Асинхронность и многопоточность

В современных программных системах всё чаще требуется одновременно выполнять несколько задач: обрабатывать запросы пользователей, работать с сетью, выполнять вычисления, обновлять интерфейс, взаимодействовать с внешними сервисами. Последовательное, однопоточное выполнение кода в таких условиях становится узким местом — программа либо работает медленно, либо перестаёт быть отзывчивой.

Многопоточность и асинхронность — это два ключевых подхода, которые позволяют эффективно использовать ресурсы компьютера и строить масштабируемые, производительные приложения на платформе .NET.

Важно сразу разделить эти два понятия, которые часто путают:

- **Многопоточность** — это одновременное выполнение нескольких потоков выполнения кода, чаще всего с целью ускорения вычислений или параллельной обработки данных.
- **Асинхронность** — это способ организации кода, при котором поток не блокируется во время ожидания результата операции (чаще всего операций ввода-вывода), а может быть использован для выполнения других задач.

1. Процессы, домены и потоки

Прежде чем переходить к практическим инструментам многопоточности в C#, необходимо разобраться в базовых понятиях, на которых строится выполнение программ: **процесс**, **домен приложения** и **поток**. Эти уровни определяют, как код запускается, изолируется и выполняется в операционной системе и в среде .NET.

Процесс

Процесс — это запущенный экземпляр программы в операционной системе.

Характеристики процесса:

- имеет собственное виртуальное адресное пространство памяти;
- изолирован от других процессов (один процесс не может напрямую обращаться к памяти другого);
- содержит как минимум один поток выполнения;
- управляется операционной системой.

Примеры процессов:

- запущенное консольное приложение;
- запущенный браузер (каждая вкладка может быть отдельным процессом);
- сервис Windows.

Если один процесс аварийно завершается, это не влияет напрямую на другие процессы. Такая изоляция повышает стабильность системы, но делает межпроцессное взаимодействие более сложным.

В .NET один процесс обычно соответствует одному запущенному приложению.

Домен приложения (Application Domain)

Домен приложения (AppDomain) — это логическая единица изоляции внутри процесса в среде .NET.

Основные идеи AppDomain:

- несколько доменов могут существовать в одном процессе;
- домены изолируют код и загруженные сборки друг от друга;
- каждый домен имеет собственный набор загруженных библиотек и статических данных.

Для базового понимания многопоточности достаточно знать, что потоки выполняются внутри домена приложения. В дальнейшем мы не будем напрямую работать с AppDomain, но этот термин может встречаться в документации.

Поток (Thread)

Поток — это минимальная единица выполнения кода.

Характеристики потока:

- выполняется внутри процесса;
- разделяет память процесса с другими потоками;
- имеет собственный стек вызовов;
- может выполняться параллельно с другими потоками (на многоядерных процессорах).

Минимально любой процесс содержит **один основной поток** (main thread), с которого начинается выполнение программы.

Когда в процессе создаётся несколько потоков, они **делят общие ресурсы** (память, статические поля, объекты в куче) и операционная система планирует их выполнение.

Это позволяет распараллелить выполнение некоторых задач, но при этом возникает риск конкурентного доступа к данным, то есть ситуации, когда несколько потоков пытаются получить/изменить одни и те же данные одновременно.

Каждый поток в .NET имеет:

- уникальный идентификатор (`ManagedThreadId`);
- состояние выполнения;
- приоритет (используется редко).

Простейший пример получения информации о текущем потоке:

```
using System;
using System.Threading;

class Program
{
    static void Main()
    {
        Console.WriteLine(
            $"Поток: {Thread.CurrentThread.ManagedThreadId}"
        );
    }
}
```

Этот код всегда выполняется в **основном потоке** приложения.

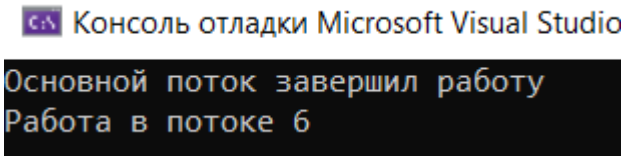
Позже мы увидим, как при создании дополнительных потоков этот идентификатор будет отличаться.

2. Класс Thread

Класс `Thread` из пространства имён `System.Threading` предоставляет **низкоуровневый механизм управления потоками** в C#. Он позволяет напрямую создавать, запускать и контролировать отдельные потоки выполнения. Понимание работы `Thread` важно для осознания того, как в принципе работает многопоточность, однако на практике этот класс используется всё реже из-за своей сложности и ограничений.

Для создания потока используется конструктор класса `Thread`, в который передаётся метод, выполняемый в новом потоке. Простейший пример:

```
class Program
{
    static void Main()
    {
        Thread thread = new Thread(DoWork);
        thread.Start();
        Console.WriteLine("Основной поток завершил работу");
    }
    static void DoWork() => Console.WriteLine($"Работа в потоке {Thread.CurrentThread.ManagedThreadId}");
}
```



В данном примере:

- основной поток продолжает выполнение после `Start()`;
- метод `DoWork` выполняется в отдельном потоке;
- порядок вывода сообщений **не гарантирован**.

Передача параметров в поток

Метод, выполняемый в потоке, может принимать **один параметр типа** `object`.

Пример:

```
static void Main()
{
    Thread thread = new Thread(PrintNumber);
    thread.Start(42);
}

static void PrintNumber(object value)
{
    Console.WriteLine($"Значение: {value}");
}
```

Недостатки такого подхода:

- отсутствие типобезопасности;
- необходимость приведения типов.

По этой причине `Thread` плохо подходит для сложных сценариев с передачей данных.

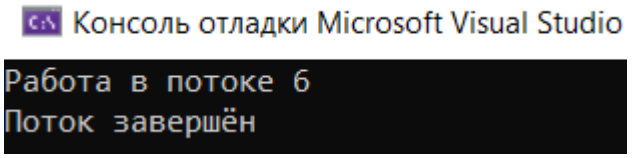
Ожидание завершения потока (Join)

Метод `Join()` блокирует текущий поток до завершения другого потока.

Пример:

```
Thread thread = new Thread(DoWork);
thread.Start();

thread.Join(); // ожидание завершения
Console.WriteLine("Поток завершён");
```



Важно понимать:

- `Join` **блокирует поток**, в котором был вызван;
- при неправильном использовании может снижать производительность и отзывчивость программы.

Фоновые и пользовательские потоки

Потоки бывают:

- **пользовательские (foreground)** — по умолчанию;
- **фоновые (background)** — завершаются автоматически при завершении процесса.

Настройка:

```
Thread thread = new Thread(DoWork);
thread.IsBackground = true;
thread.Start();
```

Особенности фоновых потоков:

- не гарантируют завершение работы;
- не подходят для сохранения данных и критических операций.

Управление временем выполнения

Временная приостановка потока:

```
Thread.Sleep(1000); // 1 секунда
```

Важно:

- `Sleep` блокирует поток полностью;
- не освобождает ресурсы;
- не является асинхронной операцией.

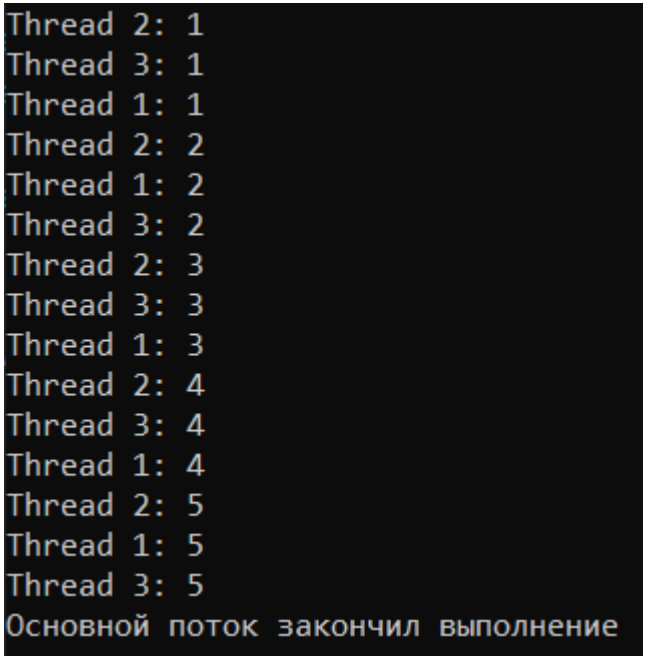
Использовать `Sleep` следует только для демонстраций или простых сценариев, но не в производственном коде.

Пример многопоточной работы

```
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.



Приоритеты потоков

Каждому потоку можно задать приоритет:

```
thread.Priority = ThreadPriority.Highest;
```

Однако:

- приоритеты являются **рекомендацией** для планировщика ОС;
- на практике редко дают ожидаемый эффект;
- неправильное использование может ухудшить работу приложения.

Ограничения и недостатки Thread

Основные проблемы при использовании `Thread`:

- высокая стоимость создания потоков;
- сложное управление жизненным циклом;
- отсутствие встроенной поддержки возврата результата;
- неудобная обработка исключений;
- плохая масштабируемость при большом количестве задач.

По этим причинам в современных приложениях `Thread` используется:

- для обучения и понимания основ;
- в редких низкоуровневых сценариях;
- в старом или специализированном коде.

3. ThreadPool и Parallel

Создание и управление потоками вручную с помощью `Thread` даёт полный контроль, но плохо масштабируется и требует значительных накладных расходов. В реальных приложениях чаще используются **высокоуровневые механизмы**, которые позволяют эффективно выполнять параллельный код без прямого управления потоками. К таким механизмам относятся **пул потоков (ThreadPool)** и библиотека **Parallel**.

ThreadPool — это набор заранее созданных и повторно используемых потоков, управляемых средой выполнения .NET.

Основная идея:

- потоки создаются заранее;
- задачи помещаются в очередь;
- свободный поток из пула берёт задачу и выполняет её;
- после завершения поток возвращается в пул и используется повторно.

Преимущества:

- значительно меньшие накладные расходы по сравнению с `new Thread`;
- автоматическое управление количеством потоков;
- хорошая масштабируемость под количество ядер процессора.

Ограничения:

- нельзя управлять жизненным циклом конкретного потока;
- нельзя задать приоритет или сделать поток фоновым;
- поток из пула не должен долго блокироваться.

Использование ThreadPool напрямую

Простейший способ поставить задачу в пул потоков:

```
using System;
using System.Threading;

class Program
{
    static void Main()
    {
        ThreadPool.QueueUserWorkItem(DoWork);
        Console.WriteLine("Задача отправлена в пул потоков");
    }

    static void DoWork(object state)
    {
        Console.WriteLine(
            $"Работа в потоке {Thread.CurrentThread.ManagedThreadId}"
        );
    }
}
```

Здесь:

- метод `DoWork` выполняется в одном из потоков пула;
- поток выбирается автоматически;
- основной поток не знает, когда именно завершится выполнение.

На практике прямое использование `ThreadPool` встречается редко, так как существуют более удобные абстракции (`Task`).

Библиотека Parallel

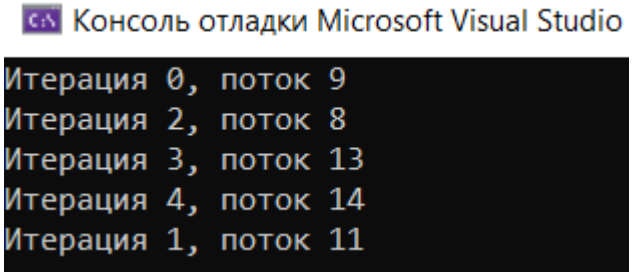
Класс `Parallel` (пространство имён `System.Threading.Tasks`) предназначен для **параллельного выполнения однотипных операций**.

Основные методы:

- `Parallel.For`
- `Parallel.ForEach`
- `Parallel.Invoke`

Пример `Parallel.For`:

```
Parallel.For(0, 5, i =>
{
    Console.WriteLine($"Итерация {i}, поток {Thread.CurrentThread.ManagedThreadId}");
});
```



Особенности:

- количество используемых потоков определяется автоматически;
- итерации могут выполняться в любом порядке;
- используется пул потоков.

Parallel.ForEach и Parallel.Invoke

Пример `Parallel.ForEach`:

```
int[] numbers = { 1, 2, 3, 4, 5 };

Parallel.ForEach(numbers, number =>
{
    Console.WriteLine(number * number);
});
```

Пример `Parallel.Invoke`:

```
Parallel.Invoke(
    () => DoWork1(),
    () => DoWork2(),
    () => DoWork3()
);
```

Эти конструкции удобны, когда:

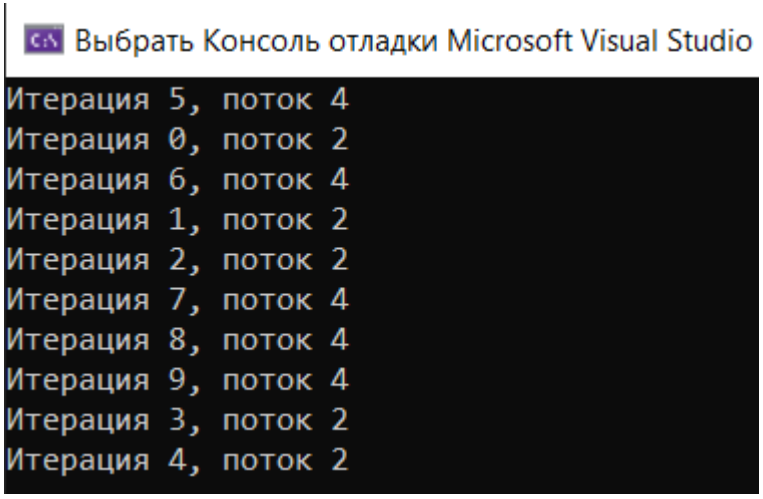
- есть набор независимых операций;
- нет необходимости управлять потоками вручную;
- операции не зависят друг от друга.

Управление параллелизмом

Для тонкой настройки используется `ParallelOptions`:

```
var options = new ParallelOptions
{
    MaxDegreeOfParallelism = 2
};

Parallel.For(0, 10, options, i =>
{
    Console.WriteLine($"Итерация {i}, поток {Thread.CurrentThread.ManagedThreadId}");
});
```



Это позволяет:

- ограничить количество одновременно выполняемых задач;
- избежать перегрузки процессора;
- учитывать ограничения внешних ресурсов.

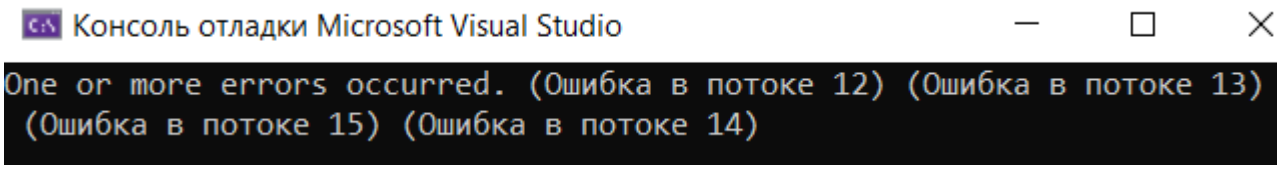
Исключения и Parallel

Если в одной из параллельных операций возникает исключение:

- выполнение может быть остановлено;
- исключения агрегируются в `AggregateException`.

Пример обработки:

```
try
{
    Parallel.For(0, 10, i =>
    {
        if (i > 5)
            throw new Exception($"Ошибка в потоке {Thread.CurrentThread.ManagedThreadId}");
    });
}
catch (AggregateException ex)
{
    Console.WriteLine(ex.Message);
}
```



4. Асинхронность

По мере усложнения приложений становится очевидно, что ручное управление потоками или даже использование `Parallel` подходит не для всех сценариев. Особенно это заметно при работе с операциями, которые **долго ждут результата**, но при этом не нагружают процессор. Для решения этих задач в .NET используется модель, основанная на **задачах (Task)** и ключевых словах `async` / `await`.

Что такое Task

`Task` — это высокоуровневая абстракция, представляющая асинхронную операцию, которая:

- может выполняться в фоне;
- может завершиться успешно, с ошибкой или быть отменённой;
- может возвращать результат (`Task<T>`).

Важно понимать: `Task` **не равен потоку**. Задача может выполняться в потоке из пула или не занимать поток вообще — например, при ожидании I/O-операции.

Типы задач:

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

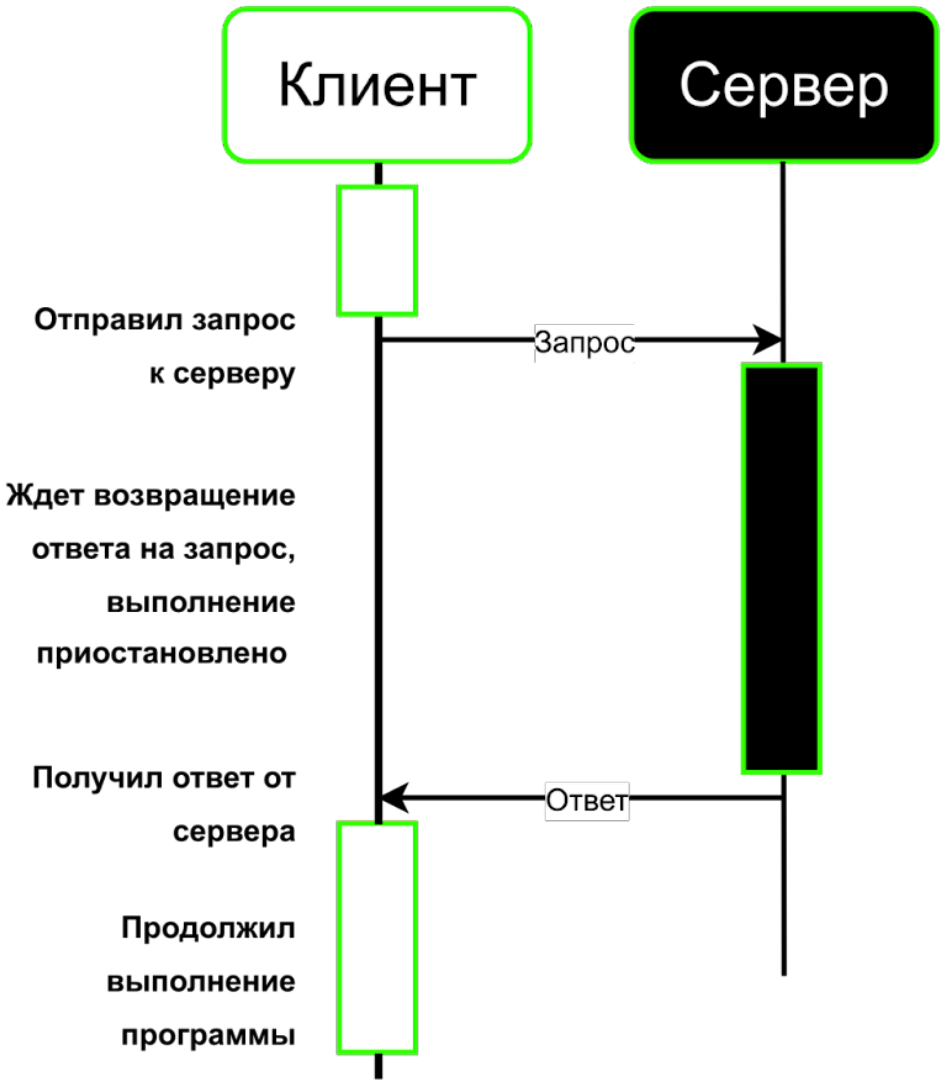
I/O-bound и CPU-bound

Существует 2 основных сценария использования асинхронных операций — **I/O-bound и CPU-bound**.

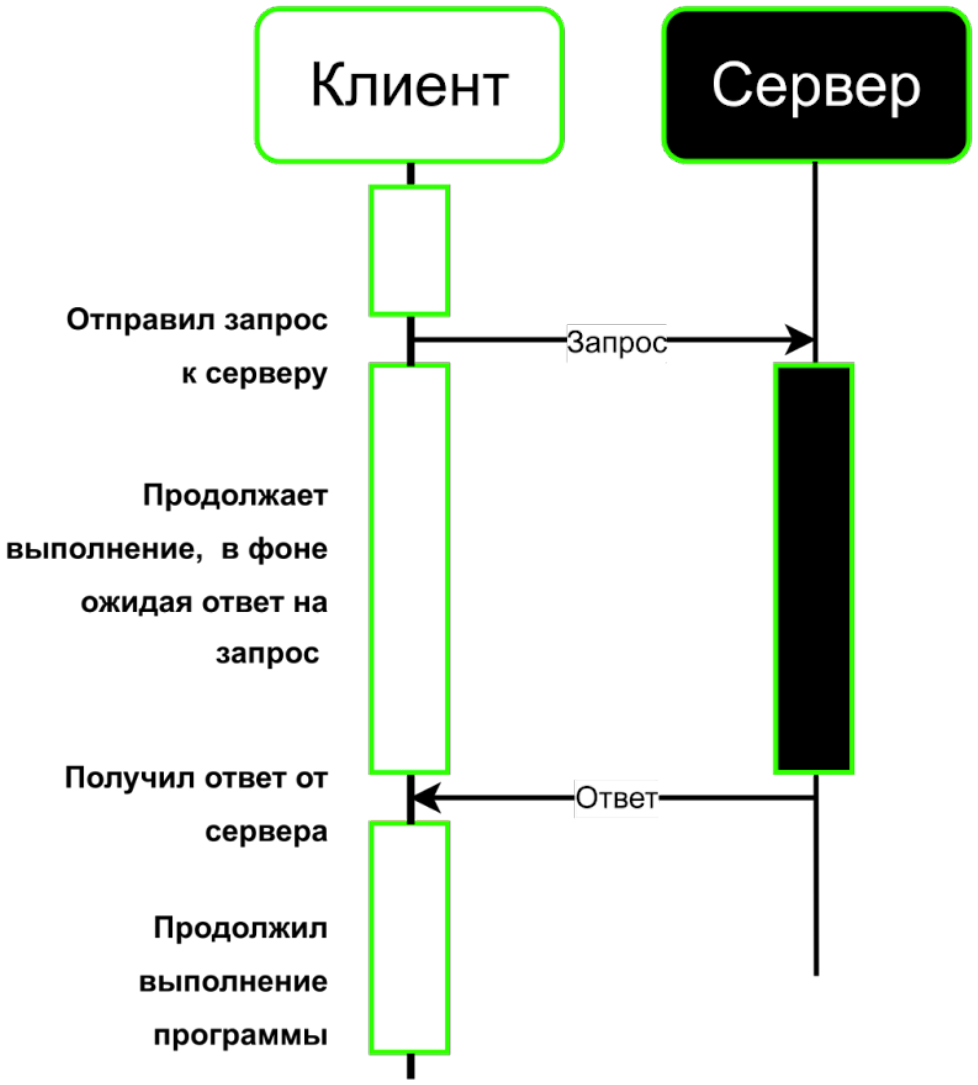
I/O-bound — операция, основной вклад в время выполнения даёт ожидание внешнего ресурса: сеть, диск, БД, таймеры. Процессор в эти моменты простаивает — он ждёт ответа. Примеры: HTTP-запросы, чтение/запись файлов, запросы к базе данных.

Рассмотрим пример, в котором приложение взаимодействует с удалённым сервером. Если код синхронный, при каждом запросе основной поток блокируется — в графическом приложении это выглядит как зависание. При использовании асинхронного кода ожидание ответа сервера не блокирует основной поток. Асинхронность здесь достигается через **обратные вызовы** на уровне ОС, без создания дополнительных потоков.

Синхронный подход



Асинхронный подход



Для I/O-bound операций метод объявляется с `async` и `await`, без создания новых потоков вручную:

```
async Task<string> GetDataAsync()
{
    HttpClient client = new HttpClient();
    string response = await client.GetStringAsync("https://example.com");
    return response;
}
```

CPU-bound — операция нагружает процессор: вычисления, обработка больших массивов данных, сложные алгоритмы. Время выполнения определяется скоростью CPU. Примеры: рендеринг изображений, шифрование, поиск простых чисел.

Для CPU-bound операций используется `Task.Run`, который явно отправляет работу в поток из пула:

```
static async Task Main()
{
    Console.WriteLine($"Основной поток: {Thread.CurrentThread.ManagedThreadId}");

    Task<long> calculationTask = Task.Run(() => CalculateSum(1_000_000_000));

    Console.WriteLine("Основной поток не блокируется!");

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

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

```
    return sum;
}
```

Оператор await

Оператор `await` приостанавливает выполнение **внутри того метода, где он объявлен** — то есть код внутри метода идёт последовательно, строка за строкой. Но сам метод при этом не блокирует вызывающий код: поток освобождается и может выполнять другие задачи.

```
async Task<int> GetResultAsync(int num) {
    await Task.Delay(1000); // выполнение приостанавливается здесь...
    return num;             // ...и продолжается здесь после завершения
}
```

Пока `GetResultAsync` ожидает `Task.Delay`, поток не заблокирован — он свободен для других задач. После завершения ожидания выполнение метода возобновляется.

Однако если вызвать `GetResultAsync` с `await` в другом методе, то и там выполнение приостановится до получения результата:

```
async Task Main() {
    var result = await GetResultAsync(1); // ждём здесь
    Console.WriteLine(result);            // выполнится только после получения result
}
```

Чтобы задача выполнялась асинхронно относительно вызывающего кода, нужно сначала запустить её **без** `await`, и только потом ожидать результат:

```
async Task Main() {
    var task = GetResultAsync(1); // запускаем, не ждём
    Console.WriteLine("Эта строка выполнится сразу");
    var result = await task;      // ждём результат здесь
    Console.WriteLine(result);
}
```

.Wait() и .Result

Для получения результата асинхронного метода также можно использовать `task.Wait()` и `task.Result`, однако они являются **блокирующими** — в отличие от `await` они удерживают поток до завершения задачи, что делает выполнение фактически синхронным:

```
async Task<int> GetResultAsync()
{
    await Task.Delay(1000);
    return 42;
}

// С await — поток свободен во время ожидания, код остаётся асинхронным:
int result = await GetResultAsync();

// С .Result — поток заблокирован до получения результата, как в синхронном коде:
int result = GetResultAsync().Result;
```

Помимо потери преимуществ асинхронности, `.Wait()` и `.Result` могут привести к **дедлокам** в некоторых типах приложений (например, в UI-приложениях). Поэтому предпочтительно всегда использовать `await`.

Параллельный запуск задач

При работе с несколькими независимыми асинхронными операциями важно не допускать их последовательного выполнения. Рассмотрим неправильный подход:

```
static async Task Main()
{
    var start = Stopwatch.StartNew();
    var result1 = await GetResultAsync(1); // ждём завершения...
    var result2 = await GetResultAsync(2); // ...затем стартуем следующую
    var result3 = await GetResultAsync(3);
    var result4 = await GetResultAsync(4);
    Console.WriteLine($"Результаты: {result1}, {result2}, {result3}, {result4}");
    Console.WriteLine($"Время: {start.ElapsedMilliseconds} мс"); // ~4000 мс
}

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

Каждый `await` останавливает выполнение до завершения текущей задачи, поэтому операции выполняются строго последовательно — ~4 секунды, хотя они независимы друг от друга.

Правильный подход — сначала запустить все задачи, затем ожидать их завершения:

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

    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: {start.ElapsedMilliseconds} мс"); // ~1000 мс
}
```

Более компактный способ — `Task.WhenAll`, который ожидает завершения всех задач сразу:

```
static async Task Main() {
    var start = Stopwatch.StartNew();
    int[] results = await Task.WhenAll(
        GetResultAsync(1),
        GetResultAsync(2),
        GetResultAsync(3),
        GetResultAsync(4)
    );
    Console.WriteLine($"Результаты: {string.Join(", ", results)}");
    Console.WriteLine($"Время: {start.ElapsedMilliseconds} мс"); // ~1000 мс
}
```

Исключения в асинхронном коде

Исключения в асинхронных методах сохраняются внутри `Task` и выбрасываются при `await`. Это делает обработку ошибок привычной и простой:

```
async Task ErrorAsync()
{
    await Task.Delay(100);
    throw new InvalidOperationException();
}

try
{
    await ErrorAsync();
}
catch (InvalidOperationException)
{
    Console.WriteLine("Ошибка обработана");
}
```

5. Синхронизация и общие ресурсы

При работе с несколькими потоками или задачами одной из самых сложных и критичных проблем становится **доступ к общим ресурсам**. Если несколько потоков одновременно читают и изменяют одни и те же данные без координации, программа может вести себя непредсказуемо. Этот раздел посвящён причинам таких проблем и основным механизмам синхронизации в C#.

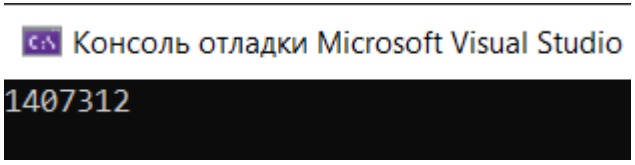
Общий ресурс — это любые данные или объекты, к которым могут одновременно обращаться несколько потоков: поля классов, статические переменные, элементы коллекций, файлы и соединения.

Гонка данных (race condition) возникает, когда несколько потоков одновременно работают с общими данными и результат зависит от порядка их выполнения, который никак не контролируется.

Простейший пример проблемы

```
int counter = 0;
Parallel.For(0, 5, i => Increment());
void Increment() {
    for (int i = 0; i < 1_000_000; i++)
        counter++;
}
Console.WriteLine(counter);
```

Если запустить `Increment` в нескольких потоках, итоговое значение `counter` почти всегда будет меньше ожидаемого. Причина в том, что операция `counter++` не атомарна — она состоит из трёх шагов: прочитать значение, увеличить, записать обратно. Когда несколько потоков выполняют эти шаги одновременно, часть обновлений теряется.



Ключевое слово lock

Основной и наиболее часто используемый механизм синхронизации в C# — ключевое слово `lock`. Оно гарантирует, что только **один поток** может находиться внутри защищённого участка кода одновременно.

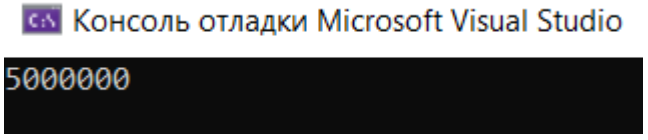
Участок кода внутри `lock` называется **критической секцией**. Чем она меньше, тем выше параллелизм — длительные операции внутри `lock` резко ухудшают производительность, поэтому блокировка должна защищать **данные**, а не логику целиком.

```
object _lockObj = new object();
int counter = 0;

Parallel.For(0, 5, i => Increment());

void Increment()
{
    for (int i = 0; i < 1_000_000; i++)
    {
        lock (_lockObj)
        {
            counter++;
        }
    }
}

Console.WriteLine(counter);
```



Deadlock (взаимная блокировка)

Deadlock — ситуация, когда два или более потоков навсегда ждут друг друга. Классический пример: первый поток захватил `lockA` и ждёт `lockB`, а второй поток захватил `lockB` и ждёт `lockA`. Оба потока заблокированы навсегда.

```
// Поток 1:
lock (lockA)
{
    lock (lockB)
    {
        /* работа */
    }
}

// Поток 2:
lock (lockB)
{
    lock (lockA)
    {
        /* работа */
    }
}
```

Чтобы избежать deadlock: всегда захватывайте блокировки в одном и том же порядке, минимизируйте количество вложенных `lock` и старайтесь сокращать время удержания блокировки.

Mutex

Mutex (mutual exclusion) — примитив синхронизации, который, как и `lock`, разрешает доступ к секции только одному потоку. Его принципиальное отличие в том, что Mutex работает **между процессами**, а не только внутри одного приложения. Это делает его полезным, например, чтобы запретить запуск второго экземпляра программы.

```
using var mutex = new Mutex(false, "MyAppMutex");

if (!mutex.WaitOne(0))
{
    Console.WriteLine("Приложение уже запущено.");
    return;
}

// Только один экземпляр приложения выполнит этот код
Console.WriteLine("Работаем...");
mutex.ReleaseMutex();
```

В рамках одного приложения для синхронизации потоков предпочтительнее использовать `lock` — он значительно легче и быстрее.

Semaphore и SemaphoreSlim

Semaphore — примитив синхронизации, который ограничивает количество потоков, одновременно входящих в защищённую секцию. В отличие от `lock`, который пропускает строго один поток, семафор позволяет задать любое допустимое число — например, не более 3 потоков одновременно.

В .NET существует два варианта: `Semaphore` (тяжёлый, работает между процессами) и `SemaphoreSlim` (лёгкий, только внутри процесса, поддерживает `async`). В большинстве случаев следует использовать `SemaphoreSlim`.

```
SemaphoreSlim semaphore = new SemaphoreSlim(3); // не более 3 потоков одновременно

var tasks = Enumerable.Range(1, 10).Select(async i =>
{
    await semaphore.WaitAsync(); // ждём разрешения войти
    try
    {
        Console.WriteLine($"Поток {i} начал работу");
        await Task.Delay(1000); // имитация работы
        Console.WriteLine($"Поток {i} завершил работу");
    }
    finally
    {
        semaphore.Release(); // освобождаем слот
    }
});

await Task.WhenAll(tasks);
```

В этом примере 10 задач запускаются одновременно, но в любой момент времени активно работают не более 3 из них — остальные ожидают освобождения слота.

Concurrent-коллекции

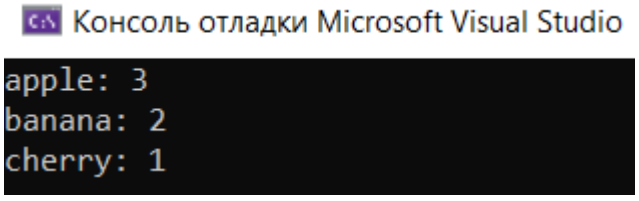
Альтернатива ручной синхронизации — потокобезопасные коллекции из пространства имён `System.Collections.Concurrent`. Они спроектированы для многопоточного доступа и внутри используют более эффективные механизмы, чем простой `lock` на всю коллекцию.

Основные варианты и их обычные аналоги:

Потокобезопасная	Обычная
`ConcurrentDictionary<K,V>`	`Dictionary<K,V>`
`ConcurrentQueue<T>`	`Queue<T>`
`ConcurrentStack<T>`	`Stack<T>`
`ConcurrentBag<T>`	`List<T>` (неупорядоченная)

Пример с `ConcurrentDictionary` — счётчик слов из нескольких потоков без явных блокировок:

```
var wordCount = new ConcurrentDictionary<string, int>();
string[] words = { "apple", "banana", "apple", "cherry", "banana", "apple" };
Parallel.ForEach(words, word => {
    wordCount.AddOrUpdate(word, 1, (key, count) => count + 1);
});
foreach (var (word, count) in wordCount)
    Console.WriteLine($"{word}: {count}");
```



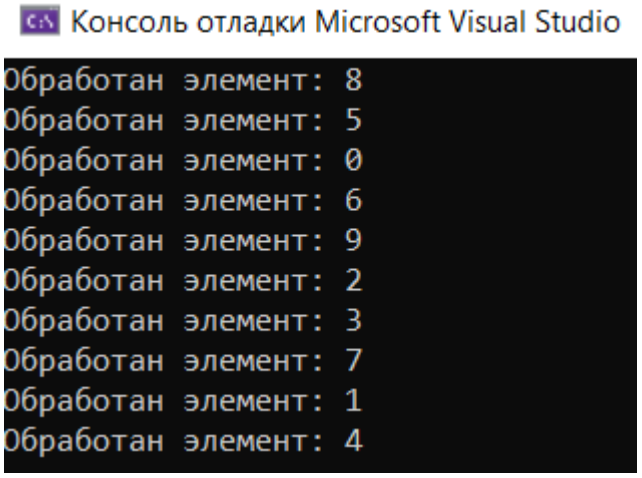
`ConcurrentQueue` удобен для сценариев «производитель — потребитель»: один поток добавляет задачи, другой их забирает:

```
var queue = new ConcurrentQueue<int>();

// Производитель
Parallel.For(0, 10, i => queue.Enqueue(i));

// Потребитель
while (queue.TryDequeue(out int item))
    Console.WriteLine($"Обработан элемент: {item}");
```

Concurrent-коллекции упрощают код и снижают риск ошибок синхронизации, однако не заменяют `lock` полностью — если нужно атомарно выполнить несколько операций над коллекцией, ручная синхронизация всё равно потребуется.



7. Отмена, таймауты и обработка исключений

В реальных приложениях фоновые и асинхронные операции должны быть **управляемыми**: их нужно уметь корректно останавливать, ограничивать по времени и надёжно обрабатывать ошибки. Без этого многопоточный и асинхронный код становится источником зависаний, утечек ресурсов и трудноуловимых багов.

Зачем нужна отмена операций

Отмена необходима, когда пользователь отменяет действие, операция больше не актуальна (например, устаревший запрос), превышен лимит времени или приложение завершает работу.

Важно понимать: отмена в .NET **кооперативная** — операция должна сама проверять сигнал отмены и корректно завершаться. Принудительной остановки потоков не существует.

CancellationToken и CancellationTokenSource

Механизм отмены строится на двух элементах:

- `CancellationTokenSource` — создаёт сигнал отмены и управляет им;
- `CancellationToken` — передаётся в асинхронные методы и проверяется ими.

Типичный сценарий: создаём `CancellationTokenSource`, передаём его токен в метод, и через некоторое время вызываем `Cancel()`:

```
async Task DoWorkAsync(CancellationToken token)
{
    for (int i = 0; i < 10; i++)
    {
        token.ThrowIfCancellationRequested();
        await Task.Delay(500, token);
        Console.WriteLine($"Шаг {i}");
    }
}

async Task ExampleAsync()
{
    var cts = new CancellationTokenSource();

    Task task = DoWorkAsync(cts.Token);

    await Task.Delay(2000);
    cts.Cancel(); // отправляем сигнал отмены

    try
    {
        await task;
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Операция отменена");
    }
}
```

После вызова `cts.Cancel()` токен переходит в состояние «отменён». При следующей проверке `ThrowIfCancellationRequested()` внутри `DoWorkAsync` будет выброшено исключение `OperationCanceledException`, которое мы перехватываем снаружи.

Проверка отмены

Есть два основных способа проверить, была ли запрошена отмена.

Явная проверка — подходит, когда нужно завершить работу тихо, без исключения:

```
if (token.IsCancellationRequested)
    return;
```

Генерация исключения — предпочтительный способ для асинхронных методов. Исключение поднимается по стеку вызовов и может быть перехвачено в одном месте:

```
token.ThrowIfCancellationRequested();
```

Отмену следует считать **штатным сценарием**, а не ошибкой — именно поэтому для неё существует отдельный тип исключения `OperationCanceledException`, а не общий `Exception`.

Отмена в Task и Parallel

Task поддерживает отмену через `CancellationToken` — задача сама должна проверять токен в процессе работы:

```
var cts = new CancellationTokenSource();
var token = cts.Token;

Task task = Task.Run(() => {
    while (true) {
        token.ThrowIfCancellationRequested();
        // работа
    }
}, token);
```

Передача токена вторым аргументом в `Task.Run` позволяет корректно перевести задачу в состояние `Cancelled`, а не `Faulted`.

Parallel принимает токен через `ParallelOptions`:

```
var cts = new CancellationTokenSource();
var options = new ParallelOptions { CancellationToken = cts.Token };

try {
    Parallel.For(0, 100, options, i => {
        options.CancellationToken.ThrowIfCancellationRequested();
        // работа
    });
}
catch (OperationCanceledException) {
    Console.WriteLine("Параллельная операция отменена");
}
```

Таймауты

Таймаут — ограничение времени выполнения операции. Если операция не укладывается в отведённое время, она отменяется.

Способ 1: CancellationTokenSource с таймаутом — самый простой и распространённый. Токен автоматически отменяется по истечении заданного времени:

```
var cts = new CancellationTokenSource(TimeSpan.FromSeconds(2));
try {
    await DoWorkAsync(cts.Token);
}
catch (OperationCanceledException) {
    Console.WriteLine("Превышено время ожидания");
}
```

Способ 2: Task.WhenAny как таймер — удобен, когда нужно отреагировать на таймаут, не прерывая саму операцию, или когда метод не принимает токен:

```
var cts = new CancellationTokenSource();
Task work = DoWorkAsync(cts.Token);
Task timeout = Task.Delay(2000);
Task completed = await Task.WhenAny(work, timeout);
if (completed == timeout) {
    cts.Cancel(); // отменяем операцию
    Console.WriteLine("Превышено время ожидания");
}
```

`Task.WhenAny` завершается, как только завершится любая из переданных задач. Если первой завершилась `timeout`, значит операция не уложилась в отведённое время.

Практическое задание

Что нужно сделать:

В этом задании нужно будет реализовать имитацию параллельной загрузки файлов с отображением прогресс-баров в консоли.

1. Добавить в проект команду:

   ```
   load <количество_скачиваний> <размер_скачиваний>
   ```

   Пример:

   ```
   load 3 100
   ```

   Это означает:
   - создать 3 параллельные загрузки
   - каждая загрузка должна увеличивать прогресс от 0 до 100
   - для каждой загрузки отображается отдельный прогресс-бар
   - все прогресс-бары должны обновляться одновременно
2. Реализация `LoadCommand`

   Асинхронная логика должна находиться в методе:

   ```
   private async Task RunAsync()
   ```

   Метод `Execute()` должен остаться синхронным, это можно сделать например так:

   ```
   public void Execute() { RunAsync().Wait(); }
   ```

3. Проверка аргументов

   Необходимо:

   - проверить, что аргументы переданы
   - проверить, что это числа
   - проверить, что значения > 0

   При ошибке выбрасывать пользовательское исключение.
4. Резервирование строк под прогресс-бары

   Перед запуском задач необходимо:

   1. Запомнить текущую строку:

      ```
      int startRow = Console.CursorTop;
      ```

   2. Вывести пустые строки по количеству загрузок:

      ```
      for (int i = 0; i < downloadsCount; i++)
          Console.WriteLine();
      ```

   Теперь у каждой загрузки есть собственная строка:

   ```
   row = startRow + index
   ```

5. Запуск параллельных задач

   В `RunAsync()`:

   - создать список `List<Task>`
   - в цикле добавить задачи `DownloadAsync(index)`
   - вызвать `await Task.WhenAll(tasks)`
   - После завершения всех задач выводится:

     ```
     Все загрузки завершены.
     ```

   Запрещено:
   - запускать задачи последовательно
   - использовать `Thread.Sleep()`
6. Имитация загрузки

   В методе `DownloadAsync`:
   - цикл от `0` до `размер_скачиваний`
   - на каждой итерации:
     - вычислить процент
     - обновить прогресс-бар
     - сделать `await Task.Delay(...)` со случайной задержкой

7. Реализация прогресс-бара

   - 20 делений
   - каждые 5% добавляется один `#`
   - незаполненная часть `-`
   - справа отображается процент

   Пример:

   ```
   [##########----------] 50%
   ```

8. Потокобезопасная работа с консолью

   Консоль — общий ресурс и несколько задач одновременно пишут в неё.

   поэтому необходимо:

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

   При обновлении строки:

   ```
   lock (_consoleLock)
   {
       Console.SetCursorPosition(0, row);
       Console.Write(bar);
   }
   ```