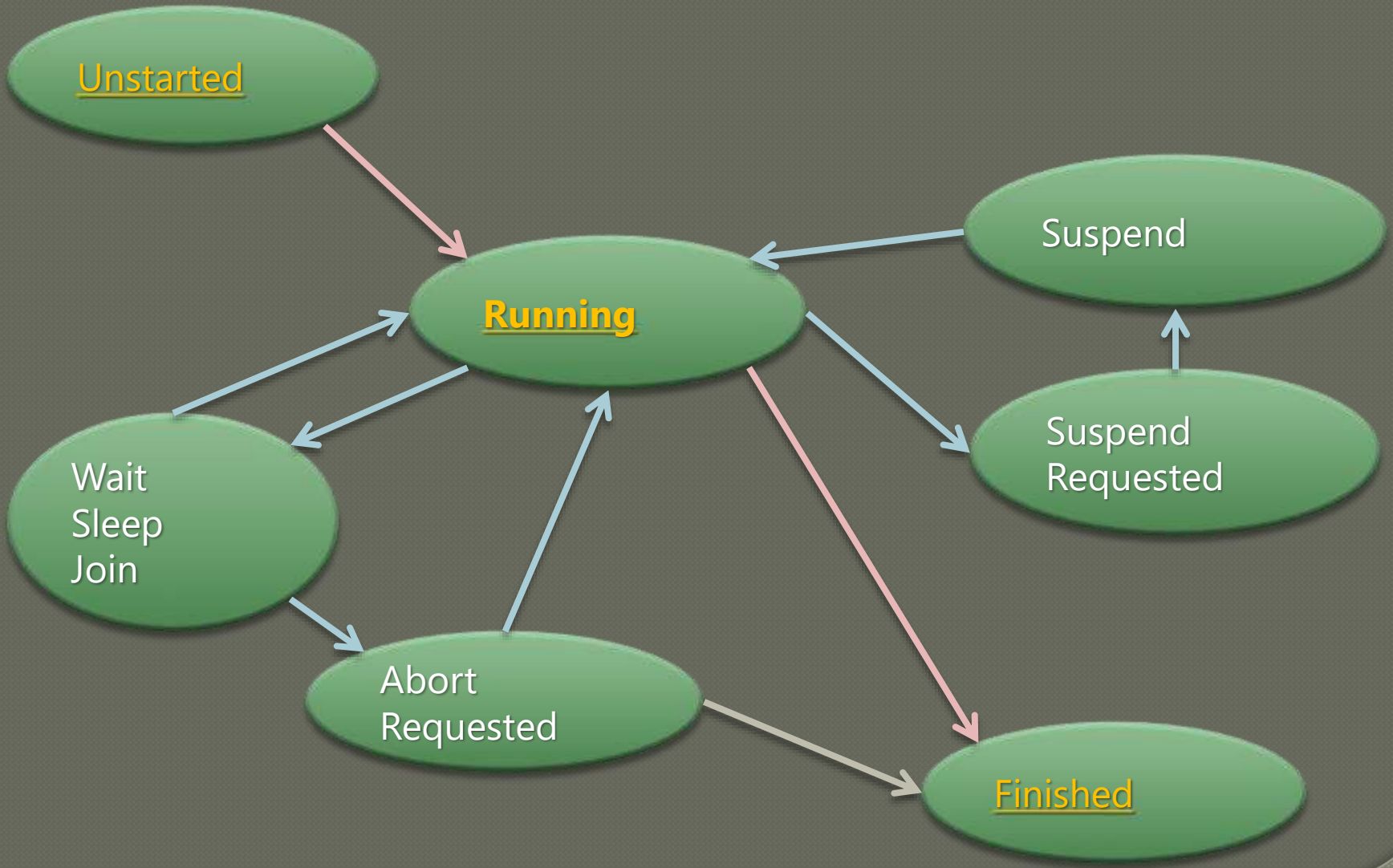
Thread

Передача параметров

Класс Thread

- Свойства потока
  - Name – имя потока (удобно использовать для отладки)
  - ManagedThreadId – уникальный ID потока
  - Priority – приоритет потока
  - IsAlive – поток запущен и не приостановлен
  - ThreadState – состояние потока
  - IsBackground – фоновый ли поток
  - IsThreadPoolThread – принадлежит ли поток пулу потоков CLR
- Полезные методы и свойства для работы с потоками
  - Thread.CurrentThread – ссылка на текущий поток (статическое вычислимое свойство)
  - Thread.Sleep() – заставляет поток ожидать указанное время (статический метод)
  - thread.Join() – заставляет ожидать текущий поток завершения указанного потока.
  - thread.Abort() – заставляет аварийно завершить поток

Состояния потоков



Завершение потока

- Поток завершится при выходе из метода
- thread.Abort() – аварийное завершение потока
  - При этом у прерываемого потока возникает исключение **ThreadAbortedException**
  - Прерываемый поток может обработать исключение **ThreadAbortedException**, но после этого исключение будут вызвано снова
- thread.AbortReset() – отмена прерывания потока (если успеть, пока поток еще аварийно не завершился)
- thread.Join() – блокировка текущего потока до завершения другого потока
- Завершенный поток нельзя запустить снова

Фоновые потоки

- Потоки:
  - Потоки переднего плана (по умолчанию)
  - Фоновые потоки
- Процесс не завершится пока есть работающие потоки переднего плана
- Фоновые потоки при завершении основного потока получают исключение **ThreadAbortedException** и будут завершены
- Необходима реализация безопасного завершения фонового потока
- Установка потока как фонового

```
thread.IsBackground = true;
```

Демонстрации

Завершение фонового потока

Пул потоков

- В среде выполнения уже существует несколько запущенных потоков – пул потоков
- Количество потоков связано с количеством процессоров.
- При использовании потока из пула потоков нет накладных расходов на создание потока
- В пуле потоки фоновые
- Класс **ThreadPool** – позволяет получить доступ к пулу потоков **.NET**
- Постановка задания в очередь
  - Создание экземпляра делегата **void WaitCallback(object state)**
  - Постановка в очередь **ThreadPool.QueueUserWorkItem**
  - **(new WaitCallback(threadMethod), obj);**
- Переданное задание уже нельзя отменить
- Обычно используют класс **Task** вместо класса **ThreadPool**

Демонстрации

Пул потоков

Асинхронный вызов методов

- Любой делегат имеет помимо метода для синхронного вызова – **Invoke**(), методы для асинхронного вызова **BeginInvoke**(), **EndInvoke**()

  Func <string, double, int> f = ….

  IAsyncResult f.BeginInvoke(string s, double d, AsyncCallback callback, object obj) – начинает вызов и передает параметры **string, double**

  int f.EndInvoke(IAsyncResult ires) – ожидает завершения и возвращает значение

- **AsyncCallback callback** – делегат будет вызван при окончании вычисления

- Выполнение в пуле потоков

Интерфейс IAsyncResult

- Свойство **bool IsComplated** – завершено ли вычисление
- Свойство **object AsyncState** – позволяет передавать параметры для последующей идентификации вызванного метода

Демонстрации

Асинхронный вызов делегата

Классы Task и Task<T>

- Простой запуск выполнения действия в **Thread Pool**
- **Task** - класс для вызова метода, ничего не возвращающего, а **Task<TResult>** для возвращающего результат **TResult**
- Запуск таски – **Start()**. Конструктор – настройка таски
  - `void inc() { … }`
  - `Task t = new Task(inc);`
  - `t.Start();`

  - `Task<int> t = new Task<int>(GetInt);`
  - `t.Start();`

- Быстрый старт заданий (рекомендуемый) **Task.Factory.StartNew()**
  - `Task  t = Task.Factory.StartNew(inc);`
  - `Task<int> t = Task<int>.Factory.StartNew(GetInt);`
  - `Task<int> t = Task<int>.Factory.StartNew(Add, new object[] { 5,7 } );`

- Получение результата по окончании таски **Task<T>** - свойство **Result** (если результат не готов, то текущий поток приостановиться до получения готового результата)
  - `int res = t.Result; // если t – Task<int>`

Классы Task и Task<T>

- Продолжение выполнения ContinueWith();
- Метод будет выполняться по завершении таски (сама таска пойдет на вход методу)
  - void inc() { … }
  - void a(Task t) { … }
  - Task task = Task.Factory.StartNew(inc);
  - task.ContinueWith(a);

- Синхронизация тасок
  - Ожидание завершения таски Wait();
    - t.Wait();
  - Ожидание завершения всех тасок Task.WaitAll()
    - Task.WaitAll(task1, task2, task3)
  - Ожидание завершения хотя бы одной тасок Task.WaitAny()
    - Task.WaitAny(task1, task2, task3)

Отмена Task и Task<T>

- Возможна отмена таски – передача токена отмены **CancellationToken** при старте таски

```
static void Main()
{
    CancellationTokenSource cts = new CancellationTokenSource();
    CancellationToken token = cts.Token;
    Task.Factory.StartNew(() => LongMethod(100, token), token);

    // выполнялась какая-то параллельная логика и решили остановить асинхронную задачу
    cts.Cancel();
 }

private static void LongMethod(int count, CancellationToken cancellationToken)
{
    for (int i = 0; i < 100; i++)
   {
        Console.WriteLine(i);
        // проверка, не отменена ли задача
        if (cancellationToken.IsCancellationRequested) return;
        Thread.Sleep(1000);
   }
}
```

Класс Parallel

- Parallel.For(initvalue, endvalue, Action<T>); - Выполнение цикла в максимально возможном числе потоков (ThreadPool). В цикле выполняется делегат Action<T> (который принимает 1 параметр T и, ничего не возвращает). Числом потоков управляет CLR
- Parallel.ForEach<T>(IEnumerable<T>, Action<T>); - Выполнение делегата Action<T> над всеми элементами перечисления в максимально возможном числе потоков. Числом потоков управляет CLR
  - List<int> l = new List<int>();
  - public void dec(int i) {}
  - Parallel.For(0, 10, dec);
  - Parallel.ForEach<int>(l, dec);
  - Parallel.ForEach(l, dec);
- Parallel.Invoke(params Action[] actions) – выполнение делегатов в отдельных потоках, если возможно
  - Parallel.Invoke(Print, PrintToScreen, SendToEmail, () => Console.WriteLine("Печатаем"));
- Класс ParallelOptions может использоваться для подстройки операций Parallel
  - *MaxDegreeOfParallelism* – ограничивает максимально число одновременно выполняющихся задач в классом Parallel.
  - *CancellationToken* – позволяет отменять задания, выполняющиеся классом Parallel

Демонстрации

Parallel

Task

Порядок выполнения потоков непредсказуем

- Потоки выполняются параллельно и независимо. Нельзя предсказать очередность выполнения блоков кода потоками.

```
static void Main()
    {
        Thread t = new Thread(Write1);
       t.Start();
        while (true) Console.Write("-"); // Все время печатать '-'
    }

static void Write1()
    {
       while (true) Console.Write("1"); // Все время печатать '1'
    }
```

Независимый стек локальных переменных

- У каждого потока свой стек локальных переменных. Они независимые.

```
static void Main()
      {
         new Thread(Go).Start();      // Выполнить Go() в новом потоке
         Go();                        // Выполнить Go() в главном потоке
      }

static void Go()
      {
         // Определяем и используем локальную переменную 'cycles'
          for (int cycles = 0; cycles < 5; cycles++) Console.Write('+');
      }
```

Общие переменные объекта

- Вместе с тем потоки разделяют данные, относящиеся к тому же экземпляру объекта

```
class TestClass  {
      bool done = false;
      public void Go() {
         if (!done) { done = true; Console.WriteLine("Done"); }
      }   }

class ThreadTest {
      static void Main() {
         TestClass testClass = new TestClass();
         new Thread(testClass.Go).Start();
         testClass.Go();
}  }
```

Операции не являются атомарными

```
class Increment {
      decimal l = 0;
      public void inc()  {
         for (int i = 0; i < 100000; ++i) l = l +1;
                              Console.WriteLine(l);
     }   }

class Program {
      static void Main(string[] args)  {
            Increment i = new Increment ();
            for (int j = 0; j < 10; ++j)
            new Thread(i.inc).Start();
     } }
```

Присваивание ссылочных типов атомарно (при любой разрядности ОС)

Итого

- Потоки выполняются параллельно и независимо. Нельзя предсказать какой поток отработает быстрее.
- У каждого потока свой собственный стек. Собственные неразделяемые локальные переменные
- Потоки разделяют нелокальные переменные, доступные им по области видимости
- Операции неатомарные

Синхронизация потоков

- Класс **Volatile**
- Класс **Interlocked**
- Конструкция **lock**
- Класс **Monitor**
- Классы **ReaderWriterLock**, **ReaderWriterLockSlim**
- Класс **Mutex**
- Семафоры
- Наследники от **EventWaitHandle**

Volatile

- Класс **Volatile**
  - Volatile.Read() - считывает значение указанного поля. Добавляет барьер в памяти, предотвращая изменение порядка операций процессора с памятью: если после вызова этого метода следует операции чтения или записи, процессор не сможет выполнить их перед вызовом этого метода.
  - `bool  isRunning = Volatile.Read(ref _isBusy);`
  - Volatile.Write() - записывает заданное значение в поле. Добавляет барьер в памяти, предотвращая изменение порядка операций процессора с памятью: если до вызова этого метода используются операции чтения или записи, процессор обязан будет выполнить их до вызова этого метода.
    - `Volatile.Write(ref _isBusy, true);`
- Модификатор поля volatile – все операции с полем будут выполнятся как **Volatile.Read()** и **Volatile.Write()**
  - `public volatile bool _isBusy;`

Класс Interlocked

- Атомарные операции. Статические члены
  - Interlocked.Increment(ref i); i – long или int
  - Interlocked.Decrement(ref i); i – long или int
  - Interlocked.Add(ref i1, i2); Переменные int, long

  - Interlocked.Exchange(ref i, value);
  - Interlocked.Exchange<T>(ref T i, T value);

  - Interlocked.CompareExchange(ref i, value, compared);
    - Если i == compared, то i = value. Переменные типов: int, long, float, double, object
  - Interlocked.CompareExchange <T> (ref T i, T value, T compared) – для ссылочных типов

Демонстрации

Interlocked

Конструкция lock

- Необходимо определить **единую** доступную всем потокам **ссылочную** переменную (экземпляр объекта)
- Если объект в переменной не блокирован, то поток проходит беспрепятственно через оператор **lock**, блокируя объект
- Если объект в переменной блокирован, то поток остановится на операторе **lock** и будет ожидать пока другой поток не выйдет из конструкции **lock**

- Например:

```
public object lockObject = new object();

lock (lockObject)
{
        // Операции с разделяемыми ресурсами
}
```

- Каждый объект в куче имеет индекс блока синхронизации, который и используется для блокировок при синхронизации потоков

- Не используйте **string** из-за его неизменяемой структуры и интернирования

Необходимо

- Как можно быстрее освобождать блокировку
- Избегать взаимоблокировок (deadlock)

```
lock (A)
{
    lock (B)
    {
    }
}
```

```
lock (B)
{
    lock (A)
    {
    }
}
```

- Блокировать только ссылочную переменную
- Экземпляр блокируемого объекта должен быть один и тот же для всех потоков

Демонстрации

lock

Класс Monitor

- **Monitor.Enter(lockObject);** - ожидание и вход потока в критическую секцию. Увеличение количества блокировок на 1.
- **Monitor.Exit(lockObject);** - выход из критической секции. Уменьшение количества блокировок на 1.
- Конструкция **lock** реализуется через класс **Monitor.**
- Необходимо самостоятельно следить за количеством установок / снятия блокировок.

ReaderWriterLock

- Очереди читателей и писателей.
- Много потоков могут читать данные
- Только один поток может захватить объект для записи.

```
ReaderWriterLock rwl = new ReaderWriterLock();
rwl.AcquireReaderLock(timeout);
rwl.AcquireWriterLock(timeout);
rwl.UpgradeToWriterLock(timeout);
rwl.DowngradeFromWriterLock(ref cokie);
rwl.ReleaseReaderLock();
rwl.ReleaseWriterLock();
```

ReaderWriterLockSlim

- Аналогичен **ReaderWriterLock**
- Короткие блокировки реализуются как инструкции **Spin**
- Но имеет еще одно доп. Состояние:
  - **Read mode**
  - **Write mode**
  - **Upgradeable mode**

```
ReaderWriterLockSlim sl = new ReaderWriterLockSlim();
sl.EnterReadLock();
sl.ExitReadLock();
sl.EnterWriteLock();
sl.ExitWriteLock();
sl.EnterUpgradeableReadLock();
sl.ExitUpgradeableReadLock();
```

Mutex

- Тяжеловесный. Уровня ОС
- Может использоваться для синхронизации Процессов.
- **Mutex mutex = new Mutex(false, "MyUniqueMutex");**
- **mutex.WaitOne();**
- **mutex.ReleaseMutex();**
- **mutex.Close();**
- Есть перегруженные методы с ограниченным временем ожидания блокировки.

Демонстрации

1. Приложение, допускающее только один запущенный экземпляр приложения
2. Синхронизация процессов

Семафоры

- Позволяют обеспечит доступ определенного числа потоков к разделяемым ресурсам
- Объект уровня ОС. Тяжеловесный
- Может использоваться для синхронизации Процессов.
- Semaphore sem = new Semaphore(initBlocks, maxBlocks, “MySemaphore");
- sem.WaitOne();
- sem.Release();
- sem.Close();

Демонстрации

Синхронизация процессов

Класс EventWaitHandle

- Наследники:
  - AutoResetEvent
  - ManualResetEvent
- Раздельно устанавливают блокировки и снимают.
- Один поток может ожидать, а другой по своей логике может его пропустить дальше
- Сообщает другому потоку, что событие произошло и тот может выполнять свои действия

- AutoResetEvent are = new AutoResetEvent(bool начальное состояние);
- are.Set(); - Снимает блокировку
- are.WaitOne(); - Ожидать снятия блокировки
- are.Reset(); - Устанавливает блокировку

- AutoResetEvent после прохода WaitOne автоматически устанавливает блокировку (Reset). ManualResetEvent – нет.

Демонстрации

Синхронизация

Работа с коллекциями

- Некоторые коллекции содержат объект для синхронизации (для использования с **lock**) – **SyncRoot**

```
int[] col = new int[2];
……
lock(col.SyncRoot)
{
        // работа с массивом
}
```

- Имеются специальные коллекции, доступ к которым из разных потоков не требует синхронизации, поскольку они содержат внутренние механизмы синхронизации
- **ConcurrentQueue<T>** - очередь
- **ConcurrentStack<T>** - стек
- **ConcurrentDictionary<TKey, TValue>** - словарь
- **ConcurrentBag<T>** - простой список

- **BlockingCollection<T>** - реализация **producer/consumer** паттерна

Демонстрации

Работа с коллекциями

Запуск процесса

- Класс Process
- Запуск процесса
  - Process.Start(…)

```
Process process = Process.Start(@"d:\Программка.exe");
```

  - process.Start();

```
Process process = new Process(@" d:\Программка.exe ");
process.Start();
```

- Ожидание завершения процесса WaitForExit()

```
process.WaitForExit();
```

- Получение информации о запущенных процессах Process.GetProcesses()
  - Process[] processes = Process.GetProcesses();
- Завершение процесса Kill()
  - process.Kill();

Демонстрации

Запуск и контроль другого процесса