

# Введение, многопоточное программирование

Юрий Литвинов  
yurii.litvinov@gmail.com

06.09.2019г

# О чём этот курс

- ▶ Кратко про почти всё, что обязательно знать любому прикладному программисту
  - ▶ Многопоточное программирование
  - ▶ Сетевое программирование
  - ▶ Веб-программирование
  - ▶ Продвинутое GUI-программирование
  - ▶ Работа с базами данных
  - ▶ Рефлексия
- ▶ Язык программирования — C#
  - ▶ Немного подробностей про внутреннее устройство .NET тоже будет в курсе

# Отчётность

- ▶ Домашка
- ▶ Две контрольные
- ▶ Доклады (-1 домашка) (если успеем)
- ▶ Учебная практика

# Учебная практика

- ▶ Отдельный зачёт
- ▶ Программная реализация достаточно большой и полезной задачи
- ▶ Пишется весь семестр
- ▶ В конце — защита с презентацией
- ▶ Может быть групповой
- ▶ Где брать темы
  - ▶ Продолжать начатое в летней школе
  - ▶ Студпроекты
  - ▶ Придумать самим
  - ▶ Взять что-нибудь у кого-нибудь поблизости
    - ▶ Робототехника
    - ▶ Формальные методы
    - ▶ Machine Learning

# Многопоточное программирование

Зачем это нужно:

- ▶ Оптимально использовать ресурсы процессора
  - ▶ Одноядерных процессоров практически не бывает
- ▶ Использовать асинхронные операции ввода-вывода
- ▶ Не “вешать” GUI
- ▶ Показывать прогресс

Потенциальные проблемы:

- ▶ Тысяча способов прострелить себе ногу
  - ▶ Ошибки могут воспроизводиться раз в тысячу лет и их невозможно обнаружить статически
- ▶ Не всегда многопоточная программа работает быстрее однопоточной

# Процессы и потоки

- ▶ Процесс — исполняющаяся программа
  - ▶ Загруженный в память .exe-шник со всеми его .dll-ками или аналогичные понятия
  - ▶ Имеет выделенные для него системные ресурсы:
    - ▶ Память
    - ▶ Открытые файлы
    - ▶ Открытые сетевые соединения
    - ▶ ...
- ▶ Поток — единица параллельной работы
  - ▶ Существует внутри процесса
  - ▶ Имеет свой стек и состояние регистров процессора
  - ▶ Все потоки внутри процесса разделяют общие ресурсы (например, память)

# Параллельное программирование

- ▶ Параллельная программа может быть:
  - ▶ Многопроцессной
    - ▶ Несколько процессов, возможно, несколько потоков в каждом
  - ▶ Многопоточной
- ▶ Многопроцессные программы:
  - ▶ Могут исполняться на разных компьютерах
  - ▶ Пример — веб-приложения
  - ▶ Сложное и медленное взаимодействие между процессами
- ▶ Многопоточные программы:
  - ▶ Могут исполняться только на одном компьютере (нужна общая память)
  - ▶ Быстрое общение между потоками через общую память
  - ▶ Потоки могут портить состояние друг другу

# Внезапно, операционные системы

Функции операционной системы:

- ▶ Предоставлять упрощённый доступ к оборудованию
  - ▶ Файловая система
  - ▶ Драйвера
- ▶ Управлять ресурсами компьютера
  - ▶ Виртуальная память
  - ▶ Планировщик



# Планировщик

- ▶ Управляет распределением процессорного времени между процессами и потоками
- ▶ Каждому потоку выделяется квант времени, прерывание по таймеру
- ▶ Поток может отдать ядро процессора до истечения кванта
  - ▶ Сам
  - ▶ Блокирующая операция ввода-вывода
  - ▶ Подгрузка страницы памяти из свопа
  - ▶ Аппаратное прерывание
- ▶ Хитрые алгоритмы планирования
  - ▶ Обеспечение максимального быстродействия при справедливом планировании
  - ▶ Учитываются приоритеты потоков

# Планировщик в Windows

- ▶ 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

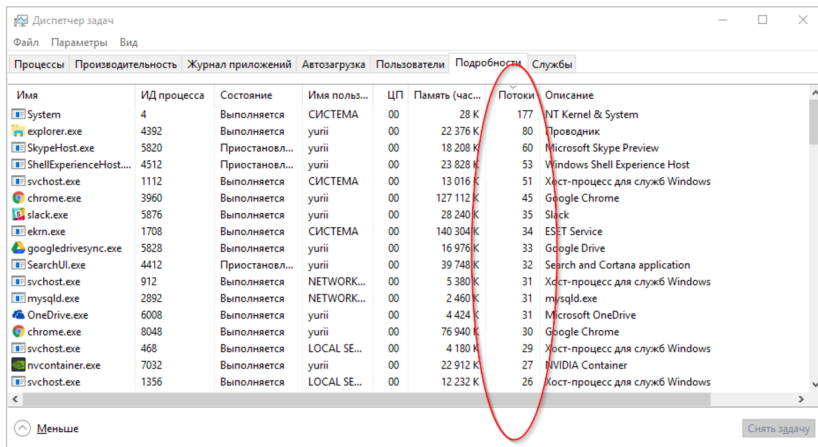
# Поток в Windows

- ▶ 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

```
using System;
using System.Threading;

namespace MultiThreadingDemo {
    class Program {
        static void Main(string[] args) {
            var otherThread = new Thread(() => {
                while (true)
                    Console.WriteLine("Hello from other thread!");
            });
            otherThread.Start();

            while (true)
                Console.WriteLine("Hello from this thread!");
        }
    }
}
```

# Параллельная обработка данных

```

static void Main(string[] args) {
    var array = new int[] { 1, 5, 2, 4, 7, 2, 4, 9, 3, 6, 5 };
    var threads = new Thread[3];
    var chunkSize = array.Length / threads.Length + 1;
    var results = new int[threads.Length];

    for (var i = 0; i < threads.Length; ++i) {
        var localI = i;
        threads[i] = new Thread(() => {
            for (var j = localI * chunkSize; j < (localI + 1) * chunkSize && j < array.Length; ++j)
                results[localI] += array[j];
        });
    }

    foreach (var thread in threads)
        thread.Start();
    foreach (var thread in threads)
        thread.Join();

    var result = 0;
    foreach (var subResult in results)
        result += subResult;

    Console.WriteLine($"Result = {result}");
}

```

# “Упрощённая” версия

```
static void Main(string[] args) {
    var array = new int[] { 1, 5, 2, 4, 7, 2, 4, 9, 3, 6, 5 };
    var threads = new Thread[3];
    var chunkSize = array.Length / threads.Length + 1;
    var result = 0;

    for (var i = 0; i < threads.Length; ++i) {
        var locall = i;
        threads[i] = new Thread(() => {
            for (var j = locall * chunkSize; j < (locall + 1) * chunkSize && j < array.Length; ++j)
                result += array[j];
        });
    }

    foreach (var thread in threads)
        thread.Start();
    foreach (var thread in threads)
        thread.Join();

    Console.WriteLine($"Result = {result}");
}
```

# Немного увеличим размер задачи...

```
static void Main(string[] args) {  
    var array = new int[1000];  
    for (var i = 0; i < array.Length; ++i)  
        array[i] = 1;  
  
    var threads = new Thread[8];  
    var chunkSize = array.Length / threads.Length + 1;  
    var result = 0;  
  
    for (var i = 0; i < threads.Length; ++i) {  
        var locall = i;  
        threads[i] = new Thread(() => {  
            for (var j = locall * chunkSize; j < (locall + 1) * chunkSize && j < array.Length; ++j)  
                result += array[j];  
        });  
    }  
  
    foreach (var thread in threads)  
        thread.Start();  
    foreach (var thread in threads)  
        thread.Join();  
  
    Console.WriteLine($"Result = {result}");  
}
```



# Почему так

```
result += array[j];
```



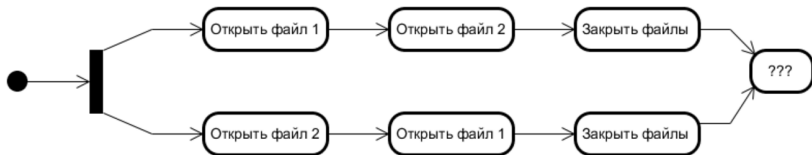
```
IL_0016: ldarg.0    // this
IL_0017: ldffd     class Program/'<>c__DisplayClass0_0' Program/'<>c__DisplayClass0_1'::'CS$<>8_locals1'
IL_001c: ldarg.0    // this
IL_001d: ldffd     class Program/'<>c__DisplayClass0_0' Program/'<>c__DisplayClass0_1'::'CS$<>8_locals1'
IL_0022: ldffd     int32 Program/'<>c__DisplayClass0_0'::result
IL_0027: ldarg.0    // this
IL_0028: ldffd     class Program/'<>c__DisplayClass0_0' Program/'<>c__DisplayClass0_1'::'CS$<>8_locals1'
IL_002d: ldffd     int32[] Program/'<>c__DisplayClass0_0'::'array'
IL_0032: ldloc.0    // j
IL_0033: ldelem.i4
IL_0034: add
IL_0035: stfld     int32 Program/'<>c__DisplayClass0_0'::result
```

Между **любыми** инструкциями поток может быть прерван

# Race condition



# Deadlock



## Какие ещё ловушки бывают

- ▶ Процессор может переставлять местами инструкции
  - ▶ Результат исполнения гарантируется таким же, как оригинальный, но промежуточные результаты другим ядрам могут быть видны странные
- ▶ У ядер процессора есть кеш (у каждого свой)
  - ▶ На самом деле, обычно три уровня кеша: L1 и L2 для каждого ядра свой, L3 общий для всех ядер
  - ▶ Кеши синхронизируются, но есть буферы чтения и записи, они нет

# Примитивы синхронизации

- ▶ Лучше необходимости синхронизации вообще избегать
- ▶ Бывают:
  - ▶ User-mode — атомарные операции, реализующиеся на процессоре и не требующие участия планировщика
  - ▶ Kernel-mode — примитивы, управляющие тем, как поток обрабатывается планировщиком
    - ▶ Более тяжеловесные и медленные (до 1000 раз по сравнению с “без синхронизации вообще”)
    - ▶ Позволяют синхронизировать даже разные процессы

# Атомарные операции

- ▶ Чтения и записи следующих типов всегда атомарны: Boolean, Char, (S)Byte, (U)Int16, (U)Int32, (U)IntPtr, Single, ссылочные типы
- ▶ Для других типов (например, Int64) операции чтения и записи могут быть прерваны посередине!
- ▶ Volatile
  - ▶ Volatile.Write
  - ▶ Volatile.Read
  - ▶ Связано с понятием Memory Fence, требует синхронизации ядер
  - ▶ Есть ключевое слово volatile: **private** volatile **int** flag = 0;
  - ▶ Volatile.Write должен быть последней операцией записи, Volatile.Read — первой операцией чтения
- ▶ Про это подробнее ближе к концу семестра, но volatile потребуется в домашке

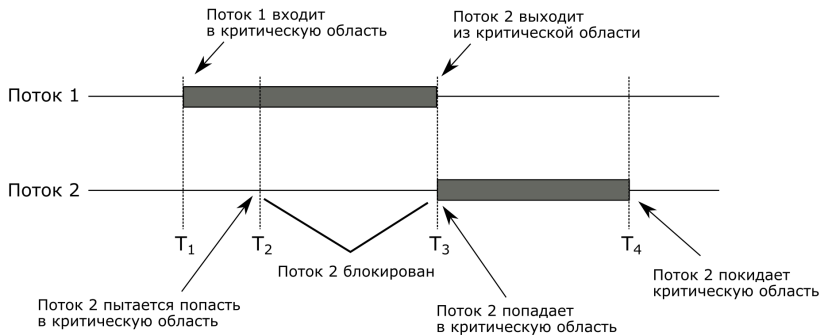
## Пример

```
private int flag = 0;  
private int value = 0;
```

```
public void Thread1() {  
    value = 5;  
    Volatile.Write(ref flag, 1);  
}
```

```
public void Thread2() {  
    if (Volatile.Read(ref flag) == 1)  
        Console.WriteLine(value);  
}
```

# Критические области





# Активное ожидание

```
private int turn = 0;
```

```
void Task1()
```

```
{
```

```
    while (true)
```

```
    {
```

```
        while (turn != 0) ;
```

```
        CriticalSection();
```

```
        turn = 1;
```

```
        NonCriticalSection();
```

```
    }
```

```
}
```

```
void Task2()
```

```
{
```

```
    while (true)
```

```
    {
```

```
        while (turn != 1) ;
```

```
        CriticalSection();
```

```
        turn = 0;
```

```
        NonCriticalSection();
```

```
    }
```

```
}
```

# Активное ожидание, обсуждение

- ▶ Не требует поддержки ОС
  - ▶ Поэтому переключение может быть очень быстрым
- ▶ Ждущий поток полностью занимает ядро
  - ▶ Греет процессор и очень быстро сажает аккумулятор
- ▶ Потоки работают строго по очереди
  - ▶ Это можно побороть, есть алгоритм Петерсона

# Проблема производителя и потребителя

## Producer-consumer problem

```
private Queue<int> buffer = new Queue<int>();
```

```
private void Producer() {  
    while (true) {  
        var item = ProduceItem();  
        if (buffer.Count == 100)  
            Sleep();  
        buffer.Enqueue(item);  
        if (buffer.Count == 1)  
            WakeUp(Consumer);  
    }  
}
```

```
private void Consumer() {  
    while (true) {  
        if (buffer.Count == 0)  
            Sleep();  
        var item = buffer.Dequeue();  
        if (buffer.Count == 100 - 1)  
            WakeUp(Producer);  
        ConsumeItem(item);  
    }  
}
```

# Семафоры

Дейкстры (того самого), 1965 год

- ▶ Целочисленный счётчик, который можно поднять и опустить (**up()** и **down()**)
- ▶ **down()** уменьшает счётчик на 1, если он больше нуля или блокирует вызывающего, если он 0
- ▶ **up()** увеличивает счётчик на один и, если он был нулём, будит одного из ожидающих потоков (случайного!)
- ▶ **down()** обычно делается при входе в критическую секцию, **up()** — при выходе
- ▶ Позволяет быть в критической секции не более чем заданному количеству потоков
  - ▶ Например, Google Drive не позволяет качать более чем с 10 подключениями одновременно, семафор решает проблему

# Производитель-потребитель на семафорах

```
private Queue<int> buffer = new Queue<int>();  
private Semaphore mutex = new Semaphore(0, 1);  
private Semaphore empty = new Semaphore(100, 100);  
private Semaphore full = new Semaphore(0, 100);
```

```
private void Producer()  
{  
    while (true)  
    {  
        var item = ProduceItem();  
        empty.WaitOne();  
        mutex.WaitOne();  
        buffer.Enqueue(item);  
        mutex.Release();  
        full.Release();  
    }  
}
```

```
private void Consumer()  
{  
    while (true)  
    {  
        full.WaitOne();  
        mutex.WaitOne();  
        var item = buffer.Dequeue();  
        mutex.Release();  
        empty.Release();  
        ConsumeItem(item);  
    }  
}
```

# Мьютекс

- ▶ Мьютекс — бинарный семафор
  - ▶ Пускает ровно один поток в критическую секцию
- ▶ Существенно проще в реализации и использовании, чем семафор
- ▶ Тоже требует поддержки операционной системы
  - ▶ Может использоваться для синхронизации даже процессов

# Производитель-потребитель на семафорах и мьютексе

```
private Queue<int> buffer = new Queue<int>();  
private Mutex mutex = new Mutex();  
private Semaphore empty = new Semaphore(100, 100);  
private Semaphore full = new Semaphore(0, 100);
```

```
private void Producer()  
{  
    while (true)  
    {  
        var item = ProduceItem();  
        empty.WaitOne();  
        mutex.WaitOne();  
        buffer.Enqueue(item);  
        mutex.ReleaseMutex();  
        full.Release();  
    }  
}
```

```
private void Consumer()  
{  
    while (true)  
    {  
        full.WaitOne();  
        mutex.WaitOne();  
        var item = buffer.Dequeue();  
        mutex.ReleaseMutex();  
        empty.Release();  
        ConsumeItem(item);  
    }  
}
```

# Монитор

Хоара, 1974 год

- ▶ Пользоваться семафорами очень сложно — например, поменять `empty.WaitOne()`; и `mutex.WaitOne()`; в `Producer()` — хороший способ устроить дедлок
  - ▶ Представим, что буфер полон. `Producer()` захватывает мьютекс и встаёт на семафоре `empty`, потому что он 0, управление передаётся `Consumer()`. Он тут же встаёт на `mutex.WaitOne()`, потому что он захвачен `Producer()`-ом. Теперь оба потока ждут друг друга.
- ▶ Поэтому придумали мониторы
- ▶ Монитор — набор методов (или функций), внутри которых может находиться ровно один поток
- ▶ Реализуется через мьютексы, требует поддержки в языке программирования



# Производитель-потребитель на мониторе

```
private class SynchronizedQueue {
    private Queue<int> buffer =
        new Queue<int>();

    public void Enqueue(int item) {
        lock (buffer) {
            while (buffer.Count == 100)
                Monitor.Wait(buffer);
            buffer.Enqueue(item);
            Monitor.Pulse(buffer);
        }
    }

    public int Dequeue() {
        lock (buffer) {
            while (buffer.Count == 0)
                Monitor.Wait(buffer);
            var result = buffer.Dequeue();
            Monitor.Pulse(buffer);
            return result;
        }
    }
}
```

```
private SynchronizedQueue buffer =
    new SynchronizedQueue();

private void Producer() {
    while (true) {
        var item = ProduceItem();
        buffer.Enqueue(item);
    }
}

private void Consumer() {
    while (true) {
        var item = buffer.Dequeue();
        ConsumeItem(item);
    }
}
```

## lock в .NET

- ▶ У каждого объекта (сылочного типа) есть скрытое поле, указывающее на структуру синхронизации
- ▶ lock использует именно её
  - ▶ То есть lock в одной критической секции, но на разные объекты — это разные мониторы
  - ▶ Но lock в разных секциях на один объект — один монитор
  - ▶ lock умеет обрабатывать исключения и отпускать замок
    - ▶ Предыдущие примеры с семафорами и мьютексами были неправильными — не учитывались исключения
- ▶ Хорошая практика — создавать объект специально для синхронизации, **lock(this)** писать нельзя!

```
private Object lockObject = new Object();
```

```
private void SomeMethod() {
    lock (lockObject) {
        ...
    }
}
```

# WaitHandle

- ▶ WaitHandle — всё, что можно ожидать
  - ▶ EventWaitHandle
    - ▶ AutoResetEvent — по сути, булевый флаг, поддерживаемый ОС
    - ▶ ManualResetEvent — тоже булевый флаг, но сбрасывается вручную
- ▶ Остальные примитивы синхронизации — наследники WaitHandle

## Пример (самодельный замок на Event-ax)

```
internal class SimpleWaitLock : IDisposable {  
    private readonly AutoResetEvent available;  
    public SimpleWaitLock() {  
        available = new AutoResetEvent(true);  
    }  
  
    public void Enter() {  
        available.WaitOne();  
    }  
  
    public void Leave() {  
        available.Set();  
    }  
  
    public void Dispose() { available.Dispose(); }  
}
```

# Литература

Эндрю Таненбаум, Х. Бос, Современные операционные системы, Питер, 2017. 1120 С.



Jeffrey Richter, CLR via C# (4th Edition), Microsoft Press, 2012. 894pp.

