IO-bound vs CPU-bound

I/O-bound операции (Input/Output Bound)

Что это?

Операции, где основное время тратится на **ожидание** внешних ресурсов, а не на вычисления.

Характеристики:

- **Ждут** ответа от внешних систем
- **Не нагружают** CPU процессор
- **Освобождают** поток во время ожидания

Примеры:

```
// 1. Работа с файловой системой
await File.ReadAllTextAsync("file.txt");

// 2. Сетевые запросы
await httpClient.GetStringAsync("https://api.example.com");

// 3. Базы данных
await dbContext.Users.ToListAsync();

// 4. Взаимодействие с оборудованием
await printer.PrintDocumentAsync(document);

// 5. Таймеры (имитация I/O)
await Task.Delay(1000);
```

Как работают в .NET:

```
public async Task<string> DownloadDataAsync()
{
   // 1. Поток инициирует операцию
   var task = httpClient.GetStringAsync(url);

   // 2. Поток ОСВОБОЖДАЕТСЯ для других задач
   // (здесь происходит магия I/O completion ports)

   // 3. Когда данные готовы, продолжается выполнение
   string result = await task;
   return result;
}
```

CPU-bound операции (Compute Bound)

Что это?

Операции, где основное время тратится на **вычисления** процессором.

Характеристики:

- **Нагружают** CPU процессор
- **Требуют** активной работы потока
- **Не освобождают** поток во время выполнения

Примеры:

```
// 1. Математические вычисления
double result = CalculatePi(1000000);

// 2. Обработка изображений
Bitmap processedImage = ApplyFilters(originalImage);

// 3. Сортировка больших массивов
Array.Sort(largeArray);

// 4. Шифрование/дешифрование
byte[] encrypted = EncryptData(data, key);

// 5. Сложные алгоритмы
var path = FindShortestPath(graph, start, end);
```

Практическое применение

I/O-bound - используем async/await напрямую:

```
// ПРАВИЛЬНО - не блокирует потоки
public async Task<string> GetUserDataAsync(int userId)
{
   var user = await dbContext.Users.FindAsync(userId);  // I/O - ждем БД
   var avatar = await httpClient.GetByteArrayAsync(user.AvatarUrl); // I/O - ждем сеть
    await File.WriteAllBytesAsync($"/avatars/{userId}.jpg", avatar); // I/O - ждем диск

   return user.Name;
}
```

CPU-bound - используем Task.Run:

```
// ПРАВИЛЬНО - выносим в фон
public async Task<double> CalculateReportAsync()
{
   // Выносим тяжелые вычисления в пул потоков
   return await Task.Run(() => {
      double result = 0;
      for (int i = 0; i < 1000000; i++)
      {
          result += ComplexCalculation(i); // CPU-intensive
      }
      return result;
   });
}
```

Частые ошибки

Ошибка 1: Task.Run для I/O операций

```
// НЕПРАВИЛЬНО - лишняя обертка
var data = await Task.Run(() => httpClient.GetStringAsync(url));

// ПРАВИЛЬНО
var data = await httpClient.GetStringAsync(url);
```

Ошибка 2: Блокировка async вызовов

```
// НЕПРАВИЛЬНО - deadlock!
var data = httpClient.GetStringAsync(url).Result;

// ПРАВИЛЬНО
var data = await httpClient.GetStringAsync(url);
```

Производительность

I/O-bound:

- **Масштабирование**: тысячи одновременных операций
- **Ресурсы**: мало потоков, много асинхронных операций
- **Пример**: веб-сервер, обрабатывающий 10K запросов

CPU-bound:

- **Масштабирование**: ограничено количеством ядер CPU
- **Ресурсы**: много потоков, каждый занят вычислениями

- **Пример**: рендеринг видео, научные расчеты

Исходный код - идеальный пример I/O-bound:

```
static async Task MethodA()
{
   Console.WriteLine("A: начало");
   await Task.Delay(1000); // ← I/O-bound операция (ожидание)
   Console.WriteLine("A: конец");
}
```

Правильно: используется чистый async/await без Task.Run, потому что Task.Delay - типичная I/O операция, которая не нагружает CPU.

Механизм IO-bound операций

Основной принцип:

I/O-bound операции используют **аппаратную поддержку** на уровне ОС, а не просто потоки.

Пошаговый механизм:

Шаг 1: Инициализация операции

```
public async Task<string> ReadFileAsync(string path)
{
  // 1. Инициируем асинхронную операцию
  var task = File.ReadAllTextAsync(path);

  // 2. Управление передается ОС
  // .NET говорит драйверу файловой системы: "прочитай файл и сообщи, когда готово"

  // 3. Поток ВОЗВРАЩАЕТСЯ в пул
  // (здесь может выполняться другой код)

  string result = await task;
  return result;
}
```

Шаг 2: Что происходит на уровне ОС:

```
[Поток .NET] → [Драйвер файловой системы] → [ОС I/O Manager] → [Жесткий диск]
     ↑                                                    ↓
     ←-------------- I/O Completion Port ←----------------
```

Ключевые технологии:

1. I/O Completion Ports (IOCP)

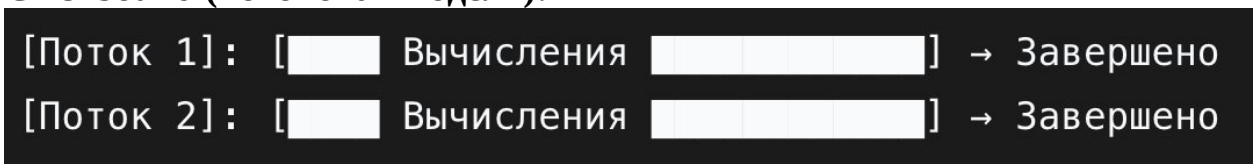
- Специальный механизм Windows для асинхронного I/O
- .NET использует IOCP для всех асинхронных операций
- Это аппаратно-оптимизированные очереди завершения

2. Overlapped I/O

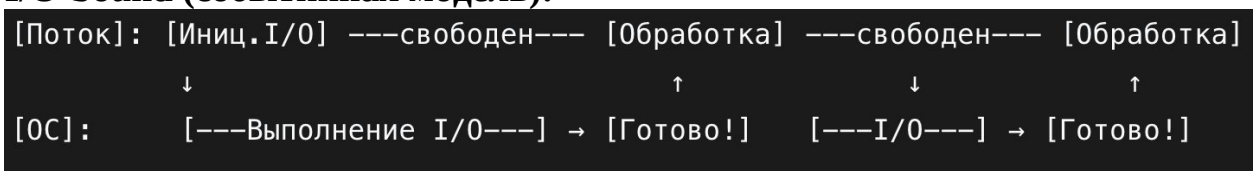
- Операция инициируется и сразу возвращает управление
- ОС сама управляет выполнением без участия потоков

Сравнение архитектур:

CPU-bound (потоковая модель):



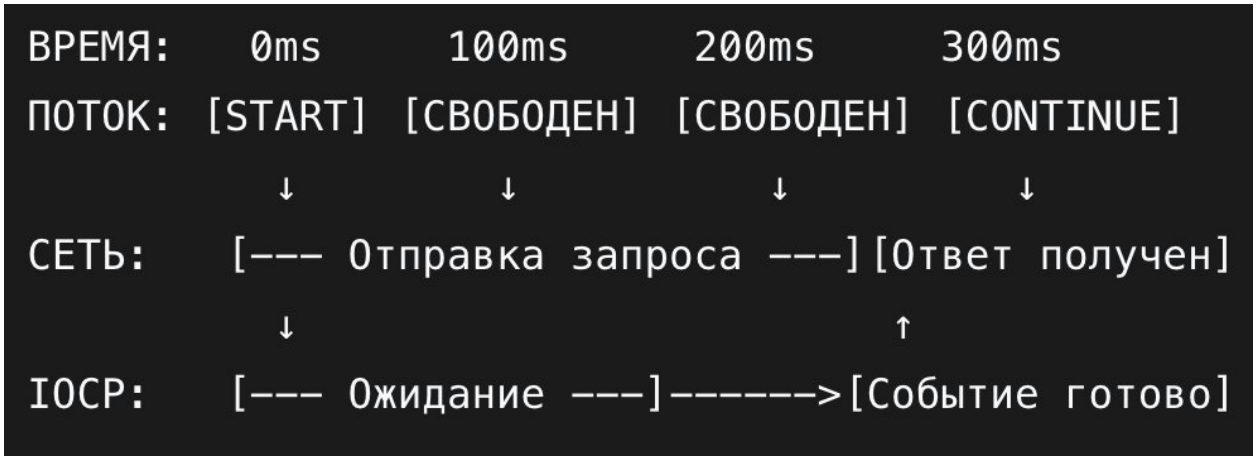
I/O-bound (событийная модель):



Детальный пример работы:

```
public async Task WebRequestExample()
{
  // ШАГ 1: Инициализация
   Console.WriteLine("1. Поток инициирует запрос");

  // ШАГ 2: Начинаем асинхронную операцию
  var response = await httpClient.GetStringAsync("http://example.com");

  // Что происходит внутри:
  // 1. .NET создает объект Task
  // 2. Вызывается Winsock (сетевая подсистема Windows)
  // 3. ОС отправляет сетевой пакет и регистрирует callback в IOCP
  // 4. Текущий поток ВОЗВРАЩАЕТСЯ в пул потоков!

   Console.WriteLine("2. Поток свободен для других задач");

  // ШАГ 3: Магия завершения
  // - Сетевой адаптер получает ответ
  // - ОС помещает событие в очередь IOCP
  // - Пул потоков получает свободный поток (не обязательно тот же!)
  // - Выполнение продолжается с этого места

   Console.WriteLine($"3. Получен ответ: {response.Length} символов");
}
```

Визуализация процесса:



Ключевые преимущества:

1. Экономия потоков

```
// 10,000 одновременных HTTP запросов:
var tasks = new List<Task>();
for (int i = 0; i < 10000; i++)
{
   tasks.Add(httpClient.GetStringAsync(url));
}
await Task.WhenAll(tasks);
// Потоков: ~10-20, а не 10,000!
```

2. Масштабируемость

- CPU-bound: ограничено ядрами CPU
- I/O-bound: ограничено аппаратными возможностями I/O системы

3. Нулевая нагрузка во время ожидания

```
await Task.Delay(5000); // 5 секунд - 0% загрузки CPU
```

Техническая реализация в .NET:

Уровень .NET:

```
// Task.Delay использует System.Threading.Timer
// который использует очередь TimerQueue
// которая использует IOCP для уведомлений
```

Уровень ОС:

- **Файлы**: FileStream с useAsync: true
- **Сеть**: Socket с overlapped I/O

- **HTTP**: HttpClient на основе сокетов
- **Базы данных**: драйверы используют IOCP

Важные нюансы:

1. “Поток не равен операции”

- Один поток может обслуживать тысячи I/O операций
- Операция может быть обработана разными потоками

2. Тайминги:

```
var sw = Stopwatch.StartNew();
await Task.Delay(1000);
Console.WriteLine(sw.ElapsedMilliseconds); // ~1000ms
// Но поток был занят всего ~0.1ms!
```

3. Синхронный vs асинхронный I/O:

```
// СИНХРОННО (плохо):
var data = File.ReadAllText("file.txt"); // Поток заблокирован

// АСИНХРОННО (хорошо):
var data = await File.ReadAllTextAsync("file.txt"); // Поток свободен
```

Вывод:

CPU-bound = “занимаем поток вычислениями”
I/O-bound = “говорим ОС сделать работу и сообщить когда готово”

I/O-bound операции используют аппаратные возможности ОС (IOCP), что позволяет достигать огромной масштабируемости без создания тысяч потоков!

Механизм CPU-bound операций

Что такое CPU-bound?

CPU-bound операции - это задачи, где основное время выполнения тратится на **вычисления процессора**, а не на ожидание внешних ресурсов.

Ключевые характеристики:

- **Нагружают** центральный процессор (CPU)
- **Требуют** активной работы потока все время выполнения
- **Масштабируются** по количеству ядер CPU
- **Не освобождают** поток во время выполнения

Механизм работы CPU-bound

Основной принцип:

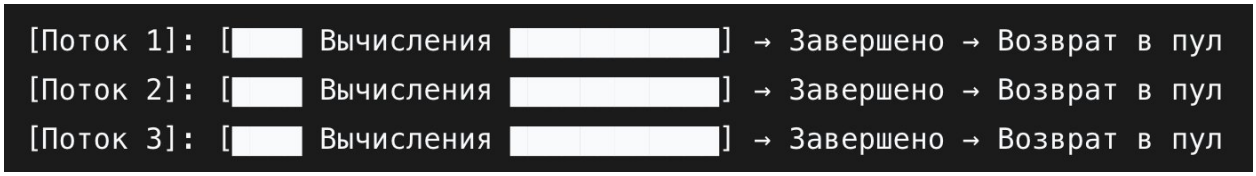
CPU-bound операции выполняются **непосредственно в потоках** из пула потоков (.NET ThreadPool).

Пошаговый механизм:

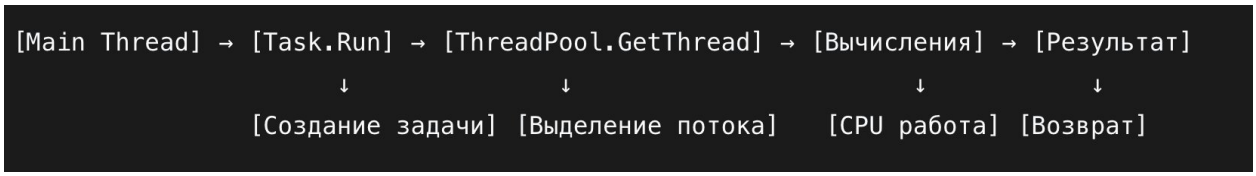
```
public async Task<int> CalculatePrimeNumbersAsync(int max)
{
   // ШАГ 1: Запрос потока из пула
   return await Task.Run(() =>
   {
      // ШАГ 2: Поток ЗАНИМАЕТСЯ вычислениями
      int count = 0;
      for (int i = 2; i <= max; i++)
      {
         if (IsPrime(i)) // CPU-intensive calculation
              count++;
      }

      // ШАГ 3: Поток возвращается в пул
      return count;
   });
}
```

Архитектура выполнения:

Визуализация потоков:



Детальная схема:



Ключевые отличия от I/O-bound:

Аспект	CPU-bound	I/O-bound
Ресурсы	Потоки ThreadPool	I/O Completion Ports
Блокировка	Поток занят	Поток освобожден
Масштабируемость	Ограничена ядрами CPU	Ограничена I/O системой
Паттерн	“Занимаем поток”	“Ждем уведомления”

Типичные примеры CPU-bound:

1. Математические вычисления

```
// Вычисление чисел Фибоначчи
public async Task<long> CalculateFibonacciAsync(int n)
{
   return await Task.Run(() =>
   {
      if (n <= 1) return n;

      long a = 0, b = 1;
      for (int i = 2; i <= n; i++)
      {
         long temp = a + b;
          a = b;
          b = temp;
```

```
        }
        return b;
    });
}
```

2. Обработка изображений

```
public async Task<Bitmap> ProcessImageAsync(Bitmap original)
{
    return await Task.Run(() =>
    {
        var result = new Bitmap(original.Width, original.Height);
        for (int x = 0; x < original.Width; x++)
        {
            for (int y = 0; y < original.Height; y++)
            {
                var pixel = ApplyFilters(original.GetPixel(x, y));
                result.SetPixel(x, y, pixel);
            }
        }
        return result;
    });
}
```

3. Сортировка и поиск

```
public async Task<int[]> SortLargeArrayAsync(int[] data)
{
    return await Task.Run(() =>
    {
        Array.Sort(data); // CPU-intensive operation
        return data;
    });
}
```

Управление потоками в ThreadPool:

Пул потоков .NET:

```
// Пример работы с ThreadPool
public async Task ParallelComputationsAsync()
{
    var tasks = new List<Task>();

    for (int i = 0; i < Environment.ProcessorCount; i++)
    {
```

```
        tasks.Add(Task.Run(() => HeavyComputation(i)));
    }

    await Task.WhenAll(tasks);
}

// ThreadPool динамически управляет потоками:
// - Минимальное количество потоков
// - Максимальное количество потоков (по умолчанию ~1024)
// - Автоматическое создание/уничтожение потоков
```

Оптимизация CPU-bound операций:

1. Parallel.For / Parallel.ForEach

```
public async Task<int[]> ProcessDataParallelAsync(int[] data)
{
    return await Task.Run(() =>
    {
        var results = new int[data.Length];
        Parallel.For(0, data.Length, i =>
        {
            results[i] = ExpensiveCalculation(data[i]);
        });
        return results;
    });
}
```

2. PLINQ (Parallel LINQ)

```
public async Task<int[]> ProcessWithPLinqAsync(int[] data)
{
    return await Task.Run(() =>
    {
        return data.AsParallel()
                   .Where(x => IsPrime(x))
                   .Select(x => Transform(x))
                   .ToArray();
    });
}
```

3. Task.Run с ограничениями

```
// Ограничиваем параллелизм по количеству ядер
private static SemaphoreSlim _throttler =
    new SemaphoreSlim(Environment.ProcessorCount);
```

```
public async Task ThrottledComputationAsync()
{
   await _throttler.WaitAsync();
   try
   {
      await Task.Run(() => HeavyWork());
   }
   finally
   {
      _throttler.Release();
   }
}
```

Распространенные ошибки:

Ошибка 1: Task.Run для всего подряд

```
// НЕПРАВИЛЬНО - I/O операция в Task.Run
var data = await Task.Run(() => httpClient.GetStringAsync(url));

// ПРАВИЛЬНО - чистый async/await
var data = await httpClient.GetStringAsync(url);
```

Ошибка 2: Блокировка UI потоков

```
// НЕПРАВИЛЬНО в UI приложении
button.Click += (s, e) =>
{
   var result = HeavyCalculation(); // Блокирует UI!
};

// ПРАВИЛЬНО - выносим в Task.Run
button.Click += async (s, e) =>
{
   var result = await Task.Run(() => HeavyCalculation());
   UpdateUI(result);
};
```

Ошибка 3: Чрезмерное создание потоков

```
// НЕПРАВИЛЬНО - 1000 потоков для маленьких задач
for (int i = 0; i < 1000; i++)
{
   tasks.Add(Task.Run(() => LightCalculation())); // Overhead!
}
```

```
// ПРАВИЛЬНО - группируем или используем Parallel
await Task.Run(() =>
{
    Parallel.For(0, 1000, i => LightCalculation());
});
```

Производительность и мониторинг:

Метрики CPU-bound операций:

```
public async Task MonitorCpuUsageAsync()
{
    var process = Process.GetCurrentProcess();
    var startCpuTime = process.TotalProcessorTime;

    await HeavyComputationAsync();

    var endCpuTime = process.TotalProcessorTime;
    var cpuUsed = endCpuTime - startCpuTime;

    Console.WriteLine($"CPU time used: {cpuUsed.TotalMilliseconds}ms");
}
```

Оптимальное количество потоков:

```
// Используем количество логических процессоров
int optimalDegree = Environment.ProcessorCount;

var options = new ParallelOptions
{
    MaxDegreeOfParallelism = optimalDegree
};

Parallel.For(0, data.Length, options, i =>
{
    ProcessItem(data[i]);
});
```

Практический пример из вашего кода:

```
// ЕСЛИ БЫ ваши методы были CPU-bound:
static async Task MethodA()
{
    Console.WriteLine("A: начало");
```

```
    // Вместо Task.Delay - тяжелые вычисления
    await Task.Run(() =>
    {
        // CPU-intensive work
        for (int i = 0; i < 1000000; i++)
        {
            Math.Sqrt(i) * Math.Cos(i);
        }
    });

    Console.WriteLine("A: конец");
}
```

Сравнительная таблица:

Параметр	CPU-bound	I/O-bound
Основной ресурс	Процессор (CPU)	Внешние системы
Потоки	Заняты постоянно	Освобождаются при ожидании
Масштабируемость	Ядра CPU × потоки	Тысячи одновременных операций
Паттерн использования	Task.Run(() => computation)	await asyncOperation()
Ограничения	Количество ядер	Пропускная способность I/O

Ключевые выводы:

1. **CPU-bound = вычисления** → используем Task.Run
2. **I/O-bound = ожидание** → используем чистый async/await

3. **Потоки дороги** - создаем их только для реальных вычислений
4. **Мониторим загрузку CPU** - понимаем, когда нужно оптимизировать