Асинхронность и многопоточность в C#

Многозадачность

Многозадачность

- Способность системы или программы выполнять несколько задач (процессов) одновременно или переключаться между ними.

Многозадачность

- Способность системы или программы выполнять несколько задач (процессов) одновременно или переключаться между ними.
- Существует как на уровне операционной системы (процессы), так и на уровне приложений (потоки).

Типы многозадачности

- **Кооперативная (cooperative)**: задачи сами передают управление системе.

Типы многозадачности

- **Кооперативная (cooperative)**: задачи сами передают управление системе.
  - Старые ОС, например, Windows 3.x.

Типы многозадачности

- **Кооперативная (cooperative)**: задачи сами передают управление системе.
  - Старые ОС, например, Windows 3.x.
  - Может все “зависнуть”, если задача не отдала управление.

Типы многозадачности

- **Кооперативная (cooperative)**: задачи сами передают управление системе.
  - Старые ОС, например, Windows 3.x.
  - Может все “зависнуть”, если задача не отдала управление.
- **Вытесняющая (preemptive)**: операционная система управляет переключением между задачами.

Типы многозадачности

- **Кооперативная (cooperative)**: задачи сами передают управление системе.
  - Старые ОС, например, Windows 3.x.
  - Может все “зависнуть”, если задача не отдала управление.
- **Вытесняющая (preemptive)**: операционная система управляет переключением между задачами.
  - Современные ОС.

Типы многозадачности

- **Кооперативная (cooperative)**: задачи сами передают управление системе.
  - Старые ОС, например, Windows 3.x.
  - Может все “зависнуть”, если задача не отдала управление.
- **Вытесняющая (preemptive)**: операционная система управляет переключением между задачами.
  - Современные ОС.
  - Если задача “зависла”, остальные продолжают работать.

Переключение контекста

Переключение задач при многозадачности

Алгоритм переключения:

1. Приостанавливается выполнение текущей задачи (либо задача возвращает управление).

Переключение задач при многозадачности

Алгоритм переключения:

1. Приостанавливается выполнение текущей задачи (либо задача возвращает управление).
2. Сохраняется состояние задачи (контекст).

Переключение задач при многозадачности

Алгоритм переключения:

1. Приостанавливается выполнение текущей задачи (либо задача возвращает управление).
2. Сохраняется состояние задачи (контекст).
3. Выбирается следующая задача для выполнения.

Переключение задач при многозадачности

Алгоритм переключения:

1. Приостанавливается выполнение текущей задачи (либо задача возвращает управление).
2. Сохраняется состояние задачи (контекст).
3. Выбирается следующая задача для выполнения.
4. Восстанавливается состояние выбранной задачи и передается ей управление.

Переключение задач при многозадачности

Механизмы необходимые для переключения задач:

1. Контекст задачи (ее состояние):
    - Блок управления процессом (Process Control Block, PCB).
    - Блок управления потоком (Thread Control Block, TCB).
2. Планировщик задач (Scheduler).
    - Определяет последовательность выполнения задач.
3. Переключение контекстов (Context Switching).
    - Сохраняет и восстанавливает состояния задач.

Переключение задач при многозадачности

Механизмы необходимые для переключения задач:

1. Контекст задачи (ее состояние):
    - Блок управления процессом (Process Control Block, PCB).
    - Блок управления потоком (Thread Control Block, TCB).
2. Планировщик задач (Scheduler).
    - Определяет последовательность выполнения задач.
3. Переключение контекстов (Context Switching).
    - Сохраняет и восстанавливает состояния задач.

Переключение задач при многозадачности

Механизмы необходимые для переключения задач:

1. Контекст задачи (ее состояние):
    - Блок управления процессом (Process Control Block, PCB).
    - Блок управления потоком (Thread Control Block, TCB).
2. Планировщик задач (Scheduler).
    - Определяет последовательность выполнения задач.
3. Переключение контекстов (Context Switching).
    - Сохраняет и восстанавливает состояния задач.

Контекст процесса PCB

- Идентификатор процесса.
- Состояние процесса (выполняется, ожидает, завершен, и т.д.).
- Приоритет процесса.
- Управление памятью (адресное пространство, таблицы страниц и пр.).
- Управление ресурсами (открытые файлы, устройства, и пр.).
- Информация о дочерних, родительских процессах.
- Список потоков (TCB).
- …

Контекст потока TCB

- Идентификатор потока.
- Состояние потока (выполняется, ожидает, завершен, и т.д.).
- Приоритет потока.
- Регистры процессора.
- Программный счетчик (указатель на следующую инструкцию).
- Указатель на стек потока.
- Указатель на PCB.
- …

Переключение между потоками одного процесса

1. Состояние текущего потока сохраняется в его TCB.
2. Состояние следующего потока загружается из его TCB.
3. Управление передается следующему потоку.

Адресное пространство не меняется, т.к. оба потока из одного процесса.

Переключение между потоками одного процесса

1. Состояние текущего потока сохраняется в его TCB.
2. Состояние следующего потока загружается из его TCB.
3. Управление передается следующему потоку.

Адресное пространство не меняется, т.к. оба потока из одного процесса.

Переключение между потоками одного процесса

1. Состояние текущего потока сохраняется в его TCB.
2. Состояние следующего потока загружается из его TCB.
3. Управление передается следующему потоку.

Адресное пространство не меняется, т.к. оба потока из одного процесса.

Переключение между потоками одного процесса

1. Состояние текущего потока сохраняется в его TCB.
2. Состояние следующего потока загружается из его TCB.
3. Управление передается следующему потоку.

Адресное пространство не меняется, т.к. оба потока из одного процесса.

Переключение между потоками разных процессов

1. Состояние текущего потока сохраняется в его TCB.
2. Состояние текущего процесса сохраняется в PCB (если требуется).
3. Переключение адресного пространства.
4. Состояние следующего потока загружается из его TCB.
5. Управление передается следующему потоку.

Переключение между потоками разных процессов

1. Состояние текущего потока сохраняется в его TCB.
2. Состояние текущего процесса сохраняется в PCB (если требуется).
3. Переключение адресного пространства.
4. Состояние следующего потока загружается из его TCB.
5. Управление передается следующему потоку.

Переключение между потоками разных процессов

1. Состояние текущего потока сохраняется в его TCB.
2. Состояние текущего процесса сохраняется в PCB (если требуется).
3. Переключение адресного пространства.
4. Состояние следующего потока загружается из его TCB.
5. Управление передается следующему потоку.

Переключение между потоками разных процессов

1. Состояние текущего потока сохраняется в его TCB.
2. Состояние текущего процесса сохраняется в PCB (если требуется).
3. Переключение адресного пространства.
4. Состояние следующего потока загружается из его TCB.
5. Управление передается следующему потоку.

Переключение между потоками разных процессов

1. Состояние текущего потока сохраняется в его TCB.
2. Состояние текущего процесса сохраняется в PCB (если требуется).
3. Переключение адресного пространства.
4. Состояние следующего потока загружается из его TCB.
5. Управление передается следующему потоку.

Когда применяются процессы в приложении

1. Изоляция ресурсов:
    - Выполнение небезопасного, стороннего API.
    - Взаимодействие с внешними системами, которые требуют изоляции (сторонние программы).
2. Создание дочерних процессов:
    - Код надо выполнить в отдельном окружении.
3. Отказоустойчивость:
    - Если один процесс завершился с ошибкой, другие продолжают работать.
4. Распределенные системы.

Когда применяются процессы в приложении

1. Изоляция ресурсов:
    - Выполнение небезопасного, стороннего API.
    - Взаимодействие с внешними системами, которые требуют изоляции (сторонние программы).
2. Создание дочерних процессов:
    - Код надо выполнить в отдельном окружении.
3. Отказоустойчивость:
    - Если один процесс завершился с ошибкой, другие продолжают работать.
4. Распределенные системы.

Когда применяются процессы в приложении

1. Изоляция ресурсов:
    - Выполнение небезопасного, стороннего API.
    - Взаимодействие с внешними системами, которые требуют изоляции (сторонние программы).
2. Создание дочерних процессов:
    - Код надо выполнить в отдельном окружении.
3. Отказоустойчивость:
    - Если один процесс завершился с ошибкой, другие продолжают работать.
4. Распределенные системы.

Когда применяются процессы в приложении

1. Изоляция ресурсов:
    - Выполнение небезопасного, стороннего API.
    - Взаимодействие с внешними системами, которые требуют изоляции (сторонние программы).
2. Создание дочерних процессов:
    - Код надо выполнить в отдельном окружении.
3. Отказоустойчивость:
    - Если один процесс завершился с ошибкой, другие продолжают работать.
4. Распределенные системы.

Многопоточность

Многопоточность

Способ реализации многозадачности на уровне потоков (threads) внутри одного процесса.

Особенности:

- Потоки разделяют общие ресурсы процесса (память, файлы), но имеют собственные стеки и регистры.
- Потоки могут выполняться параллельно на разных ядрах процессора.

Многопоточность

Способ реализации многозадачности на уровне потоков (threads) внутри одного процесса.

Особенности:

- Потоки разделяют общие ресурсы процесса (память, файлы), но имеют собственные стеки и регистры.
- Потоки могут выполняться параллельно на разных ядрах процессора.

Многопоточность

Способ реализации многозадачности на уровне потоков (threads) внутри одного процесса.

Особенности:

- Потоки разделяют общие ресурсы процесса (память, файлы), но имеют собственные стеки и регистры.
- Потоки могут выполняться параллельно на разных ядрах процессора.

Многопоточность

Преимущества:

- Повышение производительности.
- Отзывчивость приложения.
- Экономия ресурсов.

Недостатки:

- Сложность разработки.
- Проблемы синхронизации.
- Увеличение накладных расходов.
- Сложность отладки.
- Проблемы с производительностью.

Многопоточность

Преимущества:

- Повышение производительности.
- Отзывчивость приложения.
- Экономия ресурсов.

Недостатки:

- Сложность разработки.
- Проблемы синхронизации.
- Увеличение накладных расходов.
- Сложность отладки.
- Проблемы с производительностью.

Проблемы синхронизации потоков

1. Состояние гонки (race condition).
2. Взаимная блокировка (deadlock).
3. Голодание потоков (starvation).
4. Инверсия приоритетов (priority inversion).
5. Ложное разделение кэша (false sharing).
6. Избыточная синхронизация.

Проблемы синхронизации потоков

1. Состояние гонки (race condition).
2. Взаимная блокировка (deadlock).
3. Голодание потоков (starvation).
4. Инверсия приоритетов (priority inversion).
5. Ложное разделение кэша (false sharing).
6. Избыточная синхронизация.

Проблемы синхронизации потоков

1. Состояние гонки (race condition).
2. Взаимная блокировка (deadlock).
3. Голодание потоков (starvation).
4. Инверсия приоритетов (priority inversion).
5. Ложное разделение кэша (false sharing).
6. Избыточная синхронизация.

Проблемы синхронизации потоков

1. Состояние гонки (race condition).
2. Взаимная блокировка (deadlock).
3. Голодание потоков (starvation).
4. Инверсия приоритетов (priority inversion).
5. Ложное разделение кэша (false sharing).
6. Избыточная синхронизация.

Проблемы синхронизации потоков

1. Состояние гонки (race condition).
2. Взаимная блокировка (deadlock).
3. Голодание потоков (starvation).
4. Инверсия приоритетов (priority inversion).
5. Ложное разделение кэша (false sharing).
6. Избыточная синхронизация.

Проблемы синхронизации потоков

1. Состояние гонки (race condition).
2. Взаимная блокировка (deadlock).
3. Голодание потоков (starvation).
4. Инверсия приоритетов (priority inversion).
5. Ложное разделение кэша (false sharing).
6. Избыточная синхронизация.

Когда применяются потоки в .Net

1. Требуется полный контроль на потоком.
2. Долгоживущие операции.
3. Поток с особыми настройками.
4. Работа с блокирующими операциями.
5. Специфические сценарии многопоточности.
6. Избежание накладных расходов пула потоков.

Когда применяются потоки в .Net

1. Требуется полный контроль на потоком.
2. Долгоживущие операции.
3. Поток с особыми настройками.
4. Работа с блокирующими операциями.
5. Специфические сценарии многопоточности.
6. Избежание накладных расходов пула потоков.

Когда применяются потоки в .Net

1. Требуется полный контроль на потоком.
2. Долгоживущие операции.
3. Поток с особыми настройками.
4. Работа с блокирующими операциями.
5. Специфические сценарии многопоточности.
6. Избежание накладных расходов пула потоков.

Когда применяются потоки в .Net

1. Требуется полный контроль на потоком.
2. Долгоживущие операции.
3. Поток с особыми настройками.
4. Работа с блокирующими операциями.
5. Специфические сценарии многопоточности.
6. Избежание накладных расходов пула потоков.

Когда применяются потоки в .Net

1. Требуется полный контроль на потоком.
2. Долгоживущие операции.
3. Поток с особыми настройками.
4. Работа с блокирующими операциями.
5. Специфические сценарии многопоточности.
6. Избежание накладных расходов пула потоков.

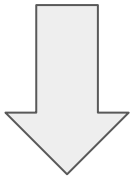
Когда применяются потоки в .Net

1. Требуется полный контроль на потоком.
2. Долгоживущие операции.
3. Поток с особыми настройками.
4. Работа с блокирующими операциями.
5. Специфические сценарии многопоточности.
6. Избежание накладных расходов пула потоков.

Асинхронность в C#

Асинхронность

- Выполнение задач, при котором программа не ждет завершения одной операции, прежде чем начать другую.
- Вместо блокировки потока выполнения программа продолжает работать, а результат операции обрабатывается позже, когда он будет готов.



Повышение отзывчивости +
Эффективное использование ресурсов

Asynchronous Programming Model (APM)

```
sealed class Handler
{
    //Синхронная версия
    public int DoStuff(string arg);

    //Асинхронная версия метода DoStuff
    public IAsyncResult BeginDoStuff(string arg, AsyncCallback? callback, object? state);
    public int EndDoStuff(IAsyncResult asyncResult);
}
```

Begin/End паттерн
или
IAsyncResult паттерн

Основные компоненты APM

1. Метод `BeginXxx`:
    - запускает асинхронную операцию;
    - возвращает объект `IAsyncResult` , который используется для отслеживания состояния операции;
    - принимает параметры, необходимые для операции (например, буфер для данных), а также callback-метод, который будет вызван при завершении, и объект состояния.

Основные компоненты APM

1. Метод `BeginXxx`.
2. Метод `EndXxx` :
   - завершает асинхронную операцию;
   - возвращает результат операции;
   - должен вызываться после завершения операции, чтобы освободить ресурсы и получить результат.

Основные компоненты APM

1. Метод `BeginXxx`.
2. Метод `EndXxx`.
3. Интерфейс `IAsyncResult`:

```
public interface IAsyncResult {
    //Объект состояния, который был передан в метод BeginXxx
    object? AsyncState { get; }
    //Объект синхронизации для ожидания завершения асинхронной операции
    WaitHandle AsyncWaitHandle { get; }
    //Завершилась ли операция
    bool IsCompleted { get; }
    //Завершилась ли операция синхронно (т.е. сразу после BeginXxx)
    bool CompletedSynchronously { get; }
}
```

Основные компоненты APM

1. Метод `BeginXxx`.
2. Метод `EndXxx`.
3. Интерфейс `IAsyncResult`.
4. Callback-метод (continue):
    - Вызывается при завершении асинхронной операции.
    - Принимает `IAsyncResult`.

```
public delegate void AsyncCallback(IAsyncResult ar);
```

Основные компоненты APM

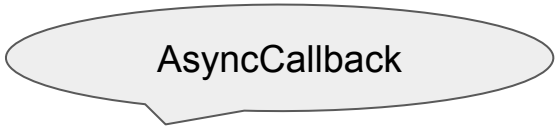
1. Метод `BeginXxx`.
2. Метод `EndXxx`.
3. Интерфейс `IAsyncResult`.
4. Callback-метод (continue).
5. Состояние (state object):
    - Объект, который передается в `BeginXxx` и может быть использован в callback-методе.

Пример APM

```
try
{
    int i = handler.DoStuff(arg);
    Use(i);
}
catch (Exception e)
{
    ... // handle exceptions from DoStuff and Use
}
```

Синхронная версия

Пример APM




```
try
{
    IAsyncResult result = handler.BeginDoStuff(arg, iar =>
    {
        try
        {
            Handler handler = (Handler)iar.AsyncState!;
            int i = handler.EndDoStuff(iar);
            Use(i);
        }
        catch (Exception e2)
        {
            ... // handle exceptions from EndDoStuff and Use
        }
    }, handler);

    //Можем что-нибудь поделать
    //Ожидаем завершение асинхронной операции
    result.AsyncWaitHandle.WaitOne();
}
catch (Exception e){ ... // handle exceptions thrown from the synchronous call to BeginDoStuff }
```

Асинхронная версия

Как достигается асинхронность в APM

Имплементация `BeginXxx` определяет как будет достигаться асинхронность операции. Какие есть варианты?

Как достигается асинхронность в APM

Имплементация `BeginXxx` определяет как будет достигаться асинхронность операции. Какие есть варианты?

1. Стандартные механизмы ОС.
    - Например, асинхронность I/O операций (чтение файла) достигается за счет **I/O Completion Ports** механизма в Windows. Но callback будет вызываться в отдельном потоке из пула потоков.

Как достигается асинхронность в APM

Имплементация `BeginXxx` определяет как будет достигаться асинхронность операции. Какие есть варианты?

1. Стандартные механизмы ОС.
2. Пул потоков.
    - Стандартная имплементация `BeginInvoke`/`EndInvoke` делегатов использует пул потоков как для самого делегата, так и для callback’а.

Как достигается асинхронность в APM

Имплементация `BeginXxx` определяет как будет достигаться асинхронность операции. Какие есть варианты?

1. Стандартные механизмы ОС.
2. Пул потоков.
3. Самостоятельное создание потока.

Как достигается асинхронность в APM

Имплементация `BeginXxx` определяет как будет достигаться асинхронность операции. Какие есть варианты?

1. Стандартные механизмы ОС.
2. Пул потоков.
3. Самостоятельное создание потока.

!!! Обновление UI внутри callback'а:

- UI элементы могут обновляться только на UI потоке.
- Приходилось использовать `Control.Invoke`/`BeginInvoke`.

Недостатки APM

1. Сложность написания и поддержки.
2. Требуется ручное управление состоянием и callback’ами.
3. Комбинировать асинхронные методы не удобно.
4. Есть возможность получить stack overflow из-за callback’ов.
5. Сложно управлять исключениями.
6. Отсутствие поддержки отмены асинхронных операций.

Event-based Asynchronous Pattern (EAP)

```
class Handler
{
    public int DoStuff(string arg);

    public void DoStuffAsync(string arg, object? userToken);
    public event DoStuffEventHandler? DoStuffCompleted;
}

public delegate void DoStuffEventHandler(object sender, DoStuffEventArgs e);

public class DoStuffEventArgs : AsyncCompletedEventArgs
{
    public DoStuffEventArgs(int result, Exception? error, bool canceled,
        object? userToken) : base(error, canceled, userToken) => Result = result;

    public int Result { get; }
}
```

Async метод + event обработчик

Основные концепции EAP

1. Асинхронные методы:
    - Выполняют асинхронные операции, имеют суффикс `Async`.

Основные концепции EAP

1. Асинхронные методы.
2. События:
    - После завершения асинхронной операции генерируется событие с суффиксом `Completed`.

Основные концепции EAP

1. Асинхронные методы.
2. События.
3. Аргумент события:
    - Событие передает объект с данными о результате операции.

Основные концепции EAP

1. Асинхронные методы.
2. События.
3. Аргумент события.
4. Отмена операции:
    - EAP поддерживает отмену асинхронных операций через метод `CancelAsync`.

Отмена операций в EAP

1. На уровне асинхронного метода имплементируется метод `CancelAsync` или `MethodNameAsyncCancel`.
2. Вызов `CancelAsync` запрашивает отмену асинхронной операции.
3. После отмены операции генерируется событие `Completed`. В аргументах события `AsyncCompletedEventArgs` свойство `Cancelled` будет иметь значение `true`.
4. Обработчик события `Completed` должен проверить `Cancelled` флаг об отмене операции.

SynchronizationContext в EAP

`SynchronizationContext`:

- это абстракция над планировщиком;
- базовая реализация представляет из себя ThreadPool;
- метод `SynchronizationContext.Post` помещает переданный callback в очередь к планировщику, который содержит.

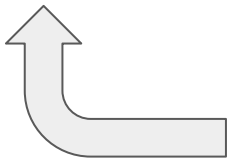
Базовый SynchronizationContext

```
public partial class SynchronizationContext
{
    public virtual void Post(SendOrPostCallback d, object? state) =>
        ThreadPool.QueueUserWorkItem(
            static s => s.d(s.state),
            (d, state),
            preferLocal: false);
    ...
}
```

Помещаем callback в очередь пула потоков

SynchronizationContext в WPF

```
public sealed class DispatcherSynchronizationContext
    : SynchronizationContext
{
   public override void Post(SendOrPostCallback d, Object state) =>
       _dispatcher.BeginInvoke(_priority, d, state);
   ...
}
```



Выполняем callback на UI потоке

Пример использования SynchronizationContext

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    ThreadPool.QueueUserWorkItem(_ =>
    {
        string message = ComputeMessage();

        button1.Dispatcher.InvokeAsync(() =>
        {
            button1.Content = message;
        });
    });
}
```

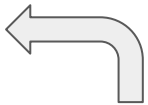
Поток из ThreadPool

UI поток

Пример использования SynchronizationContext

```
static void ComputeMessageAndInvokeUpdate(Action<string> update)
{
    SynchronizationContext? sc = SynchronizationContext.Current;
    ThreadPool.QueueUserWorkItem(_ =>
    {
        string message = ComputeMessage();
        if (sc is not null)
        {
            sc.Post(_ => update(message), null);
        }
        else
        {
            update(message);
        }
    });
}
```

Поток из ThreadPool



Поток выполнения зависит от планировщика контекста

SynchronizationContext в EAP

Стандартные классы (например, BackgroundWorker), которые поддерживают EAP используют SynchronizationContext для вызова обработчика событий в правильном потоке.

Недостатки EAP

1. Обработка результата выполняется только через обработчики событий.
2. Нет поддержки сложных сценариев отмены (например, цепочки задач).
3. Все еще сложно комбинировать асинхронные операции (например, ожидание завершения нескольких задач).

Task-based Asynchronous Pattern (TAP)

```
class Handler
{
    public int DoStuff(string arg);

    public async Task<int> DoStuffAsync(string arg);
}

var handler = new Handler();
int result = await handler.DoStuffAsync(message);
```

async/await + Task

Основные концепции TAP

1. `Task`:
    - Структура данных, представляющая асинхронную операцию.
    - Позволяет выполнять операцию асинхронно, не блокируя вызывающий поток.
    - `Task<TResult>` умеет возвращать результат `TResult` асинхронной операции (что-то вроде future).
    - Можно ждать завершения `Task` через `await`.
    - Можно комбинировать `Task`'и, создавать последовательные цепочки задач (continue или callback) или выполнять параллельно.

Основные концепции TAP

1. `Task`.
2. async/await - операторы:
    - async указывает, что метод является асинхронным;
    - await приостанавливает выполнение метода до завершения асинхронной операции, не блокируя вызывающий поток;
    - await работает только внутри async методов, и может использоваться с любым объектом, который имеет метод `GetAwaiter`.

Основные концепции TAP

1. `Task`.
2. async/await - операторы.
3. TAP поддерживает отмену асинхронных операций через `CancellationToken`.

TAP vs APM

1. В APM надо было для каждой операции создавать свой `IAsyncResult`.
2. `Task` позволяет задавать продолжение уже после вызова асинхронного метода. APM требует это знание до вызова метода `BeginXxx`.
3. В TAP нет проблем с:
    - комбинированием асинхронных операций;
    - отменой асинхронных операций.
4. TAP поддерживает более удобную работу с исключениями.

Упрощенная версия Task

```
class MyTask
{
    private bool _completed;
    private Exception? _error;
    private Action<MyTask>? _continuation;
    private ExecutionContext? _ec;
    //...
}
```

Завершилась ли задача

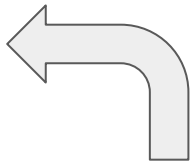
Упрощенная версия Task

```
class MyTask
{
    private bool _completed;
    private Exception? _error;
    private Action<MyTask>? _continuation;
    private ExecutionContext? _ec;
    //...
}
```

Исключение возникшее при выполнении задачи

Упрощенная версия Task

```
class MyTask
{
    private bool _completed;
    private Exception? _error;
    private Action<MyTask>? _continuation;
    private ExecutionContext? _ec;
    //...
}
```



Возможное продолжение задачи

Упрощенная версия Task

```
class MyTask
{
    private bool _completed;
    private Exception? _error;
    private Action<MyTask>? _continuation;
    private ExecutionContext? _ec;
    //...
}
```

Контекст выполнения текущего потока

ExecutionContext

Контекст выполнения текущего потока позволяет передавать следующие данные:

- локальные данные потока `AsyncLocal<T>`;
- CultureInfo;
- Principal: идентификатор и роль пользователя;
- и прочее.

ExecutionContext

Основные методы `ExecutionContext`:

- `ExecutionContext.Capture()` – захватывает текущий контекст выполнения.
- `ExecutionContext.Run(ExecutionContext, ContextCallback, object)` – выполняет код с переданным контекстом.
- `ExecutionContext.SuppressFlow()` – отключает передачу контекста между потоками.
- `ExecutionContext.RestoreFlow()` – восстанавливает передачу контекста.

ExecutionContext

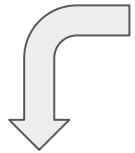
```
AsyncLocal<string> asyncLocal = new AsyncLocal<string>();
asyncLocal.Value = "Main Context";

ExecutionContext executionContext = ExecutionContext.Capture();
await Task.Run(() =>
{
    asyncLocal.Value = "Inside Task.Run";

    ExecutionContext.Run(executionContext, _ =>
    {
        Console.WriteLine($"Inside ExecutionContext.Run: {asyncLocal.Value}");
    }, null);

    Console.WriteLine($"After ExecutionContext.Run: {asyncLocal.Value}");
});

Console.WriteLine($"Outside Task.Run: {asyncLocal.Value}");
```

Упрощенная версия Task

```
class MyTask
{
    private bool _completed;
    private Exception? _error;
    private Action<MyTask>? _continuation;
    private ExecutionContext? _ec;
    //...
}
```

Упрощенная версия Task

```
public void ContinueWith(Action<MyTask> action)
{
    lock (this) //Пока считаем, что это mutex
    {
        if (_completed) {
            ThreadPool.QueueUserWorkItem(_ => action(this));
        }
        else if (_continuation is not null){
            throw new InvalidOperationException(
                        "Unlike Task, this implementation only
                       supports a single continuation.");
        }
        else {
            _continuation = action;
            _ec = ExecutionContext.Capture();
        }
    }
}
```

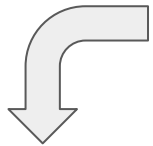


Здесь автоматически захватывается текущий контекст выполнения

Захватываем текущий контекст выполнения

Упрощенная версия Task

```
private void Complete(Exception? error){
    lock (this) //Пока считаем, что это mutex
    {
        if ( completed)
            throw new InvalidOperationException("Already completed");

         error = error;
        _completed = true;

        if (_continuation is null) return;

        ThreadPool.QueueUserWorkItem(_ =>
        {
            if ( ec is not null)
                ExecutionContext.Run(_ec, _ => _continuation(this), null);
            else
                _continuation(this);
        });
    }
}
```



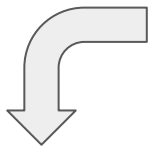
Используем сохраненный контекст выполнения

Упрощенная версия Task

```
private void SetResult() => Complete(null);

private void SetException(Exception error) => Complete(error);
```

Упрощенная версия Task

```
public void Wait()
{
    ManualResetEventSlim? mres = null; //что-то вроде std::conditional_variable
    lock (this)
    {
        if (!_completed)
        {
            mres = new ManualResetEventSlim();
            ContinueWith(_ => mres.Set());
        }
    }

    mres?.Wait();
    if (_error is not null)
        ExceptionDispatchInfo.Throw(_error);
}
```



Пробрасываем исключение дальше с исходным стеком вызовов

Упрощенная версия Task

```
public static MyTask Run(Action action)
{
    var t = new MyTask();

    ThreadPool.QueueUserWorkItem(_ =>
    {
        try
        {
            action();
            t.SetResult();
        }
        catch (Exception e)
        {
            t.SetException(e);
        }
    });

    return t;
}
```

Создание и запуск таски

Упрощенная версия Task

Комбинируем несколько задач в одну

```
public static MyTask WhenAll(MyTask t1, MyTask t2){
    var t = new MyTask();

    int remaining = 2;
    Exception? e = null;

    Action<MyTask> continuation = completed =>
    {
        e ??= completed. error; // для простоты сохраним только одно исключение
        if (Interlocked.Decrement(ref remaining) == 0) //Потокобезопасный декремент
        {
            if (e is not null) t.SetException(e);
            else t.SetResult();
        }
    };

    t1.ContinueWith(continuation);
    t2.ContinueWith(continuation);

    return t;
}
```

Упрощенная версия Task

```
var firstTask = MyTask.Run(ComputeSomething);
var secondTask = MyTask.Run(() => Console.WriteLine("Hello World!"));

var aggregatedTask = MyTask.WhenAll(firstTask, secondTask);
aggregatedTask.ContinueWith(_ => Console.WriteLine("All tasks completed!"));
aggregatedTask.Wait();
```

Упрощенная версия Task

```
var firstTask = MyTask.Run(ComputeSomething);
var secondTask = MyTask.Run(() => Console.WriteLine("Hello World!"));

var aggregatedTask = MyTask.WhenAll(firstTask, secondTask);
aggregatedTask.ContinueWith(_ => Console.WriteLine("All tasks completed!"));
aggregatedTask.Wait();
```

А как же async/await ?

Что мы знаем?

- async указывает, что метод является асинхронным. Такой метод может содержать await.
- await приостанавливает выполнение текущего метода до завершения асинхронной операция, не блокируя поток.
- `Task` представляет асинхронную операцию.
- Метод с async обычно возвращает `Task` или `Task<T>`.

async/await на пальцах

Когда используется await, компилятор разбивает метод на части:

1. До await:
    - код выполняется синхронно.
2. На await:
    - метод приостанавливается, и управление возвращается вызывающему коду.
3. После await:
    - Когда асинхронная операция завершается, выполнение метода продолжается с того места, где оно было приостановлено.

async/await на пальцах

```
static async Task<string> DownloadDataAsync(string url)
{
    Console.WriteLine("Начало загрузки данных");

    using (HttpClient client = new HttpClient())
    {
        // Асинхронный HTTP-запрос
        string result = await client.GetStringAsync(url);

        Console.WriteLine("Загрузка данных завершена");
        return result;
    }
}
```

async/await на пальцах

```
async Task Main(params string[] args)
{
    Console.WriteLine("Начало программы");

    // Асинхронный вызов
    string data = await DownloadDataAsync("https://example.com");

    Console.WriteLine($"Данные: {data}");
    Console.WriteLine("Конец программы");
}
```

Что происходит под капотом на пальцах

```
static Task Main(string[] args)
{
    var stateMachine = new MainStateMachine();
    stateMachine.MoveNext();
    return stateMachine.Task;
}
```

1. Компилятор преобразует метод с async/await в стейт-машину.
    - Это позволяет приостанавливать и возобновлять выполнение метода.
2. Код идущий после await'а добавляется как continuation к задаче.

Что происходит под капотом на пальцах

```
private class MainStateMachine
{
    private int state = 0;
    private Task<string> downloadTask;
    private string result;

    public Task Task { get; private set; }
    //...
}
```

продолжение на следующем слайде…

Что происходит под капотом на пальцах

```
public void MoveNext()
{
    switch (state)
    {
        case 0:
            Console.WriteLine("Начало программы");
            downloadTask = DownloadDataAsync("https://example.com");
            state = 1;
            downloadTask.ContinueWith(_ => MoveNext());
            break;

        case 1:
            result = downloadTask.Result;
            Console.WriteLine($"Данные: {result}");
            Console.WriteLine("Конец программы");
            state = -1;
            Task = Task.CompletedTask;
            break;
    }
}
```

Код после await'а добавили как продолжение

Тонкости async/await

Рассмотрим метод:

```
public async Task CopyStreamToStreamAsync (Stream source, Stream destination)
{
    var buffer = new byte[0x1000];
    int numRead;
    while ((numRead = await source.ReadAsync(buffer, 0, buffer.Length)) != 0)
    {
        await destination.WriteAsync(buffer, 0, numRead);
    }
}
```

Тонкости async/await

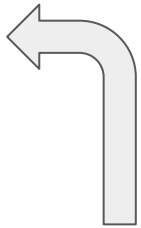
Преобразования компилятора:

```
[AsyncStateMachine(typeof(<CopyStreamToStreamAsync>d__0))]
public Task CopyStreamToStreamAsync(Stream source, Stream destination)
{
   <CopyStreamToStreamAsync>d__0 stateMachine = default;
   stateMachine.<>t__builder = AsyncTaskMethodBuilder.Create();
   stateMachine.source = source;
   stateMachine.destination = destination;
   stateMachine.<>1__state = -1;
   stateMachine.<>t__builder.Start(ref stateMachine);
   return stateMachine.<>t__builder.Task;
}
```

Тонкости async/await

Преобразования компилятора:

```
[AsyncStateMachine(typeof(<CopyStreamToStreamAsync>d__0))]
public Task CopyStreamToStreamAsync(Stream source, Stream destination)
{
    <CopyStreamToStreamAsync>d__0 stateMachine = default;
    stateMachine.<>t__builder = AsyncTaskMethodBuilder.Create();
    stateMachine.source = source;
    stateMachine.destination = destination;
    stateMachine.<>1__state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);
    return stateMachine.<>t__builder.Task;
}
```



Чтобы работал async/await с кастомными тасками надо реализовать свой AsyncTaskMethodBuilder и пометить свою таску атрибутом [AsyncMethodBuilder(typeof(MyTaskMethodBuilder))]

Тонкости async/await

Преобразования компилятора:

```
[AsyncStateMachine(typeof(<CopyStreamToStreamAsync>d__0))]
public Task CopyStreamToStreamAsync(Stream source, Stream destination)
{
    <CopyStreamToStreamAsync>d__0 stateMachine = default;
    stateMachine.<>t__builder = AsyncTaskMethodBuilder.Create();
    stateMachine.source = source;
    stateMachine.destination = destination;
    stateMachine.<>1__state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);
    return stateMachine.<>t__builder.Task;
}
```

В release <CopyStreamToStreamAsync>d_0 является struct'ом. В debug - class'ом.

Тонкости async/await

Преобразования компилятора:

```
[AsyncStateMachine(typeof(<CopyStreamToStreamAsync>d__0))]
public Task CopyStreamToStreamAsync(Stream source, Stream destination)
{
    <CopyStreamToStreamAsync>d__0 stateMachine = default;
    stateMachine.<>t__builder = AsyncTaskMethodBuilder.Create();
    stateMachine.source = source;
    stateMachine.destination = destination;
    stateMachine.<>1__state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);
    return stateMachine.<>t__builder.Task;
}
```

TaskBuilder может как создать новую задачу, если асинхронный метод приостанавливался, так и вернуть завершенную таску, если асинхронный метод выполнился синхронно.

Тонкости async/await

Преобразования компилятора:

```
[AsyncStateMachine(typeof(<CopyStreamToStreamAsync>d__0))]
public Task CopyStreamToStreamAsync(Stream source, Stream destination)
{
    <CopyStreamToStreamAsync>d__0 stateMachine = default;
    stateMachine.<>t__builder = AsyncTaskMethodBuilder.Create();
    stateMachine.source = source;
    stateMachine.destination = destination;
    stateMachine.<>1__state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);
    return stateMachine.<>t__builder.Task;
}
```

А что делает Start()?

Тонкости async/await

```
public void Start<TStateMachine>(ref TStateMachine stateMachine)
    where TStateMachine : IAsyncStateMachine
{
    ExecutionContext previous = Thread.CurrentThread._executionContext;
    try
    {
        stateMachine.MoveNext();
    }
    finally
    {
        ExecutionContext.Restore(previous);
    }
}
```

Восстанавливаем контекст текущего потока, чтобы предотвратить утечку данных из асинхронного метода в вызывающий

Тонкости async/await

Посмотрим немного на содержимое стейт-машины:

```
private struct <CopyStreamToStreamAsync>d__0 : IAsyncStateMachine
{
    public int <>1__state;
    public AsyncTaskMethodBuilder <>t__builder;
    public Stream source;
    public Stream destination;
    private byte[] <buffer>5__2;
    private TaskAwaiter <>u__1;
    private TaskAwaiter<int> <>u__2;

    //...
}
```

текущее состояние машины

Тонкости async/await

Посмотрим немного на содержимое стейт-машины:

```
private struct <CopyStreamToStreamAsync>d__0 : IAsyncStateMachine
{
    public int <>1__state;
    public AsyncTaskMethodBuilder <>t__builder;
    public Stream source;
    public Stream destination;
    private byte[] <buffer>5__2;
    private TaskAwaiter <>u__1;
    private TaskAwaiter<int> <>u__2;

    //...
}
```

Builder асинхронного метода, который возвращает Task. Если хотим поддерживать кастомные таски делаем свой builder.

Тонкости async/await

Посмотрим немного на содержимое стейт-машины:

```
private struct <CopyStreamToStreamAsync>d__0 : IAsyncStateMachine
{
    public int <>1__state;
    public AsyncTaskMethodBuilder <>t__builder;
    public Stream source;
    public Stream destination;
    private byte[] <buffer>5__2;
    private TaskAwaiter <>u__1;
    private TaskAwaiter<int> <>u__2;

    //...
}
```

Stream’ы - это аргументы асинхронного метода.

buffer - это локальная переменная асинхронного метода CopyStreamToStreamAsync.

Тонкости async/await

Посмотрим немного на содержимое стейт-машины:

```
private struct <CopyStreamToStreamAsync>d__0 : IAsyncStateMachine
{
    public int <>1__state;
    public AsyncTaskMethodBuilder <>t__builder;
    public Stream source;
    public Stream destination;
    private byte[] <buffer>5__2;
    private TaskAwaiter <>u__1;
    private TaskAwaiter<int> <>u__2;

    //...
}
```

Механизм, который управляет ожиданием Task в await.

TaskAwaiter

1. Получение TaskAwaiter:
    - Когда вы пишите await task, компилятор вызывает внутри стейт-машины GetAwaiter() у объекта task (duck typing требование).

TaskAwaiter

1. Получение `TaskAwaiter`.
2. Проверка завершения задачи:
    - Компилятор вызывает свойство `IsCompleted` у `TaskAwaiter`, чтобы проверить, завершена ли задача.
    - Если задача уже завершена, выполнение продолжается синхронно.

TaskAwaiter

1. Получение `TaskAwaiter`.
2. Проверка завершения задачи.
3. Приостановка выполнения:
    - Если задача не завершена, компилятор приостанавливает выполнение метода и регистрирует callback через метод `OnCompleted` у `TaskAwaiter`.
    - Когда задача завершается, callback вызывается, и выполнение метода возобновляется.

TaskAwaiter

1. Получение `TaskAwaiter`.
2. Проверка завершения задачи.
3. Приостановка выполнения.
4. Получение результата:
    - После завершения задачи компилятор вызывает метод `GetResult()` у `TaskAwaiter`, чтобы получить результат (если задача возвращает значение).

Пример своего TaskAwaiter

```
class MyTask{
    //...
    public MyTaskAwaiter GetAwaiter() => new MyTaskAwaiter { _task = this };

    public struct MyTaskAwaiter : INotifyCompletion    {
        internal MyTask _task;

        public bool IsCompleted => _task._completed;
        public void OnCompleted(Action continuation)
                            => _task.ContinueWith(_ => continuation());
        public void GetResult() => _task.Wait();
    }
}
```

Тонкости async/await

```
private void MoveNext()
{
    try { ... }
    catch (Exception exception)
    {
        <>1__state = -2;
        <buffer>5__2 = null;
        <>t__builder.SetException(exception);
        return;
    }

    <>1__state = -2;
    <buffer>5__2 = null;
    <>t__builder.SetResult();
}
```

MoveNext ответственный за перехват исключений

Тонкости async/await

```
private void MoveNext()
{
    try { ... }
    catch (Exception exception)
    {
        <>1__state = -2;
        <buffer>5__2 = null;
        <>t__builder.SetException(exception);
        return;
    }

    <>1__state = -2;
    <buffer>5__2 = null;
    <>t__builder.SetResult();
}
```

MoveNext ответственный за завершение задачи

Тонкости async/await

```
private void MoveNext()
{
    try { ... }
    catch (Exception exception)
    {
        <>1__state = -2;
        <buffer>5__2 = null;
        <>t__builder.SetException(exception);
        return;
    }

    <>1__state = -2;
    <buffer>5__2 = null;
    <>t__builder.SetResult();
}
```

Как же builder проставляет результат, если Task уже существует?

Тонкости async/await

```
private void MoveNext()
{
    try { ... }
    catch (Exception exception)
    {
        <>1__state = -2;
        <buffer>5__2 = null;
        <>t__builder.SetException(exception);
        return;
    }

    <>1__state = -2;
    <buffer>5__2 = null;
    <>t__builder.SetResult();
}
```

Как же builder проставляет результат, если Task уже существует?

Ответ:
через TaskCompletionSource

TaskCompletionSource

```
public Task<int> CreateTaskWithResultAsync()
{
    var tcs = new TaskCompletionSource<int>();

    // Симулируем асинхронную операцию
    Task.Delay(100).ContinueWith(_ =>
    {
        tcs.SetResult(42); // Проставляем результат
    });

    return tcs.Task;
}
```

TaskCompletionSource позволяет:

1. Создать задачу, которая не привязана к асинхронной операции.
2. Управлять состоянием задачи извне (завершить задачу с результатом, ошибкой или отменой).

SynchronizationContext в async/await

async/await учитывает текущий SynchronizationContext. Как это работает?

1. Захват SynchronizationContext перед выполнением асинхронной операции.
2. Возобновление (continuation) в исходном контексте.

Т.е. SynchronizationContext определяет, на каком потоке будет выполняться продолжение кода после await.

SynchronizationContext в async/await

```
public async Task UpdateUIAsync()
{
    // Присваивание текста будет выполняется в UI-потоке,
    // потому что SynchronizationContext был захвачен.
    button.Text = await Task.Run(() => ComputeMessage());
}
```

ConfigureAwait

- Это метод, который позволяет управлять поведением await в отношении `SynchronizationContext`.

ConfigureAwait

- Это метод, который позволяет управлять поведением await в отношении `SynchronizationContext`.
- Принимает булевый параметр `continueOnCapturedContext`.

ConfigureAwait

- Это метод, который позволяет управлять поведением await в отношении `SynchronizationContext`.
- Принимает булевый параметр `continueOnCapturedContext`.
- ConfigureAwait(true): продолжение будет выполнено на захваченном контексте. Это поведение по умолчанию.

ConfigureAwait

- Это метод, который позволяет управлять поведением await в отношении `SynchronizationContext`.
- Принимает булевый параметр `continueOnCapturedContext`.
- ConfigureAwait(true): продолжение будет выполнено на захваченном контексте. Это поведение по умолчанию.
- ConfigureAwait(false): продолжение может выполняться на любом свободном потоке. Позволяет повысить производительность.

SynchronizationContext в async/await

```
public async Task UpdateUIAsync()
{
    // Присваивание текста будет выполняется не в UI-потоке,
    // из-за ConfigureAwait(false). Это приведет к ошибке в runtime
    button.Text = await Task.Run(() =>
ComputeMessage()).ConfigureAwait(false);
}
```

SynchronizationContext в async/await

```
public async Task UpdateUIAsync()
{
    // Присваивание текста будет выполняется не в UI-потоке,
    // из-за ConfigureAwait(false). Это приведет к ошибке в runtime
    button.Text = await Task.Run(() =>
ComputeMessage()).ConfigureAwait(false);
}
```

Если нужно обновить UI после await - не используйте ConfigureAwait(false).

SynchronizationContext в async/await

```
public async Task UpdateUIAsync()
{
    // Присваивание текста будет выполняется не в UI-потоке,
    // из-за ConfigureAwait(false). Это приведет к ошибке в runtime
    button.Text = await Task.Run(() =>
ComputeMessage()).ConfigureAwait(false);
}
```

Если нужно обновить UI после await - не используйте ConfigureAwait(false).

ConfigureAwait(false) используем когда не важен контекст исполнения.

ValueTask

1. Структура (value type), предназначенная для снижения аллокаций при частых асинхронных вызовах.
2. При синхронной операции - хранит значение. При асинхронной - хранит либо `Task<T>` , либо `IValueTaskSource` (еще один способ оптимизировать код).
3. Оптимизирован для синхронных операций.
4. Нельзя несколько раз использовать await для одной ValueTask.
5. Не поддерживает методы `ContinueWith`, `WhenAll` и другие API `Task`.

ValueTask

1. Структура (value type), предназначенная для снижения аллокаций при частых асинхронных вызовах.
2. При синхронной операции - хранит значение. При асинхронной - хранит либо `Task<T>` , либо `IValueTaskSource` (еще один способ оптимизировать код).
3. Оптимизирован для синхронных операций.
4. Нельзя несколько раз использовать await для одной ValueTask.
5. Не поддерживает методы `ContinueWith`, `WhenAll` и другие API `Task`.

ValueTask

1. Структура (value type), предназначенная для снижения аллокаций при частых асинхронных вызовах.
2. При синхронной операции - хранит значение. При асинхронной - хранит либо `Task<T>` , либо `IValueTaskSource` (еще один способ оптимизировать код).
3. Оптимизирован для синхронных операций.
4. Нельзя несколько раз использовать await для одной ValueTask.
5. Не поддерживает методы `ContinueWith`, `WhenAll` и другие API `Task`.

ValueTask

1. Структура (value type), предназначенная для снижения аллокаций при частых асинхронных вызовах.
2. При синхронной операции - хранит значение. При асинхронной - хранит либо `Task<T>` , либо `IValueTaskSource` (еще один способ оптимизировать код).
3. Оптимизирован для синхронных операций.
4. Нельзя несколько раз использовать await для одной ValueTask.
5. Не поддерживает методы `ContinueWith`, `WhenAll` и другие API `Task`.

ValueTask

1. Структура (value type), предназначенная для снижения аллокаций при частых асинхронных вызовах.
2. При синхронной операции - хранит значение. При асинхронной - хранит либо `Task<T>` , либо `IValueTaskSource` (еще один способ оптимизировать код).
3. Оптимизирован для синхронных операций.
4. Нельзя несколько раз использовать await для одной ValueTask.
5. Не поддерживает методы `ContinueWith`, `WhenAll` и другие API `Task`.

Отмена асинхронной операции

1. `CancellationTokenSource`:
    - Создает и управляет `CancellationToken`.
    - Вызывает отмену через метод `Cancel()`.
2. `CancellationToken`:
    - Передается в метод, которые поддерживает отмену.
    - Проверяется на наличие запроса отмены.

Передача CancellationToken в метод

```
public async Task DoWorkAsync(CancellationToken cancellationToken)
{
    for (int i = 0; i < 10; i++)
    {
        cancellationToken.ThrowIfCancellationRequested(); // Проверка отмены
        await Task.Delay(1000, cancellationToken); // Асинхронная операция с
                                                   поддержкой отмены
        Console.WriteLine($"Working... {i}");
    }
}
```

Запуск отмены

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

// Запуск задачи
var task = Task.Run(() => DoWorkAsync(token));

// Отмена через 3 секунды
await Task.Delay(3000);
cts.Cancel();

try{
   await task;
}
catch (OperationCanceledException){
   Console.WriteLine("Operation was canceled.");
}
```

Best practices

Не используйте async void, вместо этого используйте async Task.

Best practices

Не используйте async void, вместо этого используйте async Task:

- async void нельзя await’ить => нельзя узнать когда метод завершился.

Best practices

Не используйте async void, вместо этого используйте async Task:

- async void нельзя await’ить => нельзя узнать когда метод завершился.
- исключения из async void нельзя перехватить.

Best practices

Не используйте async void, вместо этого используйте async Task:

- async void нельзя await'ить => нельзя узнать когда метод завершился.
- исключения из async void нельзя перехватить.
- async void можно использовать в обработчиках событий, где сигнатура метода должна быть void.

Best practices

```
async void DoSomethingAsync()
{
    throw new Exception("Oops!");
}

void CallAsync()
{
    try
    {
        DoSomethingAsync(); // Исключение не будет поймано здесь.
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message); // Этот блок не сработает.
    }
}
```

Best practices

Избегайте блокирующих вызовов в асинхронном коде, используйте await.

```
Task<int> task = GetDataAsync();


int result = task.Result; // ❌ Возможна блокировка!
task.Wait(); // ❌ Возможна блокировка!
task.GetAwaiter().GetResult(); // ❌ Возможна блокировка!
```

Best practices

Избегайте блокирующих вызовов в асинхронном коде, используйте await.

```
Task<int> task = GetDataAsync();


int result = task.Result; // ❌ Возможна блокировка!
task.Wait(); // ❌ Возможна блокировка!
task.GetAwaiter().GetResult(); // ❌ Возможна блокировка!
```

Проблемы:

1. Может привести к deadlock в UI приложениях.
2. Блокирует поток, не давая другим async-операциям выполняться.
3. Если метод выбрасывает исключение, то оно обернуто в AggregateException.

Best practices

Используйте ConfigureAwait(false) в местах, где код не зависит от конкретного контекста выполнения.

Best practices

Используйте ValueTask для методов, которые часто завершаются синхронно.

```
public ValueTask<int> GetCachedValueAsync()
{
    if (cache.TryGetValue("key", out int value))
    {
        return new ValueTask<int>(value); // ✅ Без аллокации Task!
    }
    return GetFromDatabaseAsync();
}
```

Best practices

Избегайте излишнего асинхронного кода. Не все методы должны быть асинхронными.

```
// Плохо:
public async Task<int> AddAsync(int a, int b)
{
    return await Task.FromResult(a + b); // Излишний асинхронный код
}

// Хорошо:
public int Add(int a, int b)
{
    return a + b; // Простой синхронный метод
}
```

Best practices

Не используйте await в Parallel.ForEach().

```
var numbers = Enumerable.Range(1, 10);
Parallel.ForEach(numbers, async number =>
{
    await Task.Delay(100); // Ошибка: await внутри Parallel.ForEach
    Console.WriteLine($"Processed {number}");
});
```

В этом примере `Parallel.ForEach` завершится до того, как все асинхронные операции (`await Task.Delay`) будут выполнены, так как он не ожидает завершения await.

Best practices

Не используйте await в Parallel.ForEach().

```
var numbers = Enumerable.Range(1, 10);
Parallel.ForEach(numbers, async number =>
{
    await Task.Delay(100); // Ошибка: await внутри Parallel.ForEach
    Console.WriteLine($"Processed {number}");
});
```

В этом примере `Parallel.ForEach` завершится до того, как все асинхронные операции (`await Task.Delay`) будут выполнены, так как он не ожидает завершения await.

Но можно в этом случае использовать Parallel.ForEachAsync()

Best practices

Используйте `Task.WhenAll` для параллельного выполнения задач.

```
public async Task ProcessDataAsync()
{
    Task<int> task1 = GetDataAsync();
    Task<int> task2 = GetMoreDataAsync();

    int[] results = await Task.WhenAll(task1, task2);
}
```

Примитивы синхронизации

Monitor

Обеспечивает эксклюзивный доступ к блоку кода для одного потока.

- `Monitor.Enter(object obj)`:
  - Получение эксклюзивного доступа к объекту.
  - Потоки, пытающиеся захватить блокировку на этом объекте, будут заблокированы, пока первый поток не освободит блокировку.

Monitor

Обеспечивает эксклюзивный доступ к блоку кода для одного потока.

- `Monitor.Enter(object obj)`.
- `Monitor.Enter(object obj, ref bool lockTaken)`:
    - То же самое, только в `lockTaken` проставляется `true`, если захват блокировки удалось выполнить.

Monitor

Обеспечивает эксклюзивный доступ к блоку кода для одного потока.

- `Monitor.Enter(object obj)`.
- `Monitor.Enter(object obj, ref bool lockTaken)`.
- `Monitor.TryEnter(object obj)`:
  - Пытается захватить блокировку.
  - Возвращает true, если удалось, и false, если нет.

Monitor

Обеспечивает эксклюзивный доступ к блоку кода для одного потока.

- `Monitor.Enter(object obj)`.
- `Monitor.Enter(object obj, ref bool lockTaken)`.
- `Monitor.TryEnter(object obj)`.
- `Monitor.Exit(object obj)`:
    - Освобождает блокировку, которую захватил текущий поток через `Monitor.Enter`.

Monitor

Обеспечивает эксклюзивный доступ к блоку кода для одного потока.

- `Monitor.Enter(object obj)`.
- `Monitor.Enter(object obj, ref bool lockTaken)`.
- `Monitor.TryEnter(object obj)`.
- `Monitor.Exit(object obj)`.
- `Monitor.Wait(object obj)`:
    - Освобождает захваченную блокировку и ждет пока его не уведомят.
    - Может быть Spurious wakeup. Поэтому надо оборачивать в цикл.
    - Используется только внутри `Monitor.Enter`, иначе будет исключение.

Monitor

Обеспечивает эксклюзивный доступ к блоку кода для одного потока.

- `Monitor.Enter(object obj)`.
- `Monitor.Enter(object obj, ref bool lockTaken)`.
- `Monitor.TryEnter(object obj)`.
- `Monitor.Exit(object obj)`.
- `Monitor.Wait(object obj)`.
- `Monitor.Pulse(object obj)`:
    - Сигнализирует одному потоку, который ожидает на объекте, что можно продолжать выполнение.
    - Используется только внутри `Monitor.Enter`.

Monitor

Обеспечивает эксклюзивный доступ к блоку кода для одного потока.

- `Monitor.Enter(object obj)`.
- `Monitor.Enter(object obj, ref bool lockTaken)`.
- `Monitor.TryEnter(object obj)`.
- `Monitor.Exit(object obj)`.
- `Monitor.Wait(object obj)`.
- `Monitor.Pulse(object obj)`.
- `Monitor.PulseAll(object obj)`:
    - Сигнализирует всем потокам, которые ожидают на объекте, что можно продолжать выполнение.

Monitor

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

// Запускаем два таска, которые будут инкрементировать counter
Task task1 = Task.Run(IncrementCounter);
Task task2 = Task.Run(IncrementCounter);

// Ожидаем завершения тасков
await Task.WhenAll(task1, task2);

// Выводим результат
Console.WriteLine($"Final counter value: {counter}");
```

Monitor

```
void IncrementCounter(){
    Monitor.Enter(lockObject);
    try
    {
        counter++;
        Console.WriteLine($"Counter: {counter} (Task: {Task.CurrentId})");
    }
    finally
    {
        Monitor.Exit(lockObject);
    }
}
```

LockTaken pattern

```
void IncrementCounter()
{
    bool lockTaken = false;
    try
    {
        Monitor.Enter(lockObject, ref lockTaken);
        counter++;
        Console.WriteLine($"Counter: {counter} (Task: {Task.CurrentId})");
    }
    finally
    {
        // Если блокировка была захвачена, освобождаем её
        if (lockTaken) Monitor.Exit(lockObject);
    }
}
```

Monitor.Wait and Monitor.Pulse

```
object lockObject = new object();
bool isReady = false;

// Запуск двух задач, которые будут взаимодействовать
Task task1 = Task.Run(WaitForSignal);
Task task2 = Task.Run(SendSignal);

// Ожидаем завершения задач
await Task.WhenAll(task1, task2);

Console.WriteLine("Program finished.");
```

Monitor.Wait and Monitor.Pulse

```
// Задача, которая будет ожидать сигнал
void WaitForSignal(){
   bool lockTaken = false;
   try   {
       Monitor.Enter(lockObject, ref lockTaken);

       while (!isReady)       {
           Console.WriteLine("Task 1 waiting for signal...");
           Monitor.Wait(lockObject); // Ожидает, пока другой поток не вызовет Pulse
           // Продолжим выполнение только после того как другой поток выполнит
Monitor.Exit
       }

       Console.WriteLine("Task 1 received signal and is continuing...");
   }
   finally   {
       if (lockTaken) Monitor.Exit(lockObject);
   }
}
```

Monitor.Wait and Monitor.Pulse

```
// Задача, которая отправляет сигнал
void SendSignal(){
    Thread.Sleep(2000); // Задержка перед отправкой сигнала

    bool lockTaken = false;
    try    {
        // Захват блокировки с использованием Monitor.Enter
        Monitor.Enter(lockObject, ref lockTaken);

        // Устанавливаем флаг, что поток готов продолжить
        isReady = true;
        Console.WriteLine("Task 2 sending signal...");

        // Пробуждаем один поток, ожидающий на Monitor.Wait
        Monitor.Pulse(lockObject);
    }
    finally    {
        if (lockTaken) Monitor.Exit(lockObject);
    }
}
```

Monitor

Monitor поддерживает рекурсивные блокировки одного и того же объекта.

```
void RecursiveMethod(int level){
    // Захватываем блокировку рекурсивно
    Monitor.Enter(_lock);
    Console.WriteLine($"RecursiveMethod: Lock acquired (level{level}).");

    if (level > 0)    {
        // Рекурсивный вызов
        RecursiveMethod(level - 1);
    }

    // Освобождаем блокировку
    Monitor.Exit(_lock);
    Console.WriteLine($"RecursiveMethod: Lock released (level{level}).");
}
```

Monitor

Нельзя использовать async/await внутри `Monitor`.

Monitor

Нельзя использовать async/await внутри Monitor.

1. Блокировка потока.
    - Monitor удерживает блокировку до вызова Monitor.Exit, но await может приостановить выполнение и возобновить на другом потоке, который не удерживает блокировку.

Monitor

Нельзя использовать async/await внутри Monitor.

1. Блокировка потока.
    - Monitor удерживает блокировку до вызова Monitor.Exit, но await может приостановить выполнение и возобновить на другом потоке, который не удерживает блокировку.
2. SynchronizationLockException.
    - await переключил поток и Monitor.Exit выполнился на другом потоке.

Monitor

Плохая практика использовать this в качестве объекта для блокировки.

1. Внешний код может использовать тот же объект для блокировки.
2. this в структурах будет бокситься.
3. При наследовании будет использоваться один и тот же lockObject.

lock

- Синтаксический сахар.
- Обеспечивает эксклюзивный доступ к блоку кода для одного потока.

```
object lockObject = new object();

lock (lockObject)
{
    // критическая секция
}
```

lock

- Синтаксический сахар.
- Обеспечивает эксклюзивный доступ к блоку кода для одного потока.

```
object lockObject = new object();

lock (lockObject)
{
    // критическая секция
}
```

```
object lockObject = new object();

bool lockTaken = false;
try
{
    Monitor.Enter(lockObject, ref lockTaken);
    // критическая секция
}
finally
{
    if (lockTaken) Monitor.Exit(lockObject);
}
```

Mutex

- Предназначен для взаимного исключения при доступе к разделяемым ресурсам.
- Позволяет делать синхронизацию не только в рамках одного процесса, но и между процессами.
- В рамках одного процесса лучше использовать lock или другие примитивы синхронизации.
- Mutex работает медленнее, чем lock, т.к. взаимодействует с ядром ОС.

Mutex

- Предназначен для взаимного исключения при доступе к разделяемым ресурсам.
- Позволяет делать синхронизацию не только в рамках одного процесса, но и между процессами.
- В рамках одного процесса лучше использовать lock или другие примитивы синхронизации.
- Mutex работает медленнее, чем lock, т.к. взаимодействует с ядром ОС.

Mutex

- Предназначен для взаимного исключения при доступе к разделяемым ресурсам.
- Позволяет делать синхронизацию не только в рамках одного процесса, но и между процессами.
- В рамках одного процесса лучше использовать lock или другие примитивы синхронизации.
- Mutex работает медленнее, чем lock, т.к. взаимодействует с ядром ОС.

Mutex

- Предназначен для взаимного исключения при доступе к разделяемым ресурсам.
- Позволяет делать синхронизацию не только в рамках одного процесса, но и между процессами.
- В рамках одного процесса лучше использовать lock или другие примитивы синхронизации.
- Mutex работает медленнее, чем lock, т.к. взаимодействует с ядром ОС.

Mutex

```
string mutexName = "Global\\MyUniqueMutex"; // 🔹 Глобальное имя для межпроцессной синхронизации

using (Mutex mutex = new Mutex(false, mutexName, out bool createdNew)){
    if (!createdNew)    {
        Console.WriteLine("Другой процесс уже использует этот ресурс. Ожидание...");
    }

    mutex.WaitOne(); // 🔴 Захватываем Mutex

    try    {
        Console.WriteLine("Ресурс захвачен! Работаем...");
        Thread.Sleep(5000); // Имитация работы с ресурсом
    }
    finally    {
        Console.WriteLine("Ресурс освобожден");
        mutex.ReleaseMutex(); // ✅ Освобождаем Mutex
    }
}
```

SpinLock

- Предназначен для взаимного исключения при доступе к разделяемым ресурсам.
- Занимает процессорное время вместо блокировки потока.
  - lock и Mutex ставят поток в ожидание;
  - SpinLock не переводит поток в ожидание, а крутится в цикле, проверяя доступность блокировки.
- Нужен для минимизации переключения контекста.
- Ожидаемая блокировка должна быть короткой, иначе напрасно будем тратить процессорное время.
- Не поддерживает рекурсивную блокировку.

SpinLock

- Предназначен для взаимного исключения при доступе к разделяемым ресурсам.
- Занимает процессорное время вместо блокировки потока.
  - lock и Mutex ставят поток в ожидание;
  - SpinLock не переводит поток в ожидание, а крутится в цикле, проверяя доступность блокировки.
- Нужен для минимизации переключения контекста.
- Ожидаемая блокировка должна быть короткой, иначе напрасно будем тратить процессорное время.
- Не поддерживает рекурсивную блокировку.

SpinLock

- Предназначен для взаимного исключения при доступе к разделяемым ресурсам.
- Занимает процессорное время вместо блокировки потока.
  - lock и Mutex ставят поток в ожидание;
  - SpinLock не переводит поток в ожидание, а крутится в цикле, проверяя доступность блокировки.
- Нужен для минимизации переключения контекста.
- Ожидаемая блокировка должна быть короткой, иначе напрасно будем тратить процессорное время.
- Не поддерживает рекурсивную блокировку.

SpinLock

- Предназначен для взаимного исключения при доступе к разделяемым ресурсам.
- Занимает процессорное время вместо блокировки потока.
  - lock и Mutex ставят поток в ожидание;
  - SpinLock не переводит поток в ожидание, а крутится в цикле, проверяя доступность блокировки.
- Нужен для минимизации переключения контекста.
- Ожидаемая блокировка должна быть короткой, иначе напрасно будем тратить процессорное время.
- Не поддерживает рекурсивную блокировку.

SpinLock

- Предназначен для взаимного исключения при доступе к разделяемым ресурсам.
- Занимает процессорное время вместо блокировки потока.
  - lock и Mutex ставят поток в ожидание;
  - SpinLock не переводит поток в ожидание, а крутится в цикле, проверяя доступность блокировки.
- Нужен для минимизации переключения контекста.
- Ожидаемая блокировка должна быть короткой, иначе напрасно будем тратить процессорное время.
- Не поддерживает рекурсивную блокировку.

SpinLock

```
SpinLock spinLock = new SpinLock();
int counter = 0;

void Increment(){
   bool lockTaken = false;
   try   {
       spinLock.Enter(ref lockTaken);
       counter++;
   }
   finally   {
       if (lockTaken)
           spinLock.Exit();
   }
}
```

ReaderWriterLockSlim

- Позволяет одновременное чтение нескольким потокам.
- Блокирует доступ для всех, если кто-то выполняет запись.
- Эффективнее `ReaderWriterLock` из-за более легковесных механизмов синхронизации.
- По умолчанию не поддерживает рекурсивность (но ее можно включить).
- Поддерживает апгрейд блокировки с чтения на запись.
- Реализует политику fairness.

ReaderWriterLockSlim

- Позволяет одновременное чтение нескольким потокам.
- Блокирует доступ для всех, если кто-то выполняет запись.
- Эффективнее `ReaderWriterLock` из-за более легковесных механизмов синхронизации.
- По умолчанию не поддерживает рекурсивность (но ее можно включить).
- Поддерживает апгрейд блокировки с чтения на запись.
- Реализует политику fairness.

ReaderWriterLockSlim

- Позволяет одновременное чтение нескольким потокам.
- Блокирует доступ для всех, если кто-то выполняет запись.
- Эффективнее `ReaderWriterLock` из-за более легковесных механизмов синхронизации.
- По умолчанию не поддерживает рекурсивность (но ее можно включить).
- Поддерживает апгрейд блокировки с чтения на запись.
- Реализует политику fairness.

ReaderWriterLockSlim

- Позволяет одновременное чтение нескольким потокам.
- Блокирует доступ для всех, если кто-то выполняет запись.
- Эффективнее `ReaderWriterLock` из-за более легковесных механизмов синхронизации.
- По умолчанию не поддерживает рекурсивность (но ее можно включить).
- Поддерживает апгрейд блокировки с чтения на запись.
- Реализует политику fairness.

ReaderWriterLockSlim

- Позволяет одновременное чтение нескольким потокам.
- Блокирует доступ для всех, если кто-то выполняет запись.
- Эффективнее `ReaderWriterLock` из-за более легковесных механизмов синхронизации.
- По умолчанию не поддерживает рекурсивность (но ее можно включить).
- Поддерживает апгрейд блокировки с чтения на запись.
- Реализует политику fairness.

ReaderWriterLockSlim

- Позволяет одновременное чтение нескольким потокам.
- Блокирует доступ для всех, если кто-то выполняет запись.
- Эффективнее `ReaderWriterLock` из-за более легковесных механизмов синхронизации.
- По умолчанию не поддерживает рекурсивность (но ее можно включить).
- Поддерживает апгрейд блокировки с чтения на запись.
- Реализует политику fairness.

ReaderWriterLockSlim

```
ReaderWriterLockSlim  rwLock = new ReaderWriterLockSlim ();
List<int> data = new List<int>();

void ReadData(){
    rwLock.EnterReadLock();
    try    {
        Console.WriteLine($"Чтение: {string.Join(", ", data)}");
    }
    finally { rwLock.ExitReadLock(); }
}

void WriteData(int value){
    rwLock.EnterWriteLock();
    try    {
        data.Add(value);
        Console.WriteLine($"Запись: {value}");
    }
    finally { rwLock.ExitWriteLock(); }
}
```

ReaderWriterLockSlim

```
void ReadAndMaybeWrite(int newValue)
{
    rwLock.EnterUpgradeableReadLock();
    try
    {
        if (!data.Contains(newValue)) // Читаем данные
        {
            rwLock.EnterWriteLock();
            try
            {
                data.Add(newValue);
                Console.WriteLine($"Добавлено: {newValue}");
            }
            finally { rwLock.ExitWriteLock(); }
        }
    }
    finally { rwLock.ExitUpgradeableReadLock(); }
}
```

SemaphoreSlim

- Позволяет задать максимальное количество потоков, которые могут одновременно получить доступ к ресурсу.
- Поддерживает асинхронность.

SemaphoreSlim

- Позволяет задать максимальное количество потоков, которые могут одновременно получить доступ к ресурсу.
- Поддерживает асинхронность.

SemaphoreSlim

Когда использовать?

- Ограничение параллельного доступа:
    - Если есть пул ресурсов и вы хотите ограничить количество одновременных запросов.

SemaphoreSlim

Когда использовать?

- Ограничение параллельного доступа.
- Предотвращение “перегрузки” системы:
    - Можно ограничить количество одновременных запросов, чтобы избежать перегрузки сети или сервера.

SemaphoreSlim

Когда использовать?

- Ограничение параллельного доступа.
- Предотвращение “перегрузки” системы.
- Асинхронные операции.

SemaphoreSlim

```
SemaphoreSlim semaphore = new SemaphoreSlim(2); // Разрешаем 2 потока
одновременно

async Task ProcessAsync(int id){
    Console.WriteLine($"Поток {id} ждет...");
    await semaphore.WaitAsync(); // Захватываем семафор
    try    {
        Console.WriteLine($"Поток {id} выполняет работу...");
        await Task.Delay(2000); // Имитация работы
    }
    finally     {
        Console.WriteLine($"Поток {id} завершил работу.");
        semaphore.Release(); // Освобождаем слот
    }
}
```

AutoResetEvent

Примитив синхронизации, который позволяет потокам взаимодействовать друг с другом через сигналы.

AutoResetEvent

Примитив синхронизации, который позволяет потокам взаимодействовать друг с другом через сигналы.

Имеет два состояния:

1. Несигнальное (false) → потоки блокируются при вызове `WaitOne()`.
2. Сигнальное (true) → разблокируется **один поток**, после чего автоматически сбрасывается в false. Сигналом служит вызов метода `Set()`.

AutoResetEvent

Примитив синхронизации, который позволяет потокам взаимодействовать друг с другом через сигналы.

Имеет два состояния:

1. Несигнальное (false) → потоки блокируются при вызове `WaitOne()`.
2. Сигнальное (true) → разблокируется **один поток**, после чего автоматически сбрасывается в false. Сигналом служит вызов метода `Set()`.

AutoResetEvent

Зачем нужен?

1. Ожидание завершения задачи:
    - Один поток может ожидать сигнала от другого потока, который выполняет какую-то работу.

AutoResetEvent

Зачем нужен?

1. Ожидание завершения задачи.
2. Координация потоков:
    - Поток A должен дождаться, пока поток B выполнит определенные действия, прежде чем продолжить.

AutoResetEvent

Зачем нужен?

1. Ожидание завершения задачи.
2. Координация потоков.
3. Ограничение доступа к ресурсу:
    - AutoResetEvent может использоваться для управления доступом к ресурсу, который может быть использован только одним потоком одновременно.

AutoResetEvent

Основные методы:

- `Set()`:
    - Переводит событие в сигнальное состояние. Если есть ожидающие потоки, один из них будет разблокирован, после чего событие автоматически сбросится в несигнальное состояние.

AutoResetEvent

Основные методы:

- `Set()`.
- `WaitOne()`:
    - Блокирует текущий поток до тех пор, пока событие не перейдет в сигнальное состояние. После получения сигнала событие автоматически сбрасывается.

AutoResetEvent

Основные методы:

- `Set()`.
- `WaitOne()`.
- `Reset()`:
    - Принудительно переводит событие в несигнальное состояние.

AutoResetEvent

```
AutoResetEvent autoResetEvent = new AutoResetEvent(false);

void DoWork(){
   Console.WriteLine("Рабочий поток выполняет работу...");
   Thread.Sleep(2000); // Имитация работы

   Console.WriteLine("Рабочий поток отправляет сигнал.");
   autoResetEvent.Set(); // Сигнал основному потоку
}

Console.WriteLine("Основной поток запущен.");

// Запуск рабочего потока
Thread workerThread = new Thread(DoWork);
workerThread.Start();

// Ожидание сигнала от рабочего потока
Console.WriteLine("Основной поток ожидает сигнала...");
autoResetEvent.WaitOne(); // Блокировка до получения сигнала

Console.WriteLine("Основной поток получил сигнал и завершает работу.");
```

ManualResetEvent

- Разблокирует все ожидающие потоки при вызове `Set()`.

ManualResetEvent

- Разблокирует все ожидающие потоки при вызове `Set()`.
- Остаётся в сигнальном (true) состоянии, пока не будет вручную сброшен `Reset()`.

ManualResetEvent

- Разблокирует все ожидающие потоки при вызове `Set()`.
- Остаётся в сигнальном (true) состоянии, пока не будет вручную сброшен `Reset()`.
- Используется для массового пробуждения потоков и управления их выполнением.

ManualResetEvent

- Разблокирует все ожидающие потоки при вызове `Set()`.
- Остаётся в сигнальном (true) состоянии, пока не будет вручную сброшен `Reset()`.
- Используется для массового пробуждения потоков и управления их выполнением.

В отличие от `AutoResetEvent`, разблокирует всех сразу и не сбрасывается автоматически.

ManualResetEvent

Когда использовать?

- Разблокировка нескольких потоков.
- Долговременное сигнальное состояние.

ManualResetEventSlim

Оптимизированная версия `ManualResetEvent`, которая:

- Работает быстрее за счет активного ожидания (spin-wait), т.е. без блокировки.
- Подходит для кратковременных ожиданий.
- Работает в двух режимах:
  - Активное ожидание в течении небольшого времени.
  - Ожидание через ядро ОС, если ожидание долгое.

ManualResetEventSlim

Оптимизированная версия `ManualResetEvent`, которая:

- Работает быстрее за счет активного ожидания (spin-wait), т.е. без блокировки.
- Подходит для кратковременных ожиданий.
- Работает в двух режимах:
  - Активное ожидание в течении небольшого времени.
  - Ожидание через ядро ОС, если ожидание долгое.

ManualResetEventSlim

Оптимизированная версия `ManualResetEvent`, которая:

- Работает быстрее за счет активного ожидания (spin-wait), т.е. без блокировки.
- Подходит для кратковременных ожиданий.
- Работает в двух режимах:
  - Активное ожидание в течении небольшого времени.
  - Ожидание через ядро ОС, если ожидание долгое.

Interlocked

1. `Increment` / `Decrement`: атомарно увеличивает или уменьшает значение переменной на 1.

```
int counter = 0;
Interlocked.Increment(ref counter); // counter = 1
Interlocked.Decrement(ref counter); // counter = 0
```

Interlocked

1. `Increment` / `Decrement`.
2. `Add`: атомарно добавляет значение к переменной.

```
int value = 10;
Interlocked.Add(ref value, 5); // value = 15
```

Interlocked

1. `Increment` / `Decrement`.
2. `Add`.
3. `Exchange`: атомарно заменяет значение переменной на новое и возвращает старое значение.

```
int value = 10;
int oldValue = Interlocked.Exchange(ref value, 20); // value = 20, oldValue = 10
```

Interlocked

1. Increment / Decrement.
2. Add.
3. Exchange.
4. CompareExchange: атомарно сравнивает значение переменной с ожидаемым значением и, если они равны, заменяет его на новое значение. Возвращает исходное значение переменной.

```
int value = 10;
int oldValue = Interlocked.CompareExchange(ref value, value: 20, comparand: 10);
// value = 20, oldValue = 10
```

Interlocked

1. Increment / Decrement.
2. Add.
3. Exchange.
4. CompareExchange.
5. Read: атомарно читает значение 64-битной переменной.

```
long value = 100;
long result = Interlocked.Read(ref value); // result = 100
```

Interlocked

Преимущества:

- **Высокая производительность:**
  - Атомарные операции выполняются на уровне процессора и не требуют блокировок, что делает их очень быстрыми.

Interlocked

Преимущества:

- **Высокая производительность.**
- **Простота использования:**
  - Не нужно явно использовать блокировки (например, lock), что упрощает код.

Interlocked

Преимущества:

- **Высокая производительность.**
- **Простота использования.**
- **Минимизация deadlock:**
    - Поскольку атомарные операции не используют блокировки, они исключают возможность deadlock.

Interlocked

Ограничения:

- **Ограниченный набор операций:**
  - `Interlocked` поддерживает только простые операции (инкремент, декремент, замена, сравнение и т.д.). Для более сложных сценариев могут потребоваться другие механизмы синхронизации.

Interlocked

Ограничения:

- **Ограниченный набор операций.**
- **Только для примитивных типов:**
  - `Interlocked` работает только с примитивными типами (`int`, `long`, `float`, `double`) и ссылками на объекты.

Interlocked

Ограничения:

- **Ограниченный набор операций.**
- **Только для примитивных типов.**
- **Не подходит для сложных сценариев:**
  - Если требуется синхронизация сложных структур данных или операций, лучше использовать `lock`, `Monitor` или другие примитивы.