Многопоточное программирование

Часть 1: высокоуровневая многопоточность

Юрий Литвинов
yurii.litvinov@gmail.com

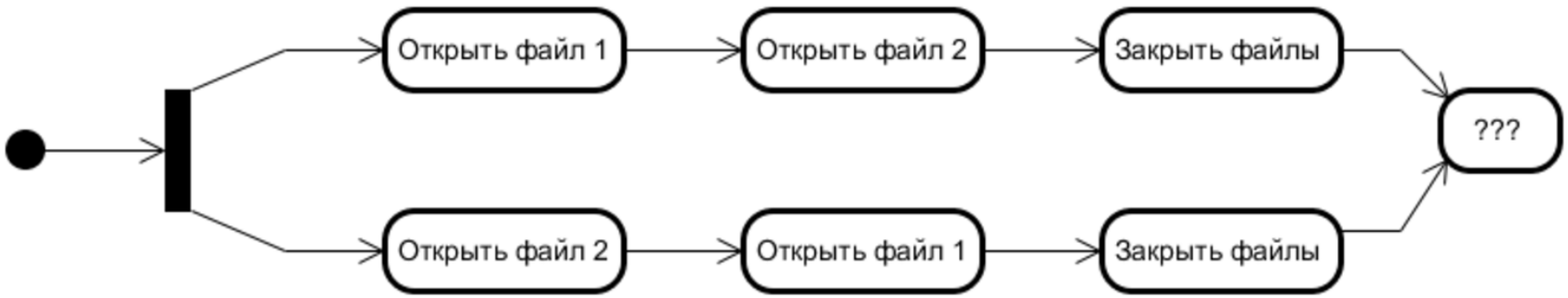
8

Многопоточное программирование вообще

- Плюсы
  - Не вешать пользовательский интерфейс
  - Равномерно распределять вычислительно сложные задачи по ядрам
  - Выполнять одновременно несколько блокирующих операций ввода-вывода
- Минусы
  - Тысяча способов прострелить себе ногу
  - Не всегда многопоточная программа работает быстрее однопоточной

Race condition

Deadlock


Открыть файл 1
Открыть файл 2
Закрыть файлы
???
Открыть файл 2
Открыть файл 1
Закрыть файлы

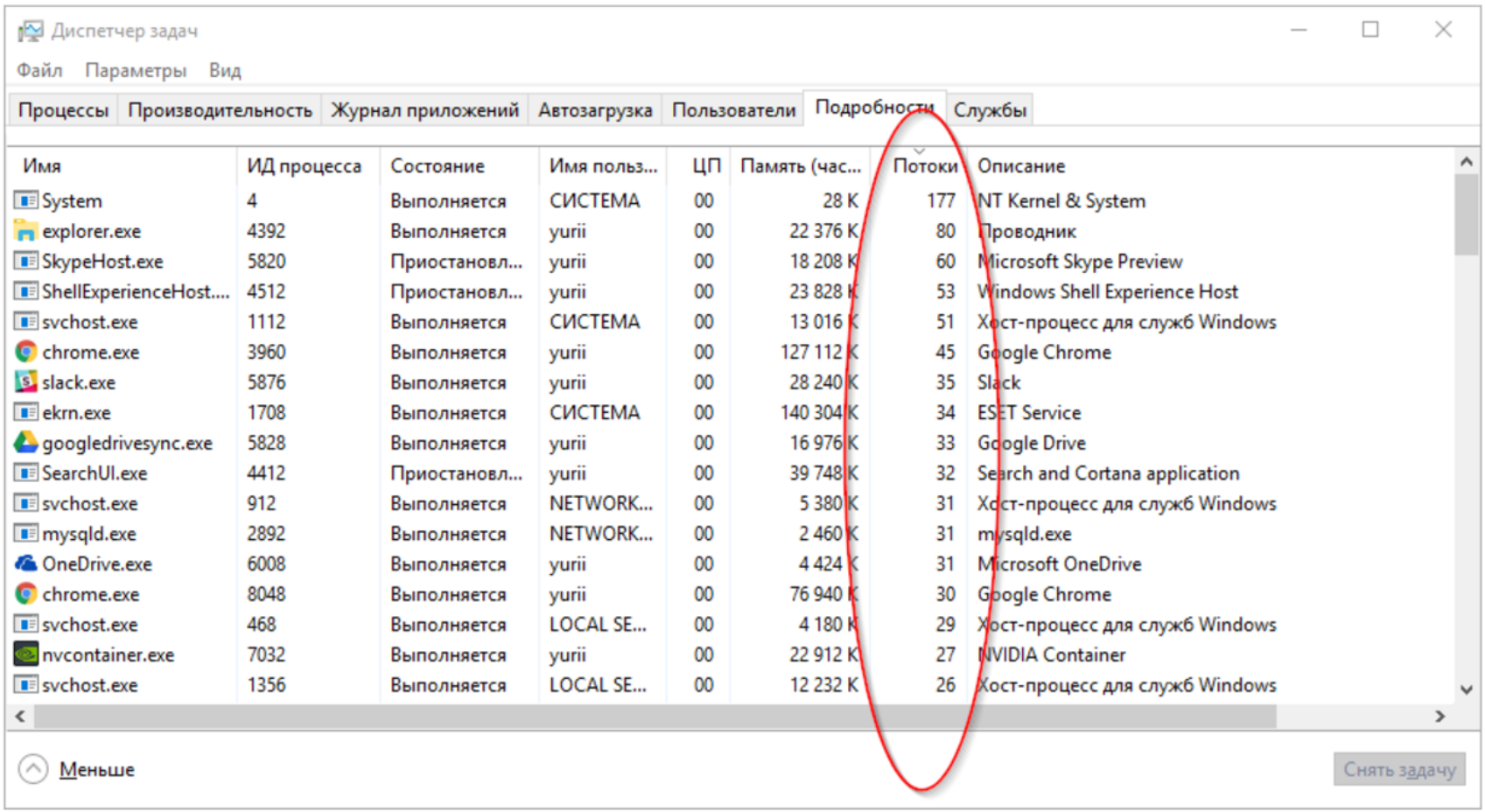
Поток в Windows

- Thread Kernel Object (1240 байт)
- Thread environment block (TEB) (4 Кб)
- User-mode stack (1 Мб)
- Kernel-mode stack (24 Кб)

Ещё для каждой dll-ки, загруженной для процесса при старте или остановке потока вызывается DllMain с параметрами DLL_THREAD_ATTACH и DLL_THREAD_DETACH

Квант времени — 20-30 мс, после чего происходит *переключение контекстов*

Как делать не надо



System.Threading.Thread

```
Thread dedicatedThread = new Thread(ComputeBoundOp);
dedicatedThread.Start(5);
Thread.Sleep(10000);  // Симуляция 10 секунд какой-то работы
dedicatedThread.Join();  // Ждём второй поток
...

private static void ComputeBoundOp(Object state) {
   Console.WriteLine($"In ComputeBoundOp: state={state}");
   Thread.Sleep(1000);  // Симуляция секунды каких-то вычислений
}
```

Планировщик

- Раз в квант времени (или чаще) выбирает поток для исполнения
  - Рассматриваются только потоки, не ждущие чего-либо
- НЕ реальное время
  - Нельзя делать предположения, когда потоку дадут поработать
- Из рассматриваемых потоков выбираются только те, у кого наибольший приоритет
  - Приоритеты потоков от 0 до 31, обычно 8
- Есть ещё приоритеты процессов: Idle, Below, Normal, Normal, Above Normal, High и Realtime
- Относительные приоритеты потоков: Idle, Lowest, Below Normal, Normal, Above Normal, Highest и Time-Critical
  - Истинный приоритет получается из относительного приоритета и приоритета процесса

Foreground- и Background-потоки

- Когда все Foreground-потоки завершили работу, рантайм останавливает все Background-потоки и заканчивает работу приложения
- Thread по умолчанию создаётся как Foreground
  - Способ прострелить себе ногу №1: создать foreground-поток и забыть о нём, приложение будет висеть в списке задач и не завершится

Способ прострелить себе ногу №2: создать background-поток и не дать ему доработать:

```
Thread t = new Thread(Worker);
t.IsBackground = true;
t.Start();
Console.WriteLine("Returning from Main");
```

Пул потоков

- Содержит набор заранее созданных потоков, которые могут исполнять задачи
- Управляется рантаймом
  - Новые потоки создаются при необходимости
  - Потоки автоматически удаляются, если они долго не используются и потоков больше, чем надо
  - “Сколько надо” рантайм определяет по количеству доступных ядер процессора
- Используется в .NET практически повсеместно
  - Идеологически многопоточное приложение оперирует не потоками, а задачами и асинхронными операциями
- Все потоки из пула — Background

Пример

```
public static void Main() {
   Console.WriteLine("Main thread: queuing an asynchronous operation");
   ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5);
   Console.WriteLine("Main thread: Doing other work here...");
   Thread.Sleep(10000);  // Симуляция работы в главном потоке
   Console.WriteLine("Hit <Enter> to end this program...");
   Console.ReadLine();
}

private static void ComputeBoundOp(Object state) {
   Console.WriteLine("In ComputeBoundOp: state={0}", state);
   Thread.Sleep(1000);  // Симуляция работы в потоке из пула
}
```

Контекст исполнения

Ассоциированная с потоком структура данных, где хранятся разные свойства потока (например, свойства безопасности):

```
public static void Main() {
   CallContext.LogicalSetData("Name", "Jeffrey");
   ThreadPool.QueueUserWorkItem(
      state => Console.WriteLine("Name={0}", CallContext.LogicalGetData("Name")));
   ExecutionContext.SuppressFlow();
   ThreadPool.QueueUserWorkItem(
      state => Console.WriteLine("Name={0}", CallContext.LogicalGetData("Name")));
   ExecutionContext.RestoreFlow();
   ...
   Console.ReadLine();
}
```

Отмена операций

- CancellationToken — отдаётся потоку, он должен сам проверять состояние токена и прерваться, если запрошена отмена
  - Может прерваться не мгновенно, проверка возможна только время от времени
- CancellationTokenSource — умеет производить CancellationToken-ы, может выставлять флаг отмены для всех созданных CancellationToken-ов, остаётся в основном потоке

Пример

```
public static void Main() {
   CancellationTokenSource cts = new CancellationTokenSource();
   ThreadPool.QueueUserWorkItem(o => Count(cts.Token, 1000));
   Console.ReadLine();
   cts.Cancel();
   Console.ReadLine();
}

private static void Count(CancellationToken token, Int32 countTo) {
   for (Int32 count = 0; count < countTo; count++) {
      if (token.IsCancellationRequested) {
         break;
      }
      Thread.Sleep(200);
   }
}
```

Полезные вещи CancellationToken

- CancellationToken.None
- CancellationToken.Register:

```
var cts = new CancellationTokenSource();
cts.Token.Register(() => Console.WriteLine("Canceled 1"));
cts.Token.Register(() => Console.WriteLine("Canceled 2"));
```

  - Возвращает CancellationTokenRegistration, реализующий IDisposable
- CancellationTokenSource.CreateLinkedTokenSource
- CancellationTokenSource.CancelAfter

Task

- Абстракция задачи, которая может быть выполнена в отдельном потоке
- Эквивалентные строки кода:

```
ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5);
new Task(ComputeBoundOp, 5).Start();
Task.Run(() => ComputeBoundOp(5));
```

- Позволяет ждать окончание задачи и получать результат
- Тоже важен для реализации некоторых вещей в C#, но часто используется и независимо

Пример

```
private static Int32 Sum(Int32 n) {
    Int32 sum = 0;
    for (; n > 0; n--)
        checked { sum += n; }
    return sum;
}
...
Task<Int32> t = new Task<Int32>(n => Sum((Int32)n), 1000000000);
t.Start();
t.Wait();  // t.Result сам делает Wait(), так что тут это только для иллюстрации
Console.WriteLine("The Sum is: " + t.Result);
```

Отмена Task-а

```
private static Int32 Sum(CancellationToken ct, Int32 n) {
    Int32 sum = 0;
    for (; n > 0; n--) {
        ct.ThrowIfCancellationRequested();
        checked { sum += n; }
    }
    return sum;
}
```

Кидает OperationCanceledException в основной поток при обращении к результату (на самом деле, AggregateException с OperationCanceledException)

Полезные вещи Task-а

- ContinueWith:

```
Task<Int32> t = Task.Run(() => Sum(CancellationToken.None, 10000));
Task cwt = t.ContinueWith(task => Console.WriteLine("The sum is: " + task.Result));
```

- Родительские задачи:

```
Task<Int32[]> parent = new Task<Int32[]>(() => {
   var results = new Int32[3];
   new Task(() => results[0] = Sum(10000), TaskCreationOptions.AttachedToParent).Start();
   new Task(() => results[1] = Sum(20000), TaskCreationOptions.AttachedToParent).Start();
   new Task(() => results[2] = Sum(30000), TaskCreationOptions.AttachedToParent).Start();
   return results;
});
var cwt = parent.ContinueWith(
      parentTask => Array.ForEach(parentTask.Result, Console.WriteLine));
```

TaskFactory

```
var cts = new CancellationTokenSource();
var tf = new TaskFactory<Int32>(cts.Token,
      TaskCreationOptions.AttachedToParent,
      TaskContinuationOptions.ExecuteSynchronously,
      TaskScheduler.Default);

   var childTasks = new[] {
   tf.StartNew(() => Sum(cts.Token, 10000)),
   tf.StartNew(() => Sum(cts.Token, 20000)),
   tf.StartNew(() => Sum(cts.Token, Int32.MaxValue))
};
```

TaskScheduler

- Класс, позволяющий управлять тем, как Task-и обрабатываются пулом потоков (и пулом потоков ли вообще)
  - По умолчанию таски ставятся в очередь в пуле потоков
- Бывает полезно, например, чтобы задача могла модифицировать элементы GUI
  - Это можно делать только из главного потока (который создал GUI)

```
Task<Int32> t = Task.Run(() => Sum(m_cts.Token, 20000), m_cts.Token);
t.ContinueWith(task => Text = "Result: " + task.Result,
    CancellationToken.None,
    TaskContinuationOptions.OnlyOnRanToCompletion,
    TaskScheduler.FromCurrentSynchronizationContext());
```

Более высокоуровневые вещи

```
for (Int32 i = 0; i < 1000; i++) DoWork(i);
```

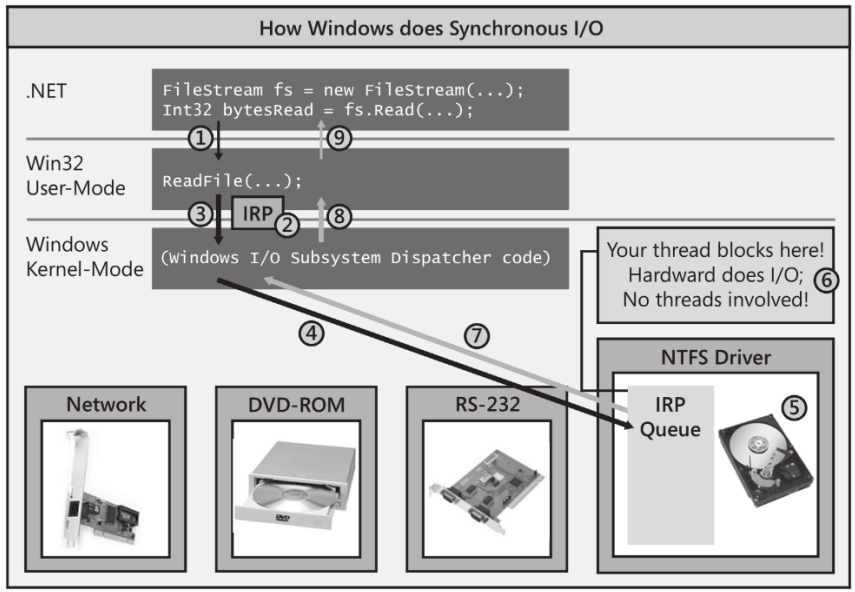
```
Parallel.For(0, 1000, i => DoWork(i));
```

Есть ещё:

- Parallel.ForEach(collection, item => DoWork(item));
- Parallel.Invoke(
  () => Method1(),
  () => Method2(),
  () => Method3());
- ParallelQuery<T> и LINQ

async/await

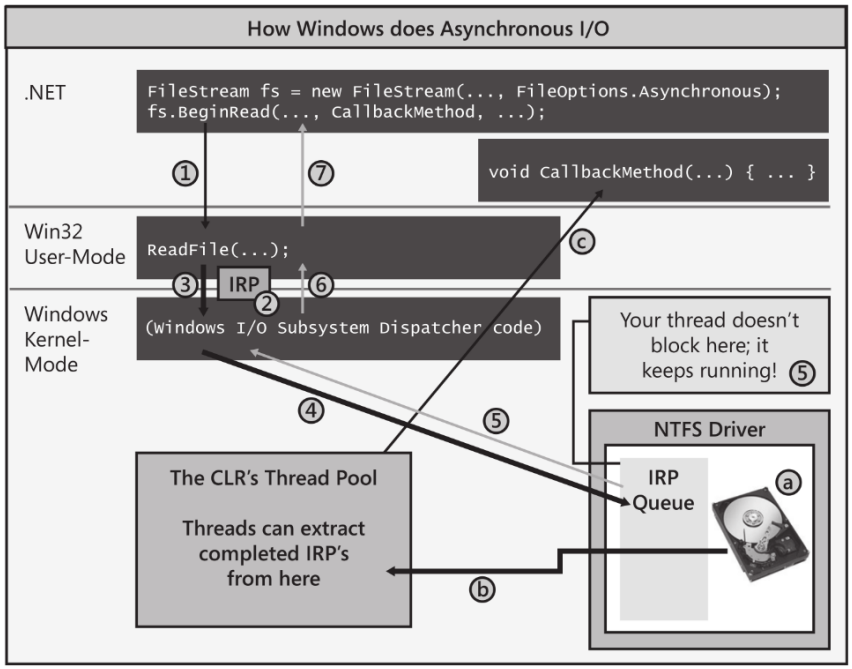
- Task и пул потоков хороши для дорогих по времени операций
- Чаще поток ждёт окончания операции ввода-вывода
- Блокирующий ввод-вывод “вешает” поток, заставляя пул потоков создавать новые



(Рисунок из Jeffrey Richter. CLR via C#)

async/await (2)

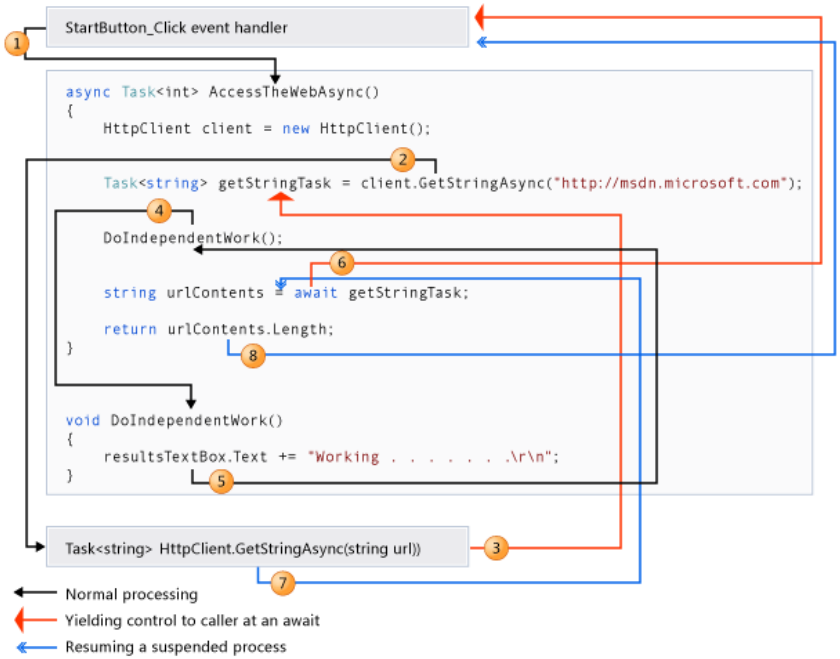
- Асинхронные операции ввода-вывода не блокируют поток, возвращая управление тут же
  - Данные, естественно, не готовы
- Старая модель в .NET — Begin...() и End...()
  - Begin...() инициирует операцию, принимая колбэк, где можно использовать End...(), чтобы забрать результат



(Рисунок из Jeffrey Richter. CLR via C#)

async/await (3)

- Новая модель: async/await
- Требуется поддержка компилятора
- Можно понимать как сопрограмму
- На самом деле, генерируется конечный автомат
  - Запоминает, на каком await сейчас мы находимся
  - Следит за исключениями



(Рисунок из MSDN)

Пример

```
internal sealed class Type1 { }
internal sealed class Type2 { }
private static async Task<Type1> Method1Async() { ... }
private static async Task<Type2> Method2Async() { ... }

private static async Task<String> MyMethodAsync(Int32 argument) {
   Int32 local = argument;
   try {
      Type1 result1 = await Method1Async();
      for (Int32 x = 0; x < 3; x++) {
         Type2 result2 = await Method2Async();
      }
   }
   catch (Exception) { Console.WriteLine("Catch"); }
   return "Done";
}
```

Особенности

- Может возвращать только Task, Task<Result> или **void**
  - **void** используется для асинхронных обработчиков событий
- Любой Task можно ждать await-ом, любой async можно не ждать
  - Вызов Result у результата async-метода заставляет его исполниться синхронно
- Работает только с .NET 4.5 и C# 5
  - Microsoft.BCL, если надо поддержать более старый рантайм

Async-методы в стандартной библиотеке

- System.IO.Stream и потомки: ReadAsync, WriteAsync, FlushAsync, CopyToAsync
- System.IO.TextReader и потомки: ReadAsync, ReadLineAsync, ReadToEndAsync, ReadBlockAsync
- System.IO.TextWriter и потомки: WriteAsync, WriteLineAsync, FlushAsync
- System.Net.Http.HttpClient: GetAsync, GetStreamAsync, GetByteArrayAsync, PostAsync, PutAsync, DeleteAsync и т.д.
- System.Net.WebRequest и потомки: GetRequestStreamAsync и GetResponseAsync
- System.Data.SqlClient.SqlCommand: ExecuteDbDataReaderAsync

Ещё один способ прострелить себе ногу

```
private sealed class MyWpfWindow : Window {
    public MyWpfWindow() { Title = "WPF Window"; }

    protected override void OnActivated(EventArgs e) {
        String http = GetHttp().Result; // Синхронно вызываемся
        base.OnActivated(e);
    }

    private async Task<String> GetHttp() {
        HttpResponseMessage msg =
              await new HttpClient().GetAsync("http://google.com/");
        return await msg.Content.ReadAsStringAsync();  // Никогда не дойдём сюда
    }
}
```