

Введение, многопоточное программирование

Юрий Литвинов
yurii.litvinov@gmail.com

01.09.2020г

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#

Отчётность

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

Критерии оценивания

- ▶ ECTS (шкала оценивания от A до F)
 - ▶ Не идёт в диплом
- ▶ Баллы за домашние задачи (для всех разные), баллы за контрольные и даже работу в аудитории
- ▶ Общий балл за домашки: $MAX(0, (n/N - 0.6)) * 2.5 * 100$
- ▶ Общий балл за контрольные: $n/N * 100$
- ▶ Итоговая оценка: минимум из этих двух баллов
- ▶ Дедлайны по домашкам, -0.5 балла за каждую неделю после дедлайна
- ▶ -0.5 балла за каждую попытку сдачи начиная с 3-й
- ▶ Сгорает не более половины баллов
- ▶ Всё не так плохо, не надо набрать 100%

Учебная практика

- ▶ Отдельный зачёт
- ▶ Программная реализация достаточно большой и полезной задачи
- ▶ Пишется весь семестр
- ▶ В конце — отчёт на 5 листов и 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
- ▶ Гуглогруппа: <https://groups.google.com/forum/#!forum/spbsu-se-bachelors-2019-2023>

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

Насколько вообще можно распараллелить

- ▶ Распараллеливание может дать неожиданно низкий прирост производительности

- ▶ Закон Амдала:

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

- ▶ α — доля строго последовательных расчётов
 - ▶ $1 - \alpha$ — доля расчётов, которые можно идеально распараллелить
 - ▶ S_p — ускорение
- ▶ Если у вас есть 9 задач на 1 минуту и 1 задача на 2 минуты, на 10 процессорах ускорение будет всего в 5.5 раз!
 - ▶ 11 единиц работы, 10 из которых идеально параллельны, одна нет
- ▶ Добавлять ядра с какого-то момента бессмысленно

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)

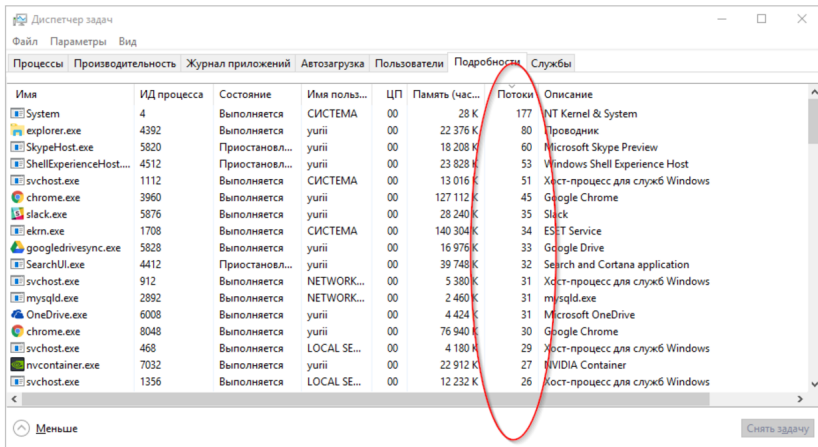
Ещё для каждой dll-ки, загруженной для процесса при старте или остановке потока, вызывается DllMain с параметрами `DLL_THREAD_ATTACH` и `DLL_THREAD_DETACH`

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

Две точки зрения на потоки

- ▶ Поток как абстракция параллельного вычисления — поток запускается, принимая функцию, которую он должен исполнять
 - ▶ Долгие вычисления, выполняющиеся независимо от остальных
 - ▶ Слежение за состоянием устройства
 - ▶ Индикация прогресса
- ▶ Поток как абстракция вычислителя — поток запускается и готов в бесконечном цикле принимать задачи
 - ▶ Куча коротких вычислений
 - ▶ Потому что запуск потока дорог
 - ▶ И нет смысла иметь активных потоков больше, чем ядер процессора
 - ▶ Реактивные системы, сетевые соединения и т.д.

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 localI = i;  
        threads[i] = new Thread(() => {  
            for (var j = localI * chunkSize; j < (localI + 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: ldfld      class Program/'<>c__DisplayClass0_0' Program/'<>c__DisplayClass0_1'::'CS$<>8_locals1'
IL_001c: ldarg.0    // this
IL_001d: ldfld      class Program/'<>c__DisplayClass0_0' Program/'<>c__DisplayClass0_1'::'CS$<>8_locals1'
IL_0022: ldfld      int32 Program/'<>c__DisplayClass0_0'::result
IL_0027: ldarg.0    // this
IL_0028: ldfld      class Program/'<>c__DisplayClass0_0' Program/'<>c__DisplayClass0_1'::'CS$<>8_locals1'
IL_002d: ldfld      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

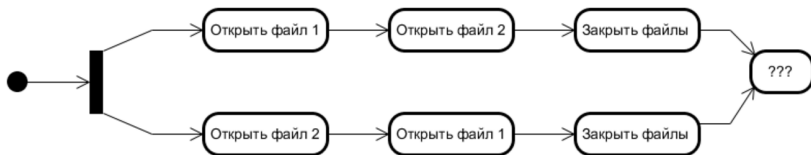
```

Между **любыми** инструкциями поток может быть прерван

Race condition



Deadlock



Условия взаимной блокировки

1. имеется разделяемый ресурс, к которому потоки хотят получить доступ, но пользоваться им может только один поток
2. таких ресурсов несколько и поток, захватив один, хочет получить доступ к другим, которые в этот момент захвачены другими потоками
3. нельзя отнять захваченный ресурс у потока
4. потоки ждут друг друга «по кругу»

Блокировка возможна, только если выполнены сразу все эти условия.

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