Здоровое || программирование

многопоточность и асинхронность

Обо мне

Родион Мостовой

Руководил разработкой в сервисе Доставка от Гудвина

Преподаю программирование, исследую технологии

Масштаб


User space
CancellationTokenSource
BlockingCollection
Data Race
Memory model
TPL
TaskCreationOptions.LongRunnin
ConfigureAwait(false)
AutoResetEvent
Race Condition
Parallel
SemaphoreSlim
PLINQ
Memory Ordering
Dispatcher.Invoke
Thread.Yield
DEADLOCK
CAS
CPU-bound
IO-bound Thread
ValueTask
ExecutionContext
Mutex
SpinLock
STORE, LOAD
Kernel space
Interlocked
Task.Yield
Parallel Stacks
CancellationToken
LOAD, LOAD
IMMUTABLE
Semaphore
ConcurrentDictionary
ReaderWriterLock
ThreadLocal
ThreadStatic
Thread Local Storage
AsyncEnumerable
Acquire semantics
TaskScheduler
CISC
ThreadPool starvation
IAsyncDisposable
Release semantics
Task
LOAD, STORE
ManualResetEvent
TaskCompletionSource
Thread starvation
ThreadPool
volatile
Actor
Monitor
Barrier
RISC
System.Threading.Channels
Callback
ConcurrentQueue
MemoryBarrier
ConcurrentBag
sync over async
AsyncLocal
CountDownEvent
Completion Port
SynchronizationContext
lock
async/await
ReaderWriterLockSlim
Dataflow
ConcurrentStack
Context Switch
STORE, STORE
Thread.Sleep(0)

О чем пойдет речь?

- Проверим, действительно ли таски быстрее потоков
- Асинхронность vs Многопоточность
- ThreadPool
- ThreadPool starvation
- Смысл в асинхронности в ASP.NET Core
- + Бонус

Поток

- Наименьшая единица исполнения задач в ОС (не считая файберов)
- Распределение работы потоков регулируется ОС (планировщиком потоков)
- Каждый поток исполняется на процессоре определенный **квант времени**

Поток

- Реальное кол-во одновременно выполняемых потоков ограничено кол-вом ядер процессора(ов)
- Каждый поток имеет стек и много чего еще (потребляет ресурсы ПК)
- Создание потока требует перехода в kernel space – это дорого

Что принадлежит потоку?

- Текущий режим доступа (пользовательский режим или режим ядра)
- **Контекст выполнения**, включающий значения регистров процессора и состояние выполнения
- **Один или два стека**, используемые для выделения памяти локальных переменных и управления вызовами
- Базовый приоритет и текущий (динамический) приоритет
- Привязка к процессору, указывающая, на каких процессорах разрешено выполнение потока
- **Массив локальной памяти потоков** (TLS, Thread Local Storage)
- и другое

Многопоточность

- **Многопоточность** – форма *конкурентности*, использующая несколько программных потоков выполнения.
- **Конкурентность** – выполнение сразу нескольких действий в одно и то же время.

Асинхронность

- **Асинхронное программирование –** разновидность конкурентности, использующая промисы или обратные вызовы **для предотвращения** создания лишних потоков
- **Конкурентность –** выполнение сразу нескольких действий в одно и то же время

Асинхронность

- Синхронный вызов: когда мы вызываем метод, то мы получим управления только после завершения всей операции
- Асинхронный: начинает операцию, сразу возвращает управления, и каким-то образом потом сообщит нам о завершении операции
- Асинхронность **не требует** создания нового потока

Зачем нужна асинхронность?

3 причины:

1. Большинство UI фреймворков работают на одном потоке, а блокировка этого потока равнозначна блокировке всего UI
2. Каждый поток (даже ожидающий) потребляет ресурсы ОС
3. Предотвращает ThreadPool starvation (об этом позже)

Многопоточность vs Асинхронность

- Многопоточность используется для CPU-bound операций
  - Тяжелые вычисления
  - Рендеринг
- Асинхронность используется в основном для IO-bound операций
  - Работа с сетью, в т. ч. с БД
  - Работа с файловой системой
  - There Is No Thread (Stephen Cleary)

Как работает честная асинхронность?

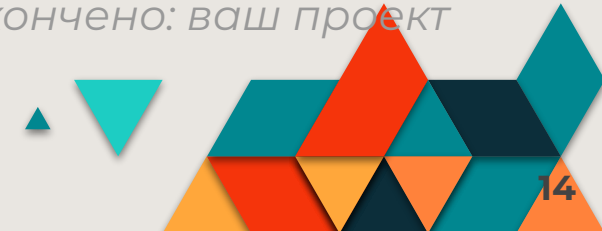
- Честные асинхронные вызовы работают без создания новых потоков
- В этом случае работа делегируется другим устройствам:
  - Сетевые запросы делегируются сетевой карте
  - Запросы к файловой системе делегируются жесткому диску

Класс Thread

- Класс `Thread` относится к низкоуровневым абстракциям, с его помощью создаются потоки “по старинке”
- Обычно потоки, созданные через `Thread` менее эффективны, чем потоки, созданные через высокоуровневые абстракции типа `Task`
- В современной разработке на C# почти не используется

Как только вы вводите команду new Thread(), все кончено: ваш проект уже содержит устаревший код.

© Stephen Cleary (“Конкурентность в C#”)



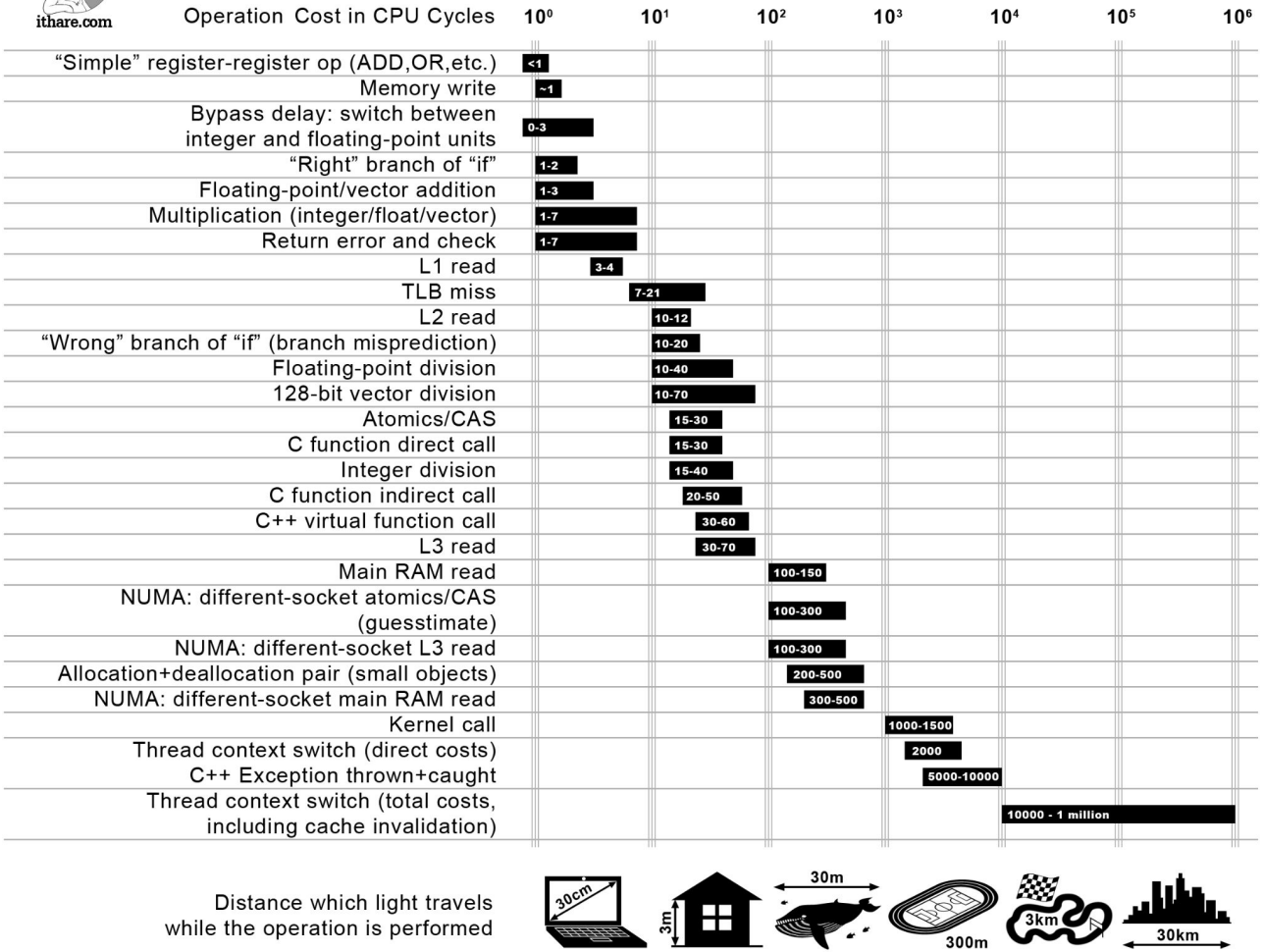
Класс Task

- Представляет доступ к созданию высокоуровневых эффективных потоков
- Имеет простой синтаксис, а также позволяет легко дождаться окончания выполнения (ключевое слово `await`)
- За создание новых потоков в этом случае отвечает пул потоков (`ThreadPool`)

ThreadPool (пул потоков)

- Когда код ставит работу в очередь пула потоков, то сам пул потоков в случае необходимости позаботится о создании потока
- Мы более не тратим время на создание потока ОС: мы работаем на уже созданных
- Даёт возможность работать на все 100% от всех процессорных ядер, либо наоборот ограничить пропускную способность
- Может **выполнять несколько делегатов за квант времени** (без переключение контекста)

Not all CPU operations are created equal



Сравним кол-во переключений

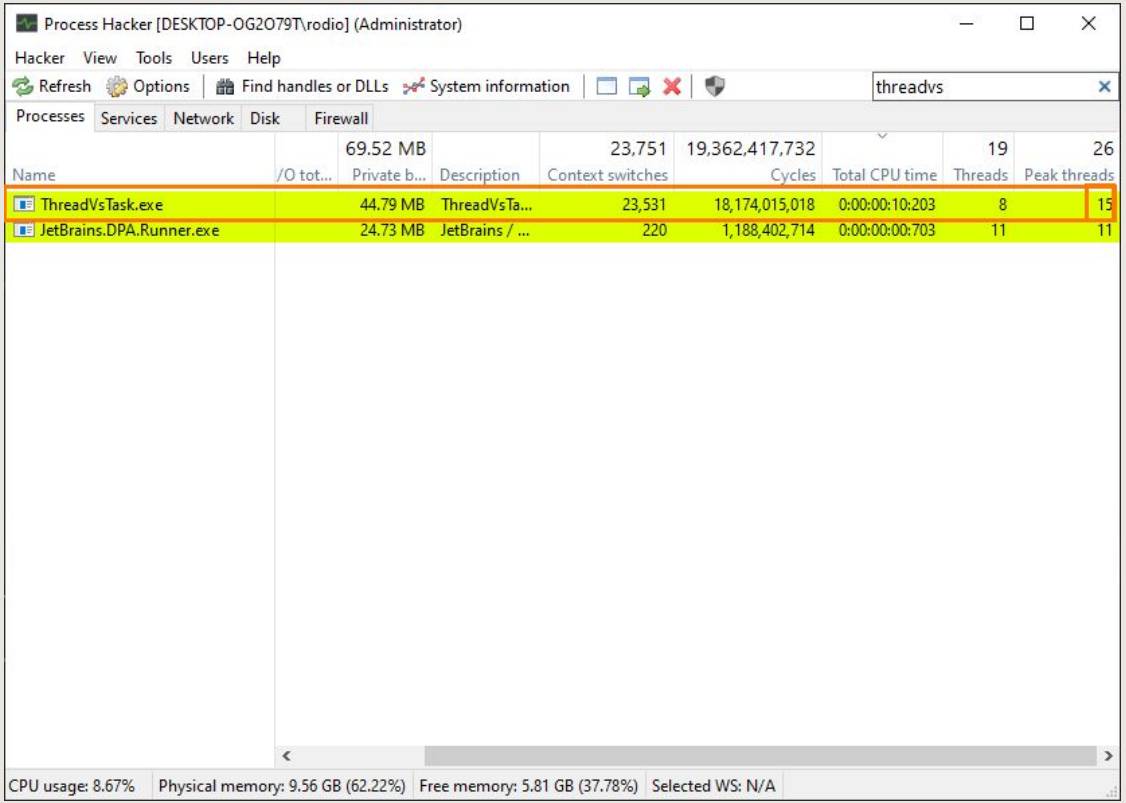
Можем ли мы посчитать кол-во переключений контекста?

- Да! Такие метрики предоставляет `Sysinternals Process Explorer` и `Process Hacker`
- Но честное общее кол-во переключений показывает только `Process Hacker v3`

Выполнение на нативных тредах

```
for (int i = 0; i < 10_000; i++)
{
    var t = new Thread(() =>
    {
        for (var dummy = 0; dummy < 1_000_000; dummy++) ;
    });
    t.Start();
}
```

Выполнение на нативных тредах

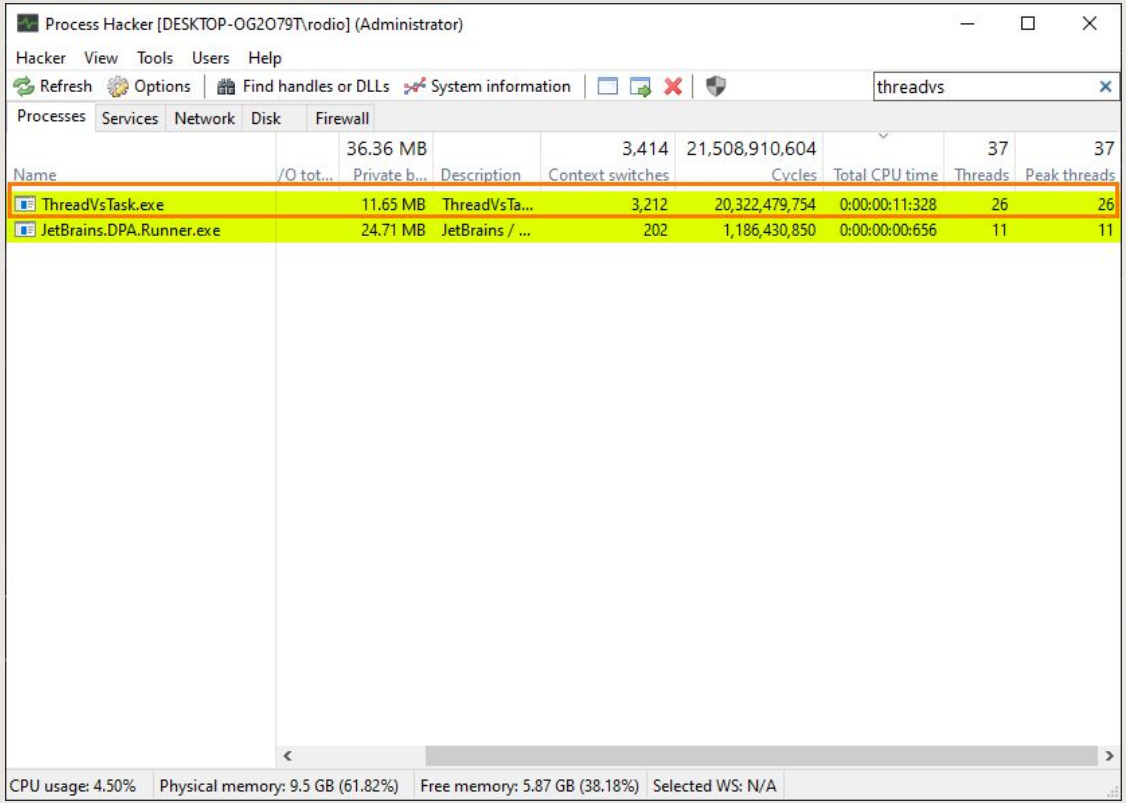


23k переключений
(время выполнения ~2000 мс)

Выполнение на тасках (ThreadPool)

```
for (int i = 0; i < 10_000; i++)
{
    Task.Run(() =>
    {
        for (var dummy = 0; dummy < 1_000_000; dummy++);
    });
}
```

Выполнение на тасках (ThreadPool)



3k переключений
(время выполнения ~800 мс)

Отлично, выбираем таски!

Неужели все так радужно и нет никаких проблем?

Есть! Точнее начнутся при вызове блокирующих вызовов:

- `Thread.Sleep(...)`
- `File.ReadAllBytes(...)`
- `dbContext.Users.Where(it ⇒ it.IsVip).ToList()`
- и множество других

Последствия: ThreadPool starvation

ThreadPool starvation (истощение пула потоков)

- Происходит, когда пулу потоков не хватает потоков для обработки поставленных в очередь задач (тасок)
- Возникает при использовании блокирующих вызовов в потоках `ThreadPool`'а
- При этом, ThreadPool отвечает увеличением числа потоков (**2** потока в секунду)
- Диагностика threadpool starvation при помощи утилиты counters

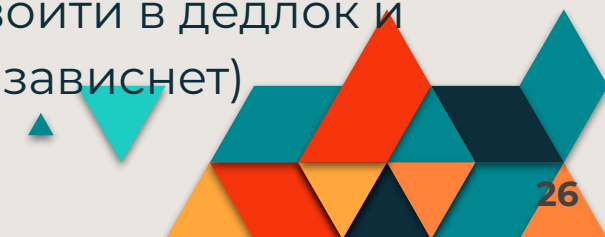
ThreadPool starvation

- Не использовать блокирующие методы
- В крайнем случае лечится увеличением `ThreadPool.SetMinThreads`

Sync over Async

`task.Wait(), task.Result, task.GetAwaiter().GetResult()`

- При вызове асинхронного метода как синхронного в лучшем случае вы лишитесь преимуществ асинхронности, т. к. вызывающий поток заблокируется на время вызова асинхронного метода
- **Также провоцирует ThreadPool starvation**
- В среде выполнения с контекстом синхронизации (WinForms, WPF, Unity, **xUnit**) ваша программа может войти в дедлок и заблокируется навсегда (попросту-говоря, зависнет)



DEMO
ThreadPool starvation

Асинхронность в ASP.NET Core

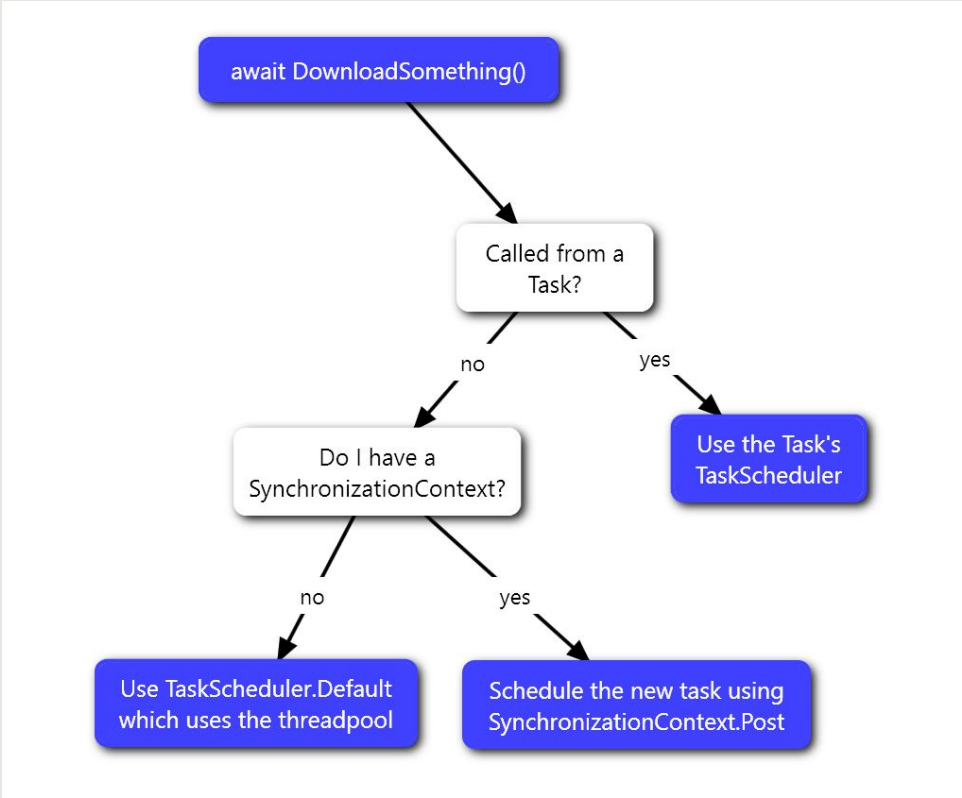
- Блокирующие вызовы блокируют и потоки
- А каждый поток потребляет ресурсы ОС
- Асинхронные же вызовы **отпускают** потоки, тем самым потребляя меньше ресурсов и предотвращают ThreadPool starvation
- В итоге сервер, на котором используются асинхронные методы сможем обработать больше запросов, чем сервер с блокирующими методами

ExecutionContext

CultureInfo.CurrentCulture
AsyncLocal<T>

Task.Run и Task.StartNew() всегда захватывает ExecutionContext
new Thread().Start() - захватывает ExecutionContext
new Thread().UnsafeStart() - Не захватывает ExecutionContext
ThreadPool.QueueUserWorkItem - захватывает ExecutionContext
ThreadPool.UnsafeQueueUserWorkItem - Не захватывает ExecutionContext

ConfigureAwait


await DownloadSomething()
Called from a Task?
no
yes
Do I have a SynchronizationContext?
Use the Task's TaskScheduler
no
yes
Use TaskScheduler.Default which uses the threadpool
Schedule the new task using SynchronizationContext.Post

Полезная информация

https://github.com/rodion-m/**learn_multithreading**

README.md

Материалы для изучения многопоточности и асинхронности

Курсы

- Семинары CLRium "Concurrency и Parallelism" (Стас Сидристый и Ко.)
- Курс "Параллельное программирование" 2022 (CS Center, Евгений Калишенко)
- Курс "Теория и практика многопоточной синхронизации (ТПМС, Concurrency)" [Лекции] 2022 (ФМПИ, Роман Липовский)
- Курс "Теория и практика многопоточной синхронизации (ТПМС, Concurrency)" [Семинары] 2022 (ФМПИ, Роман Липовский)
- Курс "Параллельные и распределённые вычисления" 2021 (ФПМИ, Ивченко О. Н.)
- [JS] Сборник лекций и докладов на тему "Асинхронное программирование" (Тимур Шемсединов)

Доклады

- Лекция "Многопоточное программирование в .NET ч. 1" (Дмитрий Иванов)
- Лекция "Многопоточное программирование в .NET ч. 2" (Дмитрий Иванов)

Полезная информация

Семинары CLRium “Concurrency и Parallelism” (Стас Сидристый)

Лекция “Многопоточное программирование в .NET ч. 1” , ч. 2 (Дмитрий Иванов)

Online-книга DotNetBook (Threads) (Стас Сидристый)

Курс “Параллельное программирование” 2022 (Евгений Калишенко)

Подкаст Подлодки про многопоточность (с Романом Елизаровым)

Sasha Goldshtein — The C++ and CLR Memory Models

Публичное собеседование по многопоточности (Kotlin)

Спасибо!

https://github.com/rodion-m/HealthyMultithreading_20220714

Канал: https://t.me/**selfmadeprog**

Мой TG: https://t.me/**rodion_m_tg**

Спасибо Марку Шевченко и Евгению Пешкову за помощь в подготовке