Лекция №6. Многопоточное программирование

Потоки, домены и процессы

Программирование в многопоточной среде позволяет эффективно использовать ресурсы процессора и повышает производительность приложений. В C# можно работать с потоками (threads), процессами (processes) и доменами приложений (application domains).

- **Процесс** — изолированная единица выполнения с собственной виртуальной памятью.
- **Поток (Thread)** — наименьшая единица выполнения внутри процесса. Потоки разделяют память процесса.
- **Домен приложения** — логический контейнер внутри процесса для изоляции кода и данных (используется редко в современных .NET).

Thread

Поток (`Thread`) представляет собой последовательность выполняемых инструкций. Потоки разделяют общую память, что позволяет им быстрее обмениваться данными, но также создаёт необходимость синхронизации доступа к общим ресурсам.

Потоки могут выполняться параллельно на многопроцессорных системах или переключаться между собой на одноядерных системах. В C# управление потоками осуществляется через пространство имен `System.Threading` . Рассмотрим пример создания и запуска нескольких потоков одновременно:

```
using System;
using System.Threading;

class Program
{
    static void Main()
    {
        // Создаём и запускаем два потока
        Thread thread1 = new Thread(PrintNumbers);
        Thread thread2 = new Thread(PrintNumbers);
        Thread thread3 = new Thread(PrintNumbers);

        // При запуске передаем передают идентификатор потока
        thread1.Start(1);
        thread2.Start(2);
```

```
            thread3.Start(3);

            // Ожидаем завершения потоков
            thread1.Join();
            thread2.Join();
            thread3.Join();

            Console.WriteLine("Основной поток закончил выполнение");
        }

        static void PrintNumbers(object threadId)
        {
            for (int i = 1; i <= 5; i++)
            {
                Console.WriteLine($"Thread {threadId}: {i}");
                Thread.Sleep(500); // Имитируем работу, задерживая поток
            }
        }
    }
```

Если мы запустим этот код, то заметим, что потоки будут выводить значения в случайном порядке. Это показывает, что потоки выполняются параллельно и независимо друг от друга.

```
Thread 2: 1
Thread 3: 1
Thread 1: 1
Thread 2: 2
Thread 1: 2
Thread 3: 2
Thread 2: 3
Thread 3: 3
Thread 1: 3
Thread 2: 4
Thread 3: 4
Thread 1: 4
Thread 2: 5
Thread 1: 5
Thread 3: 5
Основной поток закончил выполнение
```

Parallel

Для упрощения многопоточной работы можно использовать `Parallel`. Под капотом использует пул потоков. Например так можно сделать параллельное выполнение цикла `for`, что может быть полезно для повышения производительности.

```
using System;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Parallel.For(0, 10, i =>
        {
            Console.WriteLine($"Iteration {i}, thread:
{Thread.CurrentThread.ManagedThreadId}");
        });
    }
}
```

Вызываемое на каждом шаге цикла лямбда-выражение выполняется в отдельном потоке.

```
Iteration 2, thread: 7
Iteration 3, thread: 9
Iteration 7, thread: 13
Iteration 1, thread: 4
Iteration 4, thread: 11
Iteration 8, thread: 14
Iteration 6, thread: 12
Iteration 9, thread: 15
Iteration 5, thread: 10
Iteration 0, thread: 1
```

Общие ресурсы и lock

При работе с потоками могут возникать **гонки данных**. Это ситуация, при которых разные потоки пытаются одновременно изменить одни и те же данные. Рассмотрим пример, когда два потока одновременно изменяют одну и ту же переменную без синхронизации, что приводит к некорректным результатам.

```
using System;
using System.Threading;

class Program
{
    static int counter = 0;

    static void Increment()
    {
        for (int i = 0; i < 100000; i++)
        {
            counter++; // Несколько потоков изменяют одну переменную
```

```
        }
    }

    static void Main()
    {
        Thread thread1 = new Thread(Increment);
        Thread thread2 = new Thread(Increment);

        thread1.Start();
        thread2.Start();

        thread1.Join();
        thread2.Join();

        // Ожидаем 200000, но получим другое значение
        Console.WriteLine($"Counter value: {counter}");
    }
}
```

В этом примере два потока одновременно увеличивают `counter`. Ожидаемое значение — 200000, но из-за гонки данных результат может быть неожиданным. Вот результаты 3 запусков программы.

```
Counter value: 120922
Counter value: 131555
Counter value: 183418
```

В этом примере используется `lock (_lock)`, который предотвращает одновременный доступ к `counter`, обеспечивая правильный результат.

```
using System;
using System.Threading;

class Program
{
    static int counter = 0;
    static object _lock = new object();

    static void Increment()
    {
        for (int i = 0; i < 100000; i++)
        {
            lock (_lock)
            {
                counter++; // Доступ к переменной теперь безопасен
            }
        }
    }
```

```
    static void Main()
    {
        Thread thread1 = new Thread(Increment);
        Thread thread2 = new Thread(Increment);

        thread1.Start();
        thread2.Start();

        thread1.Join();
        thread2.Join();

        // Теперь всегда будет 200000
        Console.WriteLine($"Counter value: {counter}");
    }
}
```

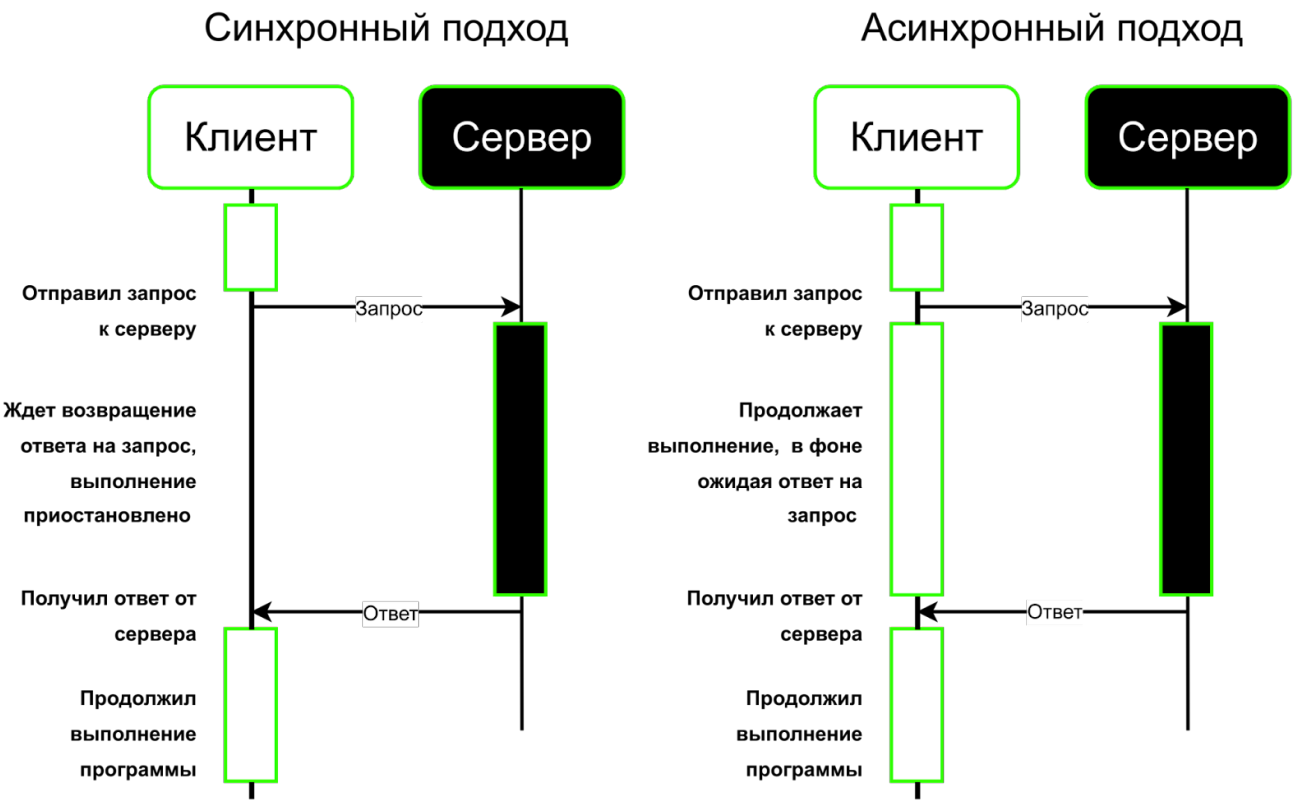
Асинхронность

Асинхронные методы используются для выполнения операций, которые могут занимать длительное время, например. Существует 2 основных сценария использования асинхронных операций — **I/O-bound и CPU-bound**.

I/O-bound — это операции ожидания внешних ресурсов такие как:

- Обращение к сетевым ресурсам (HTTP-запросы),
- Работа с файловой системой (чтение/запись файлов),
- Запросы к базе данных.

Рассмотрим пример в котором наше приложение взаимодействует с удаленным сервером. Если код синхронный, то при каждом запросе на сервер, основной поток нашей программы будет блокироваться. В случае графического приложения это будет выглядеть так, будто оно зависло. В случае использования асинхронного кода, ожидание ответа сервера не будет блокировать основной поток приложения. Асинхронность здесь достигается через **обратные вызовы** на уровне ОС, без блокировки потоков.



Для того того, чтобы метод был асинхронным перед ним необходимо указать ключевое слово `async`, также он должен возвращать один из следующих типов:

- `Task` — операция без возвращаемого значения
- `Task<T>` — операция, возвращающая `T`.

Для того чтобы получить возвращенное значение можно использовать `.Result` или `.Wait()`, которые будут ожидать значение, но нужно учитывать, что они **блокируют поток**, что сводит на нет все преимущества асинхронного программирования. Более предпочтительный способ это использование `await`, который **не блокирует поток**.

При работе с асинхронными методами нужно учитывать ряд нюансов. Давайте рассмотрим пример неправильного использования асинхронности.

```
using System;
using System.Threading.Tasks;

static async Task Main()
{
    var start = Stopwatch.StartNew();
    var result1 = await GetResultAsync(1); // Ожидаем завершения
    var result2 = await GetResultAsync(2); // Затем стартуем следующую
задачу
    var result3 = await GetResultAsync(3);
    var result4 = await GetResultAsync(4);
    Console.WriteLine($"Результаты: {result1}, {result2}, {result3},
```

```
{result4}");
    Console.WriteLine($"Время выполнения: {start.ElapsedMilliseconds} мс");
// ~4000 мс
}

static async Task<int> GetResultAsync(int num)
{
    await Task.Delay(1000); // Имитация долгой операции
    return num;
}
```

Каждый `await` останавливает выполнение до завершения текущего асинхронного метода.
Итоговое время выполнения — ~4 секунды, хотя операции не зависят друг от друга.

Правильный подход заключается в том, чтобы запустить все задачи одновременно и дождаться их завершения.

```
using System;
using System.Threading.Tasks;

static async Task Main()
{
    var start = Stopwatch.StartNew();
    var task1 = GetResultAsync(1); // Запускаем задачу
    var task2 = GetResultAsync(2); // Запускаем следующую сразу, без
ожидания
    var task3 = GetResultAsync(3);
    var task4 = GetResultAsync(4);

    // Ожидаем завершения всех задач
    var result1 = await task1;
    var result2 = await task2;
    var result3 = await task3;
    var result4 = await task4;

    Console.WriteLine($"Результаты: {result1}, {result2}, {result3},
{result4}");
    Console.WriteLine($"Время выполнения: {start.ElapsedMilliseconds} мс");
// ~1000 мс
}
```

Так как `await` для каждой задачи вызывается только после их запуска, время выполнения кода составило ~1 секунду.

Этот код так же можно переписать более компактно с помощью `Task.WhenAll`.

```
static async Task Main()
{
    var start = Stopwatch.StartNew();
    var task1 = GetResultAsync(1);
    var task2 = GetResultAsync(2);
    var task3 = GetResultAsync(3);
    var task4 = GetResultAsync(4);

    int[] results = await Task.WhenAll(task1, task2, task3, task4);

    Console.WriteLine($"Результаты: {string.Join(", ", results)}");
    Console.WriteLine($"Время выполнения: {start.ElapsedMilliseconds} мс");
// ~1000 мс
}
```

CPU-bound — это вычисления, которые требуют интенсивной загрузки процессора. Для распараллеливания используются потоки (например, через `Task.Run`). Рассмотрим пример создания задачи, которая будет выполняться в отдельном потоке.

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task Main()
    {
        Console.WriteLine($"Начало работы в основном потоке (ID:
{Thread.CurrentThread.ManagedThreadId})");

        // Запускаем тяжелые вычисления в фоновом потоке
        Task<long> calculationTask = Task.Run(() =>
CalculateSum(1000000000));

        // Основной поток свободен и может выполнять другие задачи
        Console.WriteLine("Основной поток не блокируется!");

        // Ожидаем результат асинхронно
        long result = await calculationTask;
        Console.WriteLine($"Результат: {result}");

        Console.WriteLine("Основной поток завершает работу.");
    }

    static long CalculateSum(long max)
    {
        Console.WriteLine($"Вычисление начато в потоке:
{Thread.CurrentThread.ManagedThreadId}");
        long sum = 0;
```

```
        for (long i = 0; i < max; i++) sum += i;
        return sum;
    }
}
```

В консоли получим такой результат.

```
Начало работы в основном потоке (ID: 1)
Основной поток не блокируется!
Вычисление начато в потоке:  7
Результат: 4999999995000000000
Основной поток завершает работу.
```