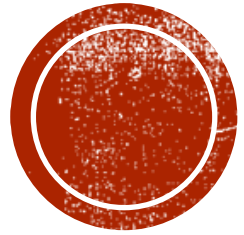


ЯЗЫКИ ПРИКЛАДНОГО ПРОГРАММИРОВАНИЯ

Лекция 10



SYSTEM.THREADING



КЛАСС THREAD

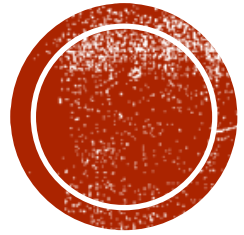
- `Thread` – класс представляющий собой поток.
- `Thread.CurrentThread` – текущий поток.
- `Thread.Sleep` – остановить поток на **X** мс.
- Поля:
 - `Priority` (`Lowest`, `BelowNormal`, `Normal`, `AboveNormal`, `Highest`) – приоритет потока
 - `IsBackground` – указывает, является ли поток фоновым
 - `ManagedThreadId` – id текущего потока
 - `Name` – имя потока
 - `IsAlive` – работает ли поток в текущий момент
 - `ThreadState` – состояние потока



ЗАПУСК ПОТОКА

- Thread(ThreadStart start)
- Thread(ThreadStart start, int maxStackSize)
- Thread(ParameterizedThreadStart start)
- Thread(ParameterizedThreadStart start, int maxStackSize)
- *ThreadStart, ParameterizedThreadStart – делегаты*





СИНХРОНИЗАЦИЯ ПОТОКОВ



ПРИМЕР БЕЗ СИНХРОНИЗАЦИИ

```
6 references
private static int _counter = 0;
4 references
private const int IterationsCount = 100000;

4 references
private static void RunThreadsForAction(ThreadStart start)
{
    _counter = 0;
    var workerOne = new Thread(start);
    var workerTwo = new Thread(start);

    workerOne.Start();
    workerTwo.Start();

    Thread.Sleep(1000);
    Console.WriteLine($"[{Environment.CurrentManagedThreadId}] Counter at the end: {_counter}");
}

1 reference
private static void ThreadAction()
{
    Console.WriteLine($"[{Environment.CurrentManagedThreadId}] Started");
    for (var i = 0; i < IterationsCount; i++)
    {
        _counter++;
    }
    Console.WriteLine($"[{Environment.CurrentManagedThreadId}] Finished");
}
```



INTERLOCKED

- Набор простых атомарных операций (реализованных на уровне процессора)
- Самый легковесный способ синхронизации.



INTERLOCKED

- **Decrement/Increment(ref int location)** – атомарные ++, -- для простых типов
- **Add(ref int location, int value)** – складывает два числа и заменяет первое на сумму
- **Exchange(ref int location, int value)** – присвоить значение в **location** из **value** и возвращает его атомарно.
- **CompareExchange (ref double location, double value, double comparand);**
 - **location** - Целевой объект, который нужно сравнить с объектом **comparand** и, возможно, заменить.
 - **value** - Значение, которым будет заменено целевое значение, если проверка покажет равенство.
 - **comparand** - Значение, которое сравнивается со значением в позиции **location1**.
- **CompareExchange<T> (ref T location1, T value, T comparand) where T : class;** - для атомарного сравнения по ссылке



INTERLOCKED ПРИМЕР

1 reference

```
private static void ThreadActionWithInterlocked()
{
    Console.WriteLine($"[{Environment.CurrentManagedThreadId}] Started");
    for (var i = 0; i < IterationsCount; i++)
    {
        Interlocked.Increment(ref _counter);
    }
    Console.WriteLine($"[{Environment.CurrentManagedThreadId}] Finished");
}
```



МОНИТОР

- Прimitives реализующий идею критической секции т.е. взаимоисключающее выполнение части кода потоками
- В один момент времени – один поток имеет доступ.
- Наиболее важные методы:
 - `static void Enter (object obj)`
 - `static void Enter (object obj, ref bool lockTaken)`
 - `static bool TryEnter()`
 - `static void Exit (object obj)`



МОНИТОР, ПРИМЕР

2 references

```
private static readonly object MonitorObj = new object();
```

1 reference

```
private static void ThreadActionWithMonitor()
{
    Console.WriteLine($"[{Environment.CurrentManagedThreadId}] Started");
    for (var i = 0; i < IterationsCount; i++)
    {
        try
        {
            Monitor.Enter(MonitorObj);
            _counter++;
        }
        finally
        {
            Monitor.Exit(MonitorObj);
        }
    }
    Console.WriteLine($"[{Environment.CurrentManagedThreadId}] Finished");
}
```



LOCK

- Оператор `lock` получает взаимоисключающую блокировку заданного объекта перед выполнением определенных операторов, а затем снимает блокировку.
- Синтаксический сахар над `Monitor`.
- Можно дважды брать `lock` на один объект – это безопасно.
- В `lock` нельзя использовать `await`

```
lock (x)
{
    // Your code...
}
```

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```



РЕКОМЕНДАЦИИ

- Не используйте один и тот же экземпляр объекта блокировки для разных общих ресурсов: это может привести к взаимоблокировке или состязанию при блокировке.
- Минимизировать количество времени блокировки
- Не использовать блокировку на используемых в коде переменных и т.п. Лучше всего создать отдельную переменную типа `object`



LOCK ПРИМЕР

1 reference

```
private static readonly object LockObj = new object();
```

1 reference

```
private static void ThreadActionWithLock()
{
    Console.WriteLine($"[{Environment.CurrentManagedThreadId}] Started");
    for (var i = 0; i < IterationsCount; i++)
    {
        lock (LockObj)
        {
            _counter++;
        }
    }
    Console.WriteLine($"[{Environment.CurrentManagedThreadId}] Finished");
}
```



ПРОЧИЕ ПРИМИТИВЫ

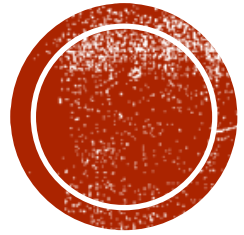
- **Semaphore** – примитив синхронизации основанный на счетчике, позволяет «запустить» в критическую секцию более одного потока
- **Mutex** – упрощенная реализация семафора.
- **ResetEvents** (Auto, Manual...)
- Изучать тут: <https://learn.microsoft.com/ru-ru/dotnet/standard/threading/overview-of-synchronization-primitives>



PARALLEL

- Позволяет выполнить параллельно итерации цикла:
 - `public static ParallelLoopResult For(int fromInclusive, int toExclusive, Action<int> body)`
 - `public static ParallelLoopResult ForEach<TSource>(IEnumerable<TSource> source, Action<TSource> body)`





SYSTEM.THREADING.TASKS



КЛАСС TASK

- **Task** – абстракция асинхронной операции, которая может быть запущена в отдельном потоке.
- **Task** – операция не возвращающая значений
- **Task<T>** - операция возвращающая значение типа **T**



СОЗДАНИЕ И ЗАПУСК ЗАДАЧИ

- **Task** можно создать несколькими способами:
 - Через конструктор передав делегат/лямбду
 - `static Task Run(Action action)` – создает и запускает задачу на выполнение
 - `Task.Factory.StartNew(Action action)`
 - ...
- Метод **Start()** – запускает задачу.
- Задачи запускаются в пуле потоков.

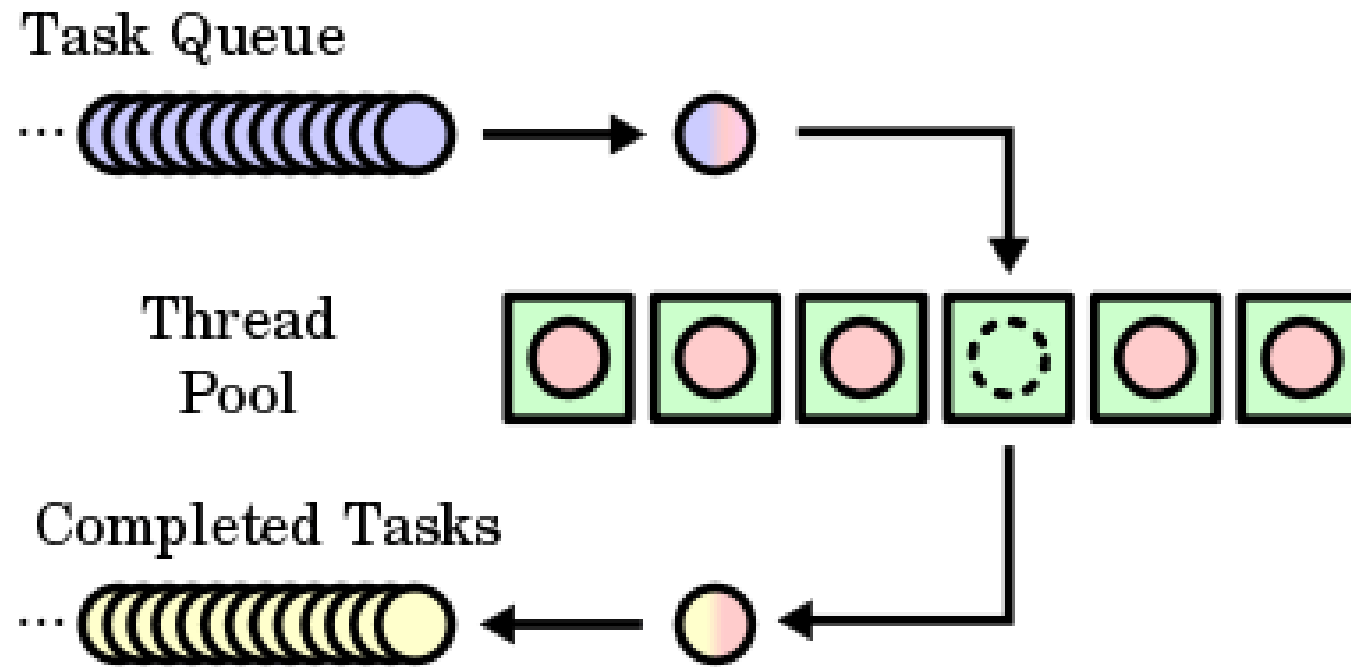


ПУЛ ПОТОКОВ

- Создание потока – не самая дешевая операция. Если необходимо выполнять много коротких задач – эффективнее создать потоки заранее и дальше распределять по ним задачи.
- В .NET для этих целей используется класс **ThreadPool**
 - эффективно управляет потоками, уменьшая количество создаваемых, запускаемых и останавливаемых потоков (по сравнению с ручным запуском) – учитывает количество ядер процессора и т.п. технические детали.
 - Позволяет сосредоточиться на решении задачи, а не на инфраструктуре потоков приложения.



ПУЛ ПОТОКОВ



ПУЛ ПОТОКОВ

Ограничения:

- Все потоки пула – фоновые, их приоритеты менять нельзя
- Потоки в пуле подходят для выполнения коротких задач, длинные операции следует запускать в отдельном от пула потоке



ПРИМЕР ЗАПУСКА ЗАДАЧИ

```
1 reference
public static void Run()
{
    var task = new Task(() =>
    {
        Console.WriteLine($"PID: {Environment.CurrentManagedThreadId} START");
        Task.Delay(1000);
        Console.WriteLine($"PID: {Environment.CurrentManagedThreadId} END");
    });

    task.Start();
    task.Wait();
    Console.WriteLine($"PID: {Environment.CurrentManagedThreadId} Main thread");
}
```



ВОЗВРАЩЕНИЕ РЕЗУЛЬТАТОВ ЗАДАЧИ

- Используем **Task<T>**, при создании соответственно вместо **Action – Func**
- Результат выполнения будет записан в свойство **Result** – при этом, если обратиться к нему то текущий поток останавливается и ждет, пока результат не будет получен.



ОСТАНОВКА ЗАДАЧ

- Для прерывания задач в .NET используется структура **CancellationToken**
- **CancellationTokenSource** – объект для создания и управления **CancellationToken**
- Для остановки – **CancellationTokenSource.Cancel()**;



ПРИМЕР ОСТАНОВКИ ЗАДАЧ

```
1 reference
public static void RunWithCancellationTokenSource()
{
    var source = new CancellationTokenSource();
    Task.Run(() =>
    {
        Console.WriteLine($"PID: {Environment.CurrentManagedThreadId} START");
        for (var i = 0; i < 100; i++)
        {
            Console.WriteLine($"PID: {Environment.CurrentManagedThreadId} {i}");
            Thread.Sleep(100);
        }
        Console.WriteLine($"PID: {Environment.CurrentManagedThreadId} END");
    }, source.Token);

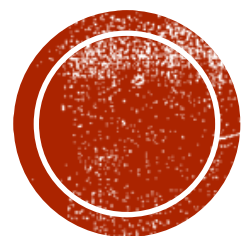
    Thread.Sleep(500);
    source.Cancel();
    Console.WriteLine($"PID: {Environment.CurrentManagedThreadId} Main thread");
}
```



ОЖИДАНИЕ ЗАДАЧ И СИНХРОНИЗАЦИЯ

- `Task.Wait()` – ожидает завершения задачи (есть варианты с `cancellationToken`, указанием времени и др.)
- `static void WaitAll(params Task[] tasks)` – ожидает завершения всех задач
- `static void WaitAny(params Task[] tasks)` – ожидает завершения хотя-бы одной задачи
- `static Task WhenAll(params Task[] tasks)` – создает задачу которая будет выполнена когда завершаться все переданные в нее.
- `WhenAny` – аналогично `WhenAll`





ASYNC / AWAIT

ASYNC

- Модификатор ***async*** позволяет указать, что метод, лямбда-выражение или анонимный метод является асинхронным. Если этот модификатор используется в методе или выражении, они называются асинхронными методами.
- ***Async*** позволяет использовать в методе ключевое слово ***await*** и указывает компилятору на необходимость создания конечного автомата для обеспечения асинхронной работы.
- Если метод, который изменяется ключевым словом ***async***, не содержит выражения или оператора ***await***, метод выполняется синхронно.



ASync

- Возвращаемые значения:
 - `void` – для обработчика событий.
 - `Task` – если нет возвращаемого значения.
 - `Task<TResult>` - если есть возвращаемое значение.
 - Любой тип, имеющий доступный **GetAwaiter** метод. Объект, возвращаемый методом **GetAwaiter**, должен реализовывать интерфейс **System.Runtime.CompilerServices.ICriticalNotifyCompletion** (`ValueTask` и `ValueTask<TResult>`)
 - `IAsyncEnumerable<T>` - асинхронный поток/коллекция, для повторяющихся асинхронных вызовов.

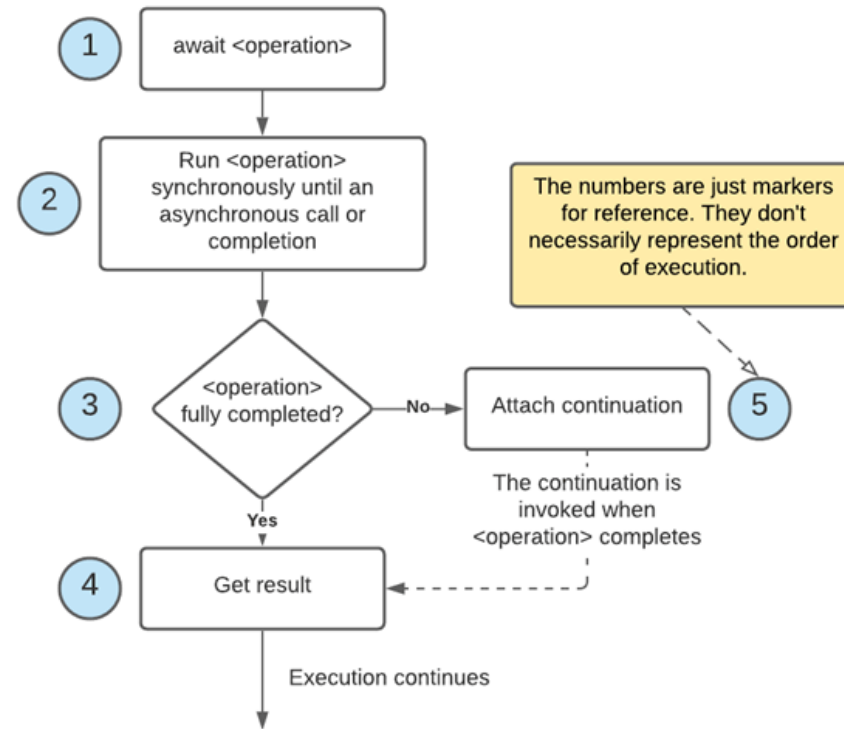


AWAIT

- **await** – унарный оператор, означающий, что необходимо дождаться выполнения асинхронной операции.
- В случае ожидания - вызывающий поток будет освобожден для последующих действий, а код метода после **await** будет выполнен в виде продолжения (**callback**).
- **Callback** в общем случае может быть выполнен уже другим потоком!



AWAIT



ASYNС/AWAIT ПРИМЕР

0 references

```
public static void Run()
{
    Console.WriteLine($"PID: {Environment.CurrentManagedThreadId} Main START");
    DoWorkAsync().Wait();
    Console.WriteLine($"PID: {Environment.CurrentManagedThreadId} Main END");
}
```

1 reference

```
private static async Task DoWorkAsync()
{
    Console.WriteLine($"PID: {Environment.CurrentManagedThreadId} START");

    await Task.Delay(1000);
    // real work with awaiting
    Console.WriteLine($"PID: {Environment.CurrentManagedThreadId} END");
}
```



РЕКОМЕНДАЦИИ

- Если метод асинхронный – добавьте в конце имени **Async**
- Если можно не использовать **async/await** (например – можно без ущерба для качества кода и обращиваний всего в лямбду вернуть **Task**) то стоит так делать.



ЧТО ЕЩЕ ИЗУЧИТЬ?

- <https://www.youtube.com/watch?v=eip2Xjrzp9g> – про конечный автомат, декомпиляция, разбор деталей.
- <https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/concepts/async/>
- <https://www.youtube.com/watch?v=zj7cveBMx5Q> – лекция 2020 про многопоточность, про Task, блокировки и некоторые практические примеры.
- <https://www.youtube.com/watch?v=TXau9UuMRSY> – лекция 2020 про async/await и Task, примеры с UI потоком и оптимизацией десктопных приложений (вынесение задач из UI потока).

