C# Асинхронное программирование

Асинхронное программирование с async await

C# Асинхронное программирование

Автор курса

Гнатюк Владислав

MCID:16354168

Асинхронное программирование

После урока обязательно

Повторите этот урок в видео формате на ITVDN.com

Проверьте как Вы усвоили данный материал на TestProvider.com

Асинхронное программирование с async await

План урока

1) Асинхронные операции
2) Асинхронные CPU операции
3) Асинхронные операции ввода-вывода
4) Асинхронность
5) Асинхронные шаблоны программирования
6) Создание асинхронных операций
7) Ограничения в использовании ключевых слов async await

Асинхронные операции

Асинхронность может применятся в следующих случаях:

- Операции CPU
  - Параллельное/неблокирующее/фоновое выполнение
  - Распараллеливание операции

- Операции ввода-вывода
  - Работа с файловой системой
  - Работа с сетью
  - Работа с удаленной базой данных
  - Работа с удаленными веб-сервисами

Асинхронные CPU операции

CPU операции – это операции, которые выполняются ресурсами центрального процессора.
Эти операции представлены обыкновенными синхронными методами, которые иногда необходимо вызывать асинхронно.

Причины вызова таких методов асинхронно, в контексте вторичного потока:

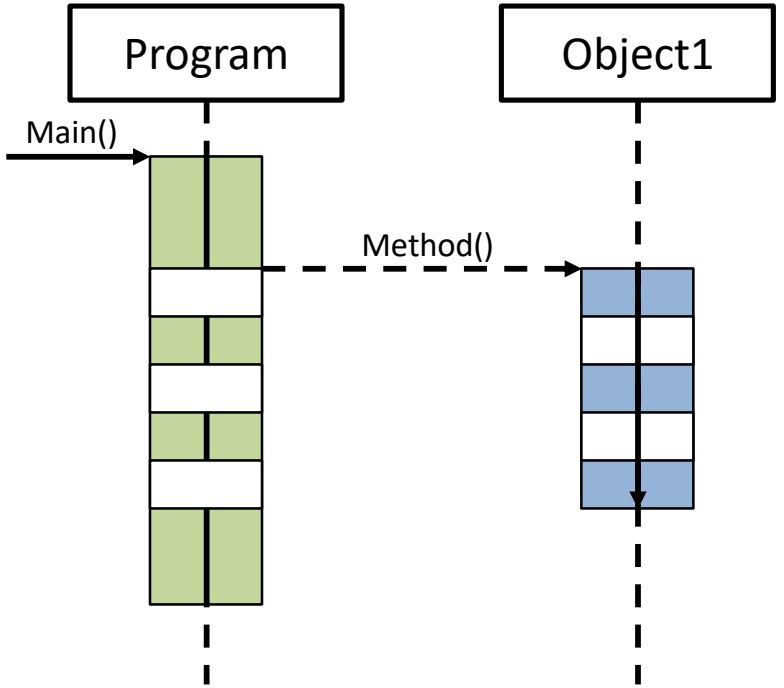
- Блокирование основного потока на время своего выполнения
- Фоновое выполнение
- Параллельное выполнение

Для запуска синхронного метода асинхронно необходимо воспользоваться статическим методом Task.**Run**().

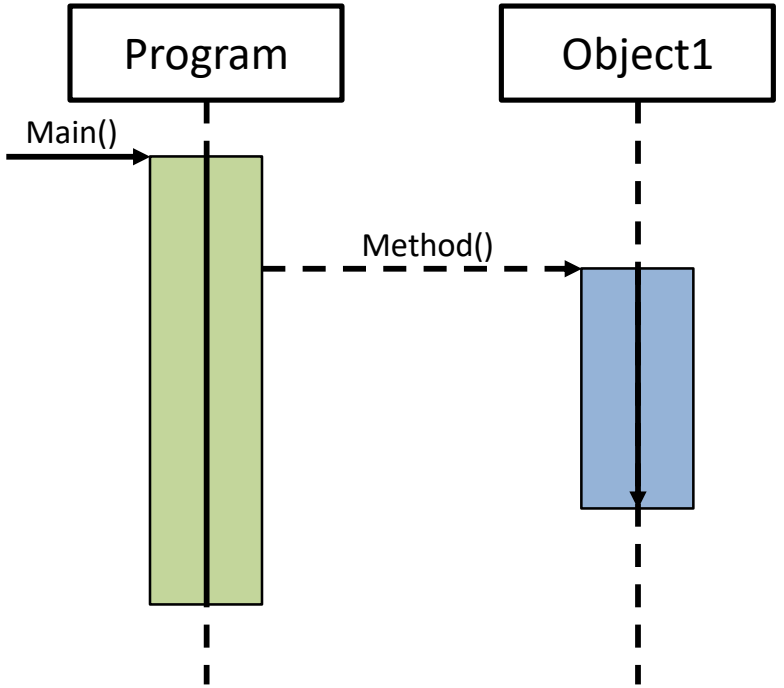
Асинхронные CPU операции

Асинхронные CPU операции используют многопоточность, чтобы в контексте вторичных потоков выполнять необходимые операции, не блокируя основной/вызывающий поток. Такие операции зависят от ресурсов центрального процессора.

Асинхронный вызов метода (CPU 1 kernel)



Асинхронный вызов метода (CPU 2 kernel's)



Операции ввода-вывода

Операции ввода-вывода – это операции передачи/получения сигнала (данных) между нашим приложением и аппаратным обеспечением. Таким образом происходит обмен информацией с аппаратными частями компьютера (жесткий диск, сетевой адаптер…).

ОС Windows запрещает напрямую «общаться» с аппаратными устройствами. Необходимо отправлять сигналы через специальные открытые API операционной системы.

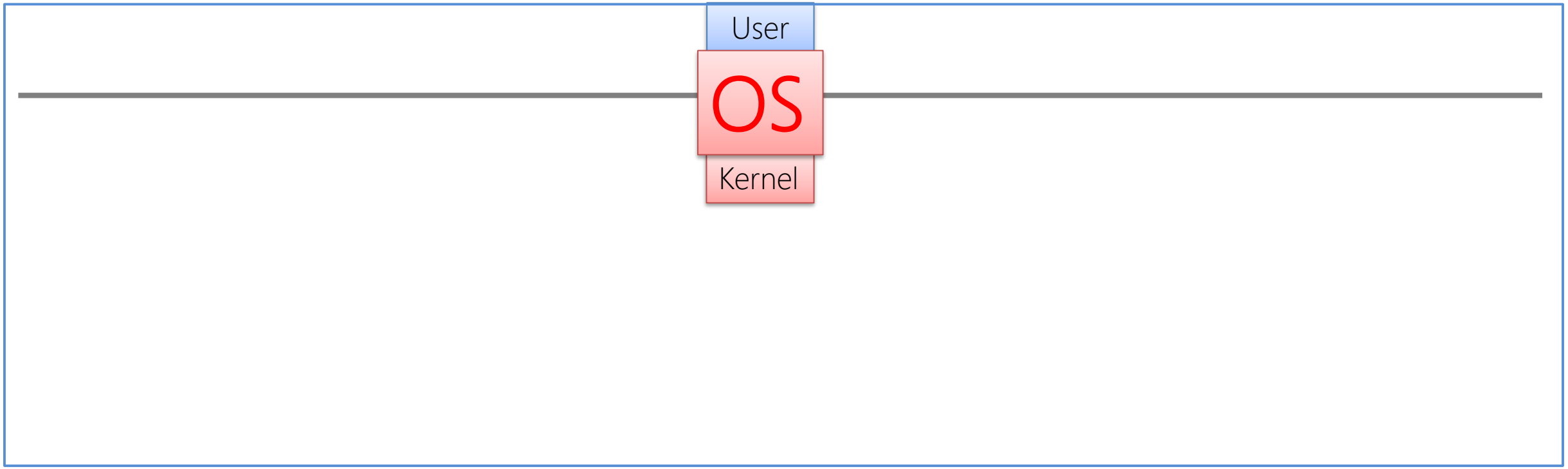
В языке C# нет необходимости использовать API операционной системы. Стандартные библиотеки уже имеют набор типов для удобной работы с операциями ввода-вывода.

C# Асинхронное программирование

Пример синхронной работы с файловой системой

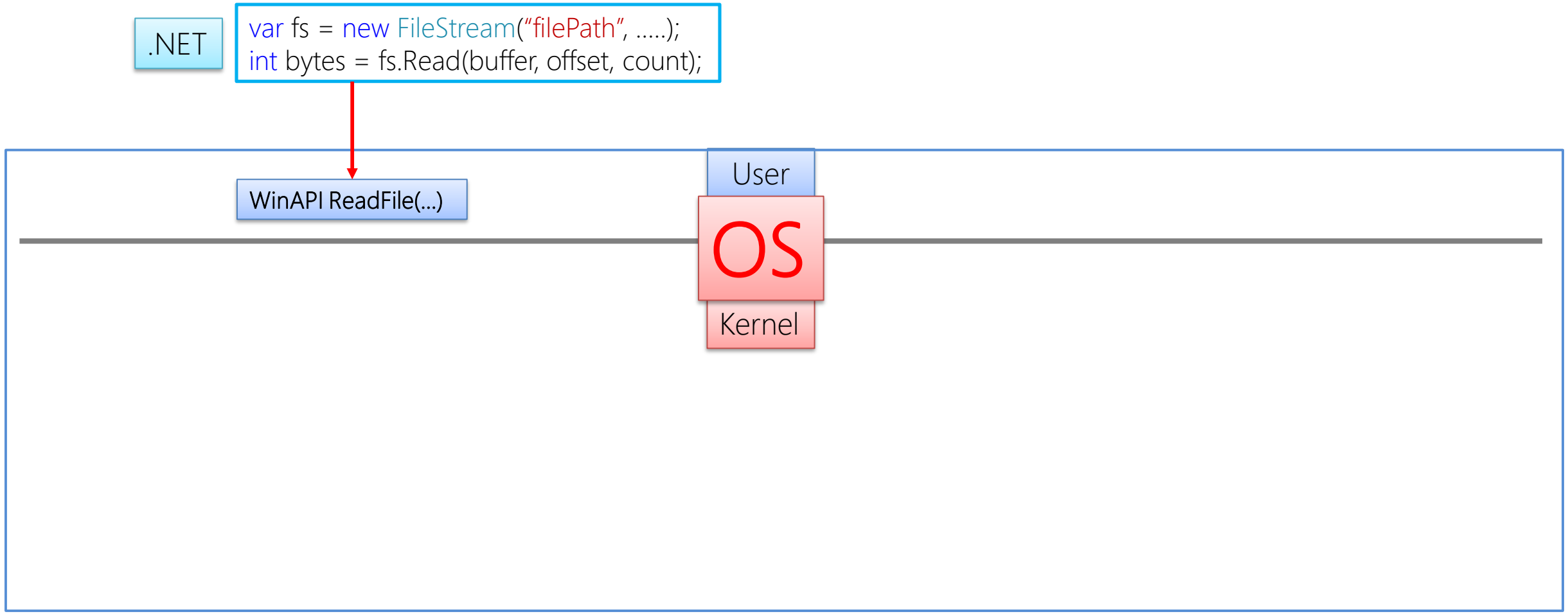
.NET

```
var fs = new FileStream("filePath", .....);
int bytes = fs.Read(buffer, offset, count);
```



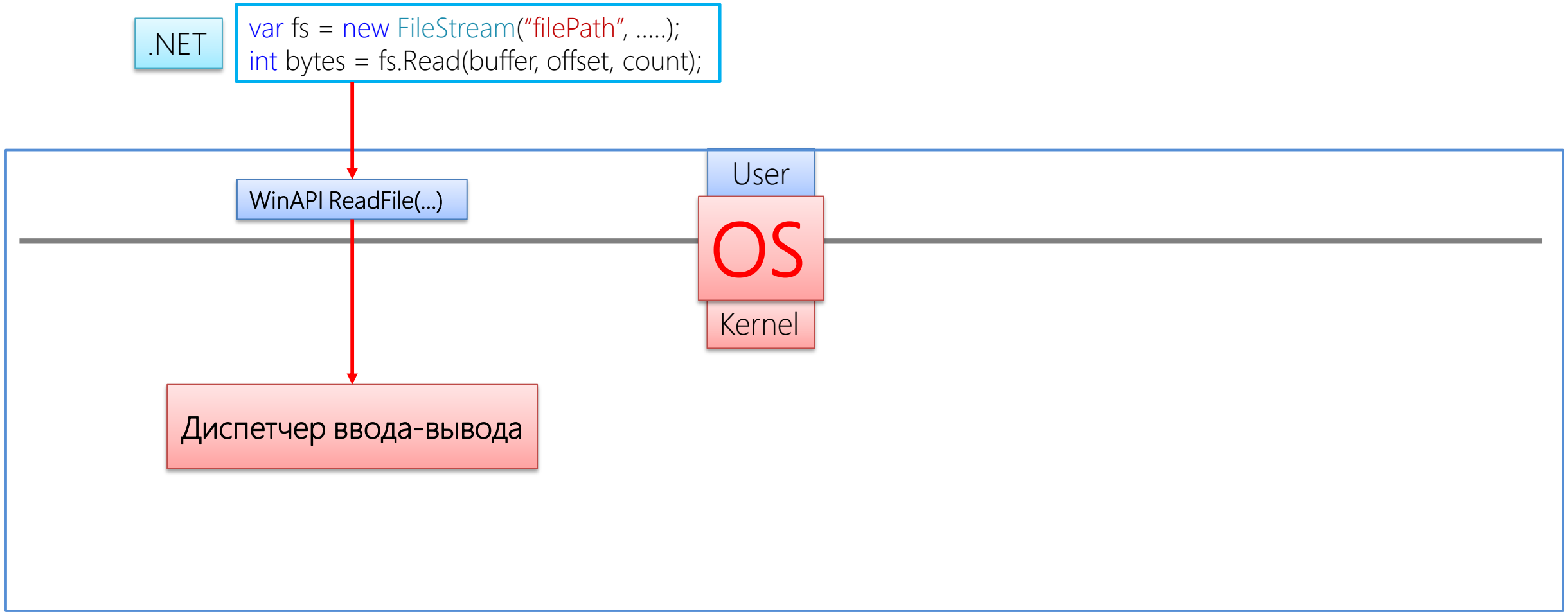
C# Асинхронное программирование

Пример синхронной работы с файловой системой



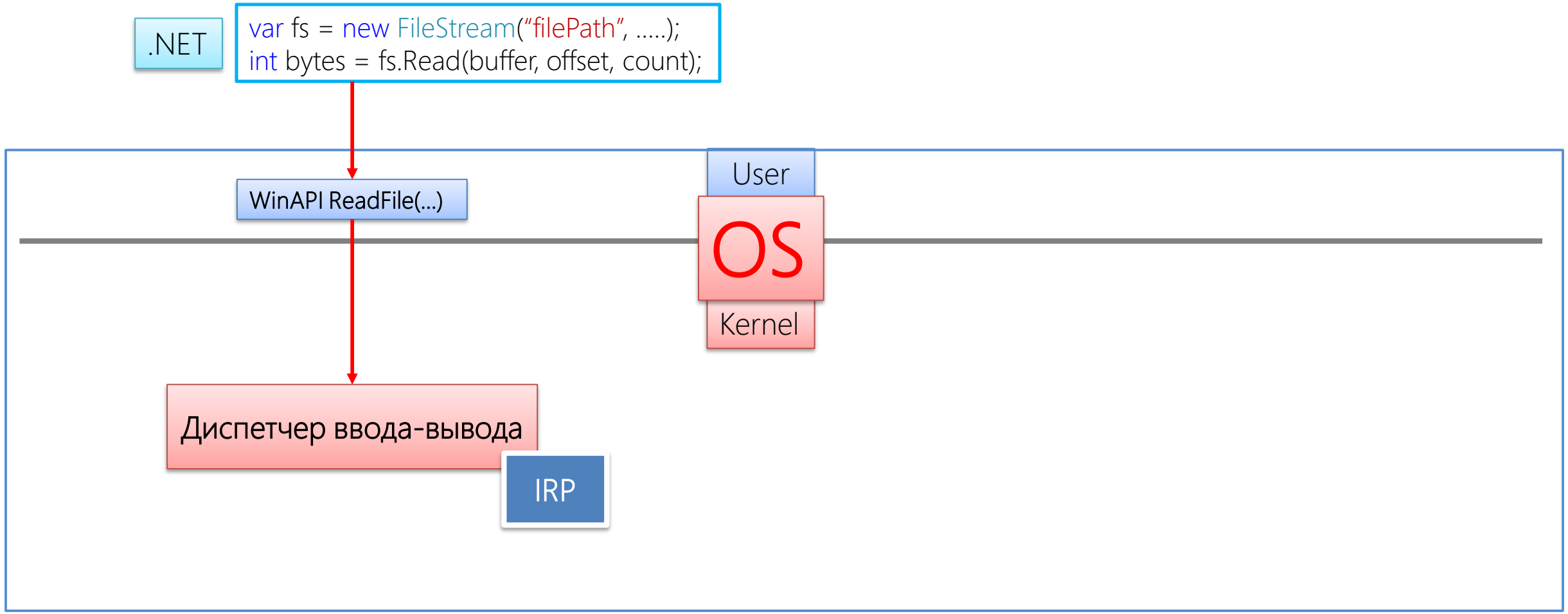
C# Асинхронное программирование

Пример синхронной работы с файловой системой



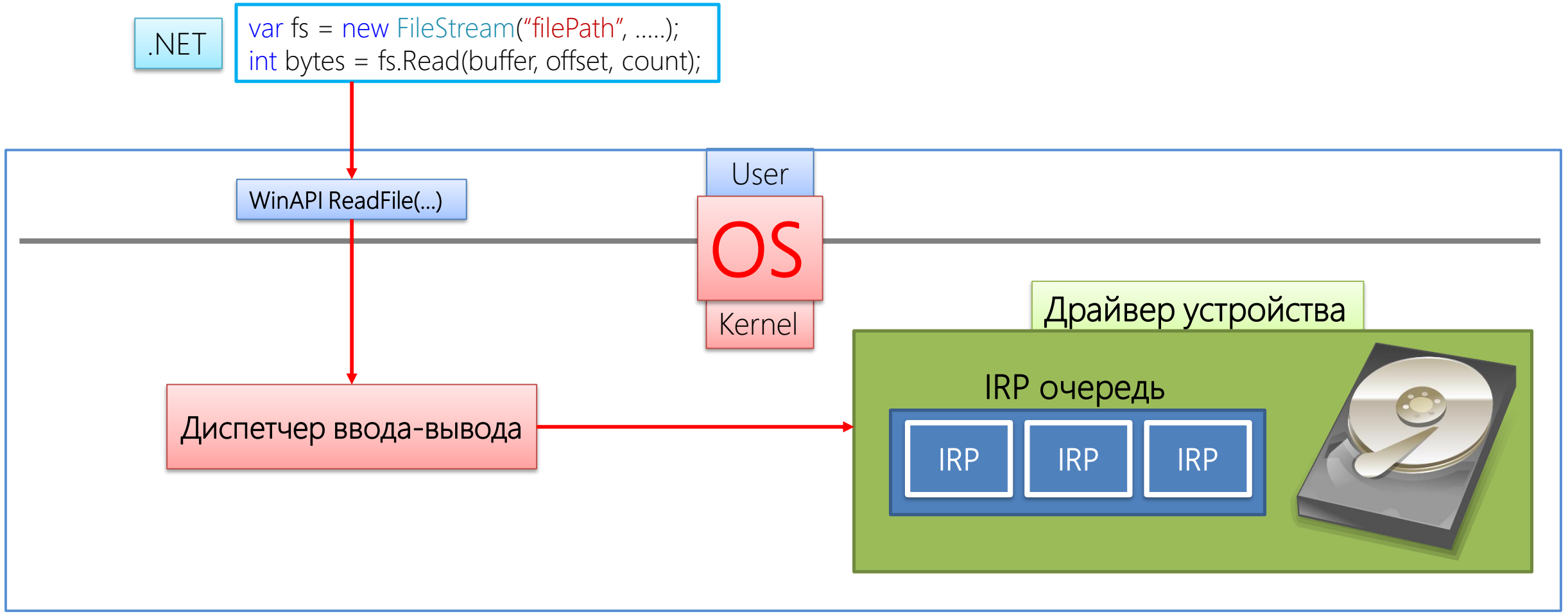
C# Асинхронное программирование

Пример синхронной работы с файловой системой



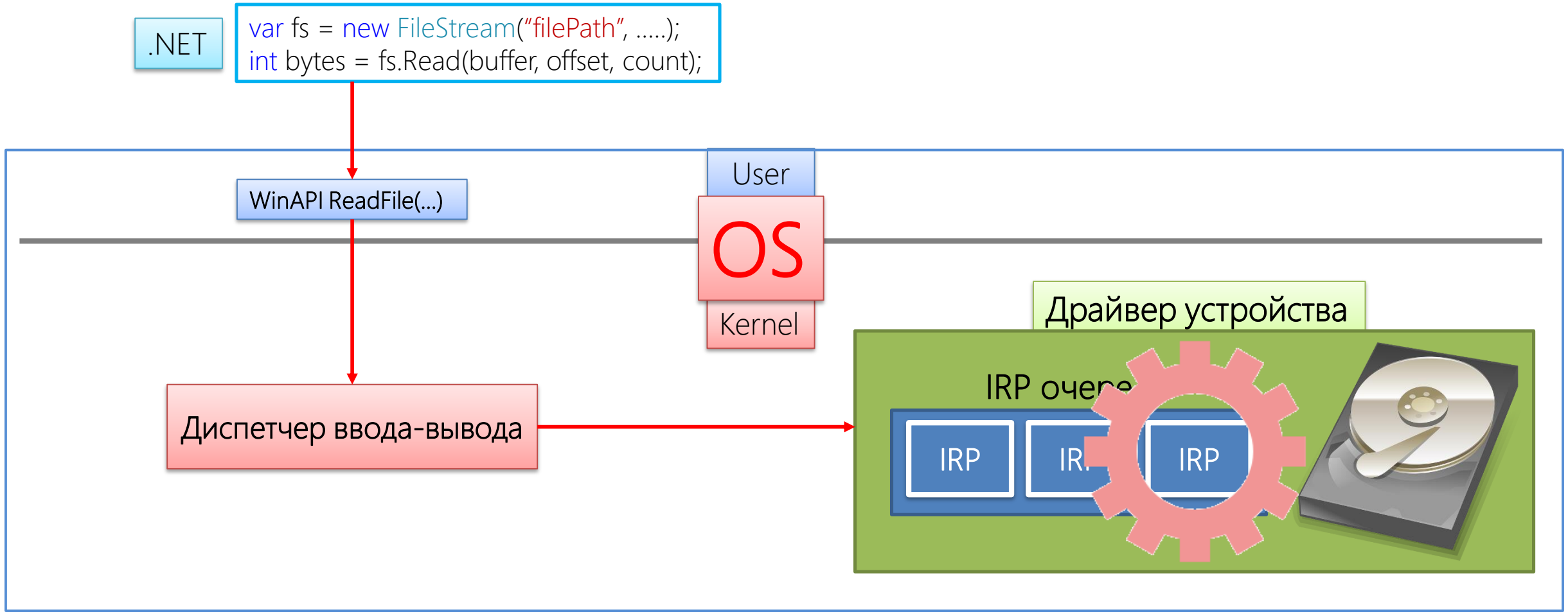
C# Асинхронное программирование

Пример синхронной работы с файловой системой



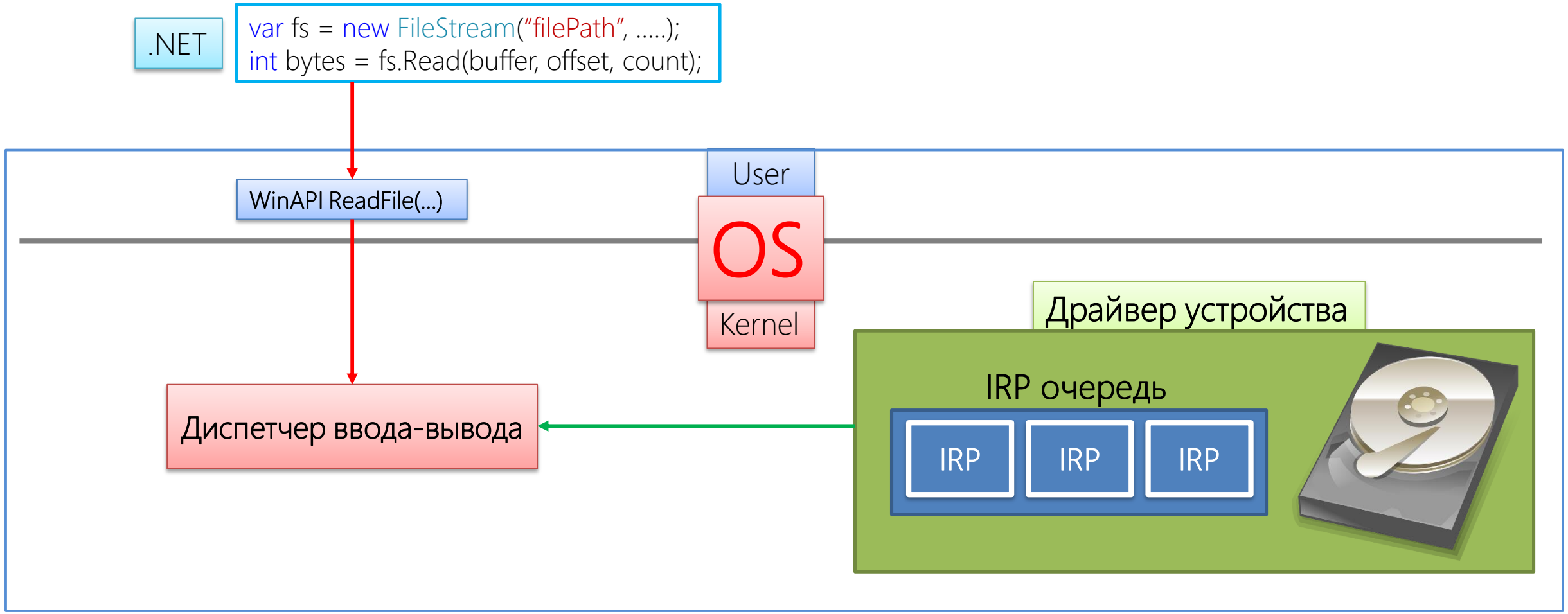
C# Асинхронное программирование

Пример синхронной работы с файловой системой



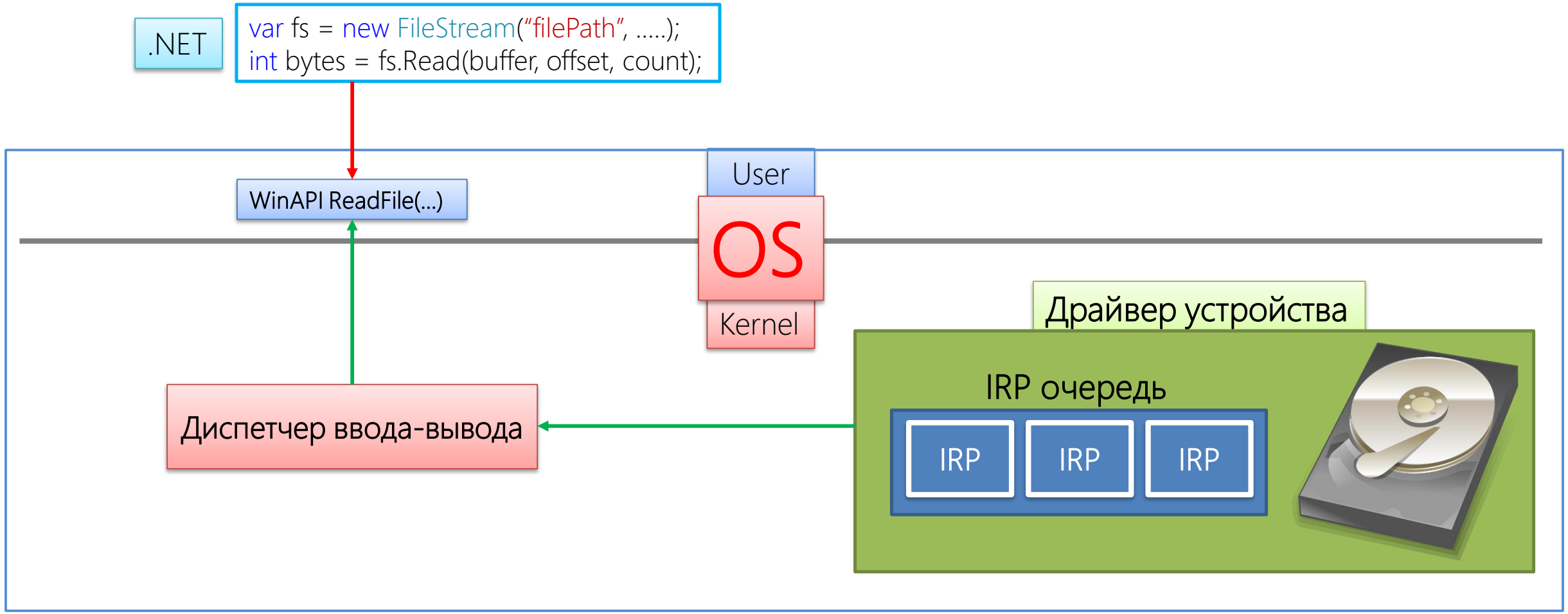
C# Асинхронное программирование

Пример синхронной работы с файловой системой



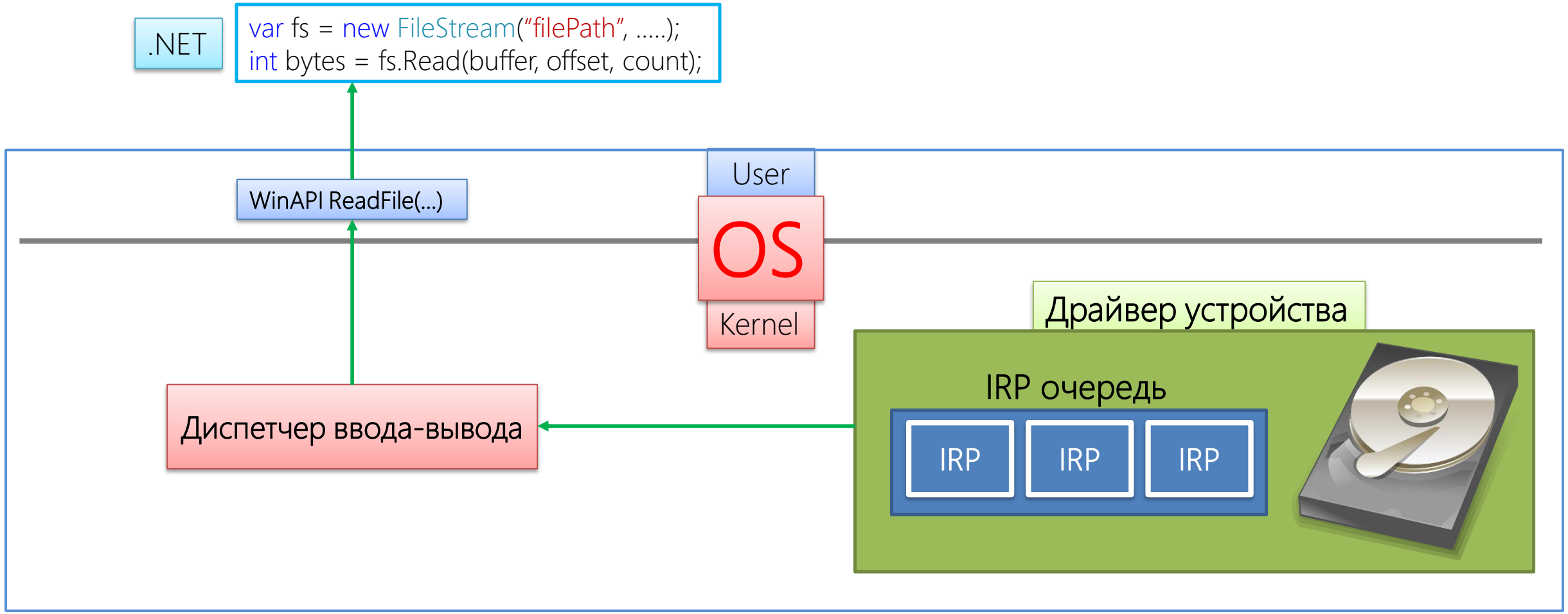
C# Асинхронное программирование

Пример синхронной работы с файловой системой



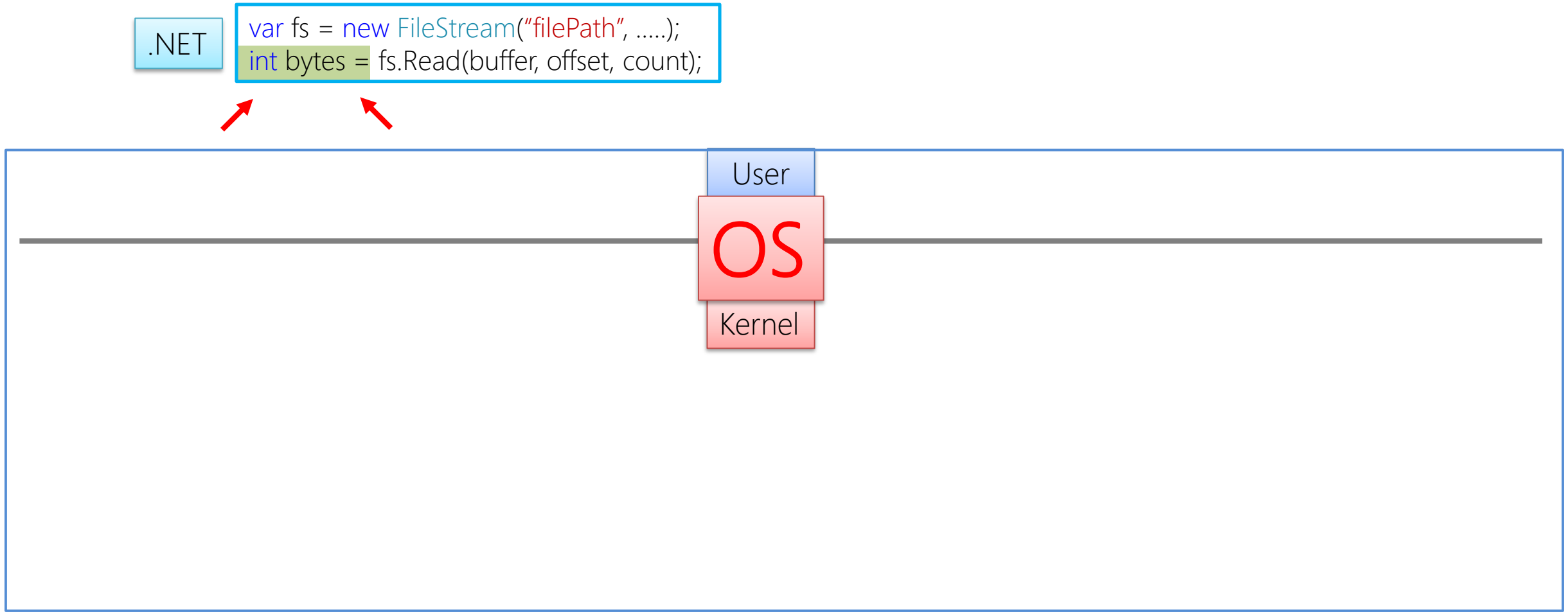
C# Асинхронное программирование

Пример синхронной работы с файловой системой



C# Асинхронное программирование

Пример синхронной работы с файловой системой



Операции ввода-вывода

Аппаратное обеспечение решает задачи ввода-вывода без участия потоков процессора. Это оказывает влияние на эффективность расходования системных ресурсов.

Цель процессора в работе ввода-вывода – это передача запроса на операцию ввода-вывода соответствующему устройству и получение результатов работы от устройства.

Асинхронные операции ввода-вывода

Асинхронный ввод-вывод – это форма неблокирующей обработки ввода-вывода, которая позволяет потоку продолжить свое выполнение, не дожидаясь окончания передачи данных.

При запуске асинхронной операции ввода-вывода, поток продолжает обрабатывать другие операции, пока ядро не отправит сигнал потоку, указывая, что асинхронная операция ввода-вывода завершена.

Перекрывающий ввод-вывод

Перекрывающий ввод-вывод (Overlapped I/O) – это название асинхронного ввода-вывода на уровне API операционной системы Windows.

Перекрывающий ввод-вывод представляет структура OVERLAPPED. Она означает, что исполнение запросов ввода-вывода перекрывается по времени с исполнением потоком других операций.

Чтобы операция ввода-вывода стала асинхронной, необходимо передать специальную структуру OVERLAPPED.

Завершение асинхронной операции ввода-вывода

Есть несколько способов завершения асинхронной (перекрытой) операции ввода-вывода:

1. Событие Win32.
2. Очередь APC (Asynchronous Procedure Call).
3. Порты завершения ввода-вывода (IO Completion Ports).

IO Completion Ports

IO Completion Ports (IOCP) – это объект, являющийся очередью, который используется для одновременного управления несколькими операциями ввода-вывода. Управление производится с помощью привязки дескрипторов к **IOCP**.

По завершении операции над дескриптором, пакет завершения ввода/вывода помещается в очередь **IOCP**.

Объект ThreadPool отвечает за мониторинг **IOCP** и диспетчеризацию задач для потоков портов завершения, отвечающих за обработку завершения операции.

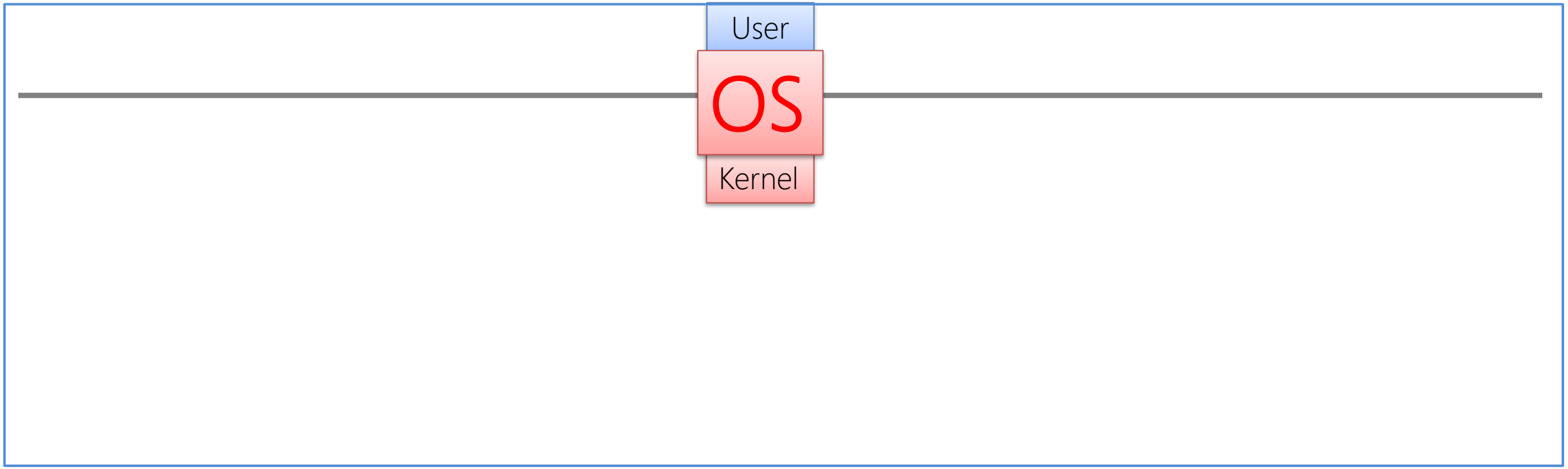
Напрямую с портами завершения сейчас не работают. Программисты используют предоставляемые классы и структуры, которые находятся в библиотеках .NET.

C# Асинхронное программирование

Пример асинхронной работы с файловой системой

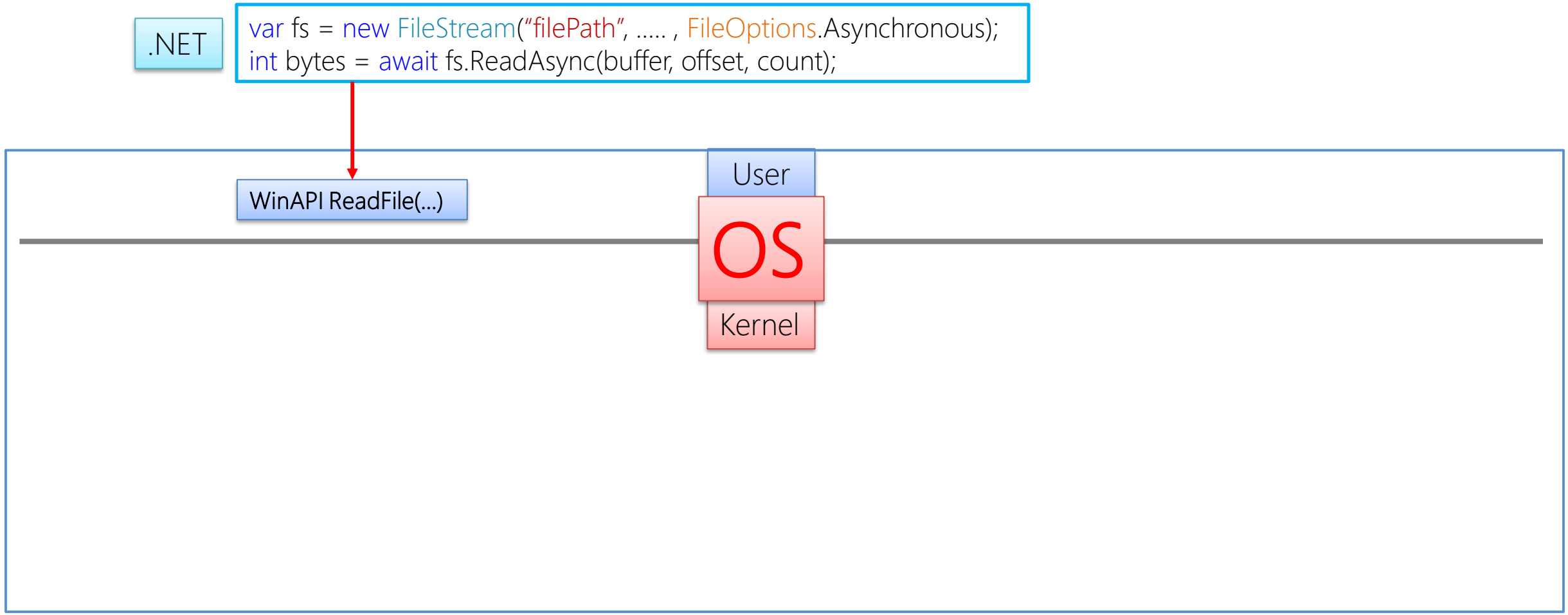
.NET

```
var fs = new FileStream("filePath", ..... , FileOptions.Asynchronous);
int bytes = await fs.ReadAsync(buffer, offset, count);
```



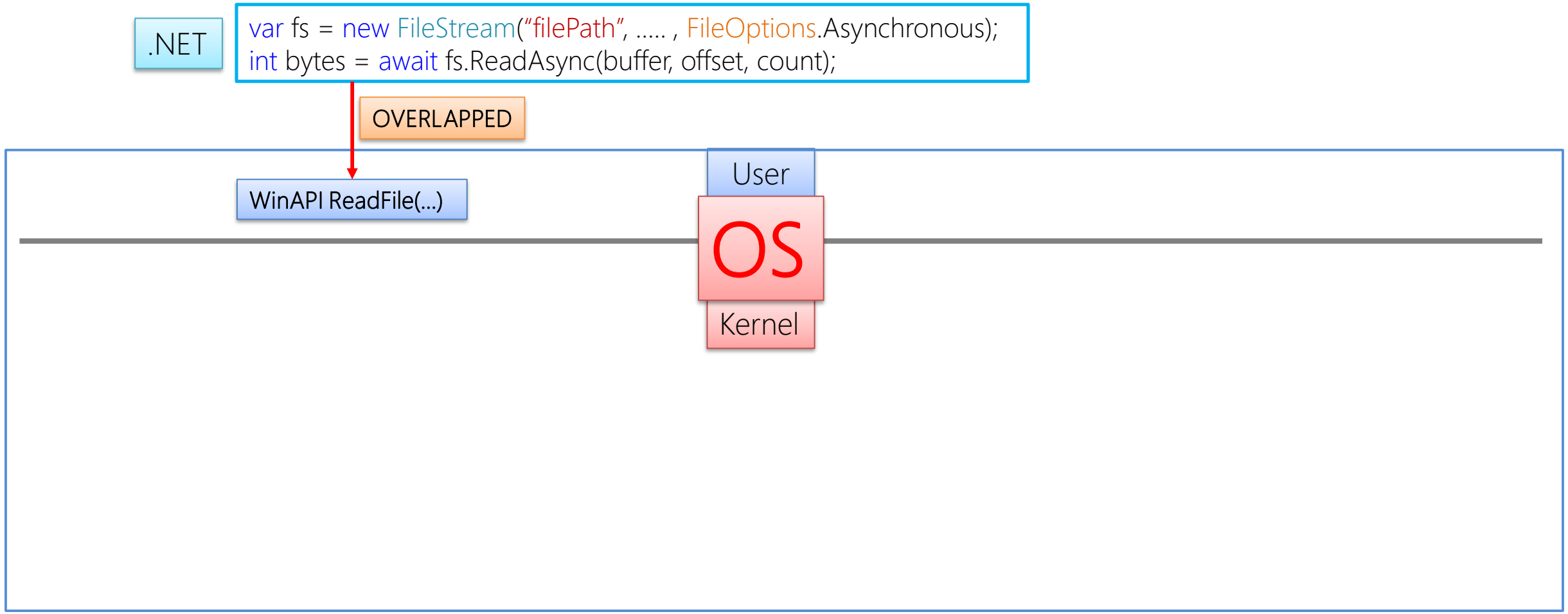
C# Асинхронное программирование

Пример асинхронной работы с файловой системой



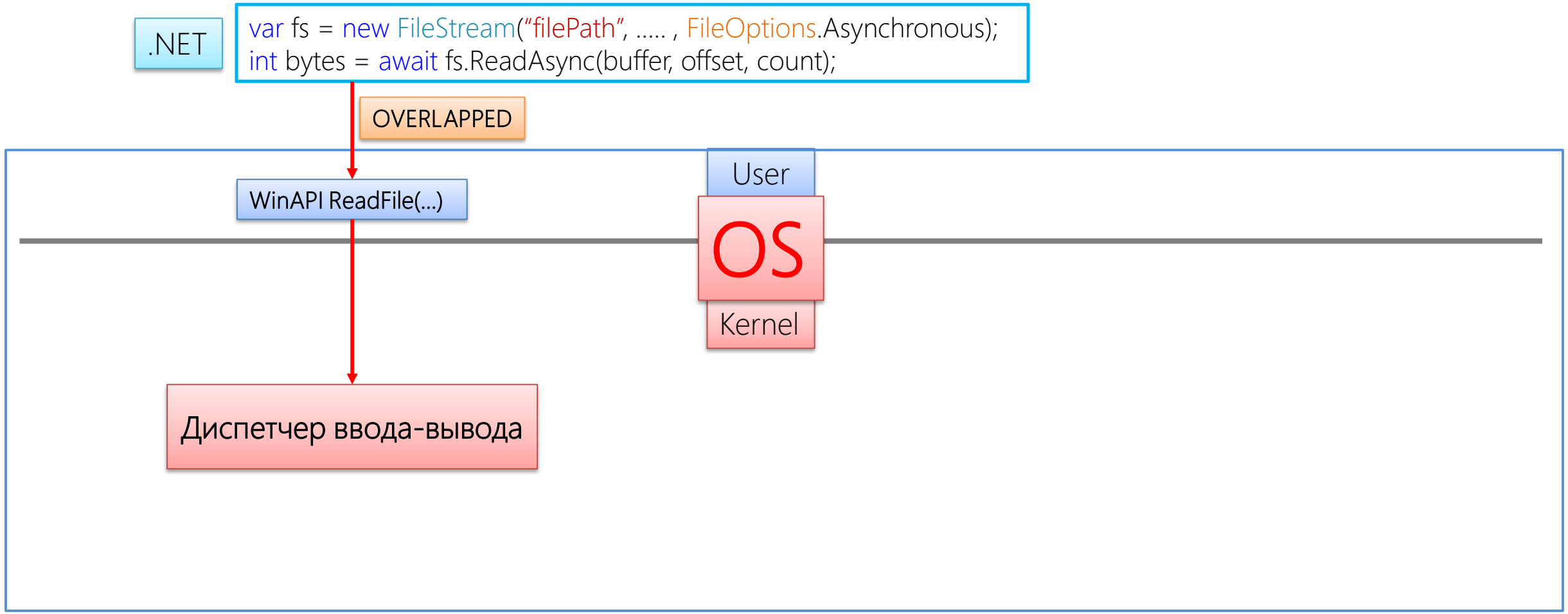
C# Асинхронное программирование

Пример асинхронной работы с файловой системой



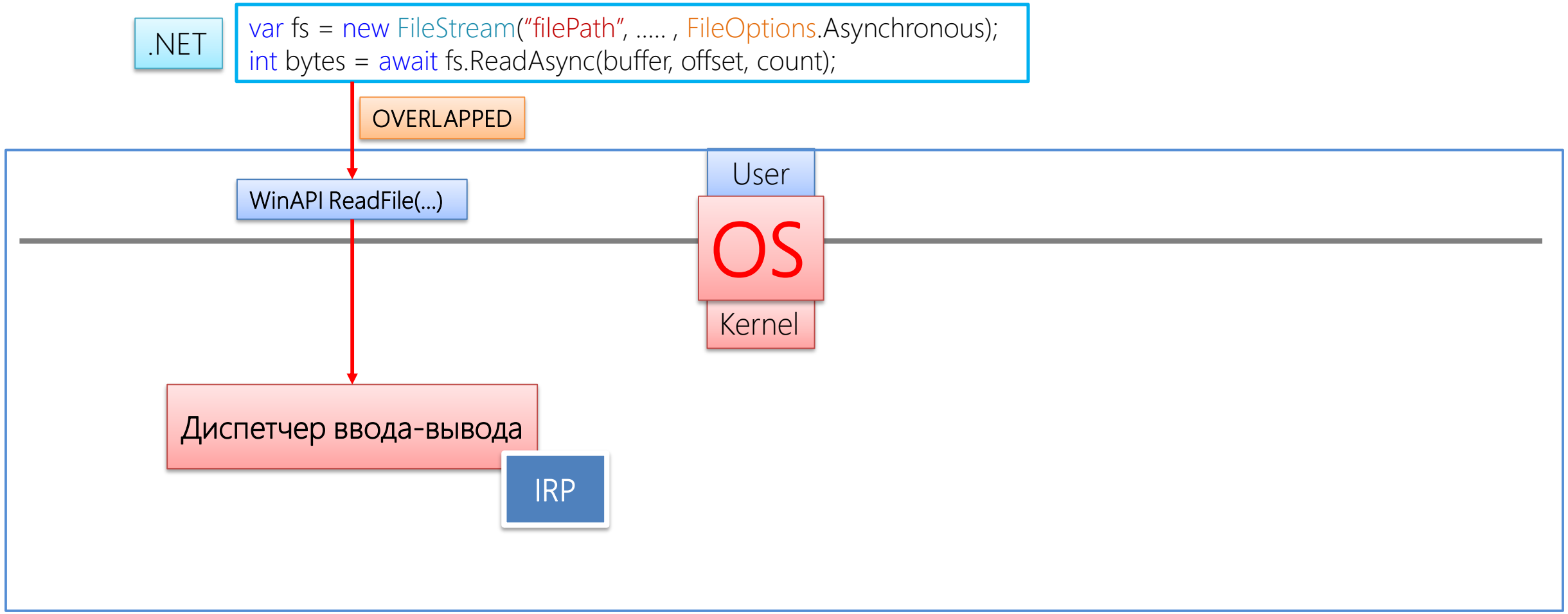
C# Асинхронное программирование

Пример асинхронной работы с файловой системой



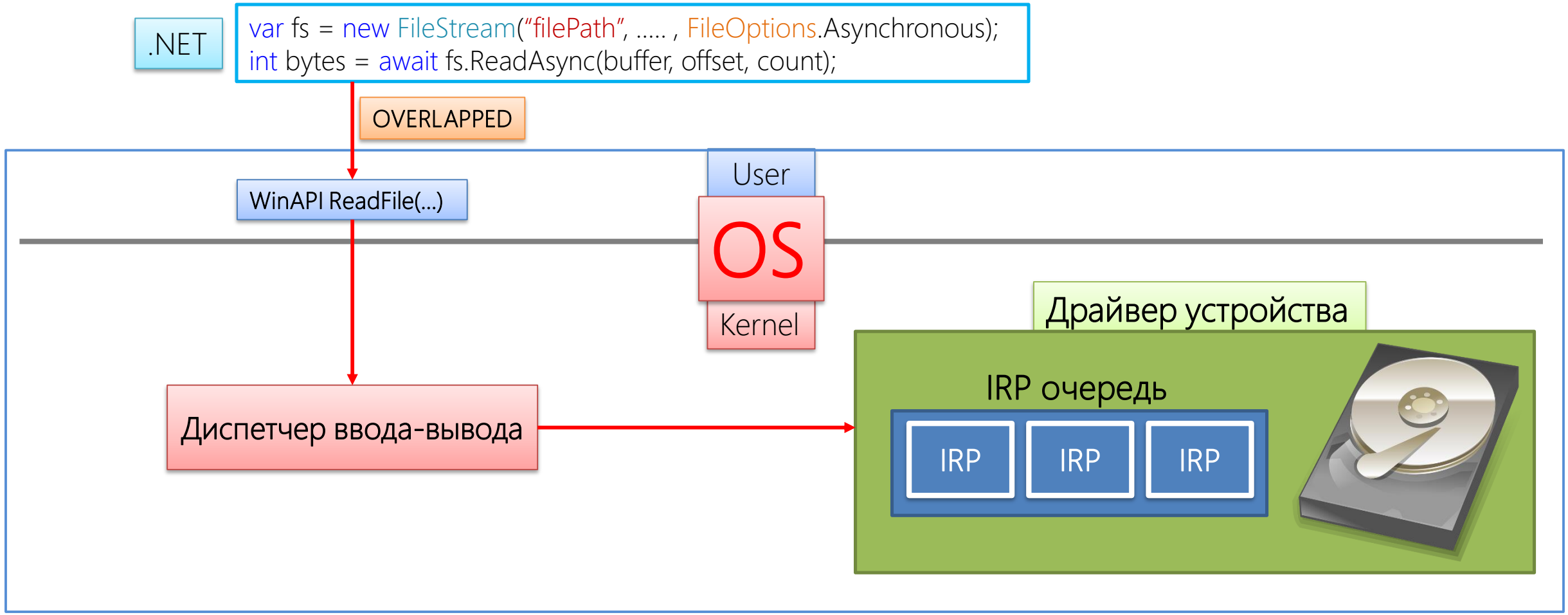
C# Асинхронное программирование

Пример асинхронной работы с файловой системой



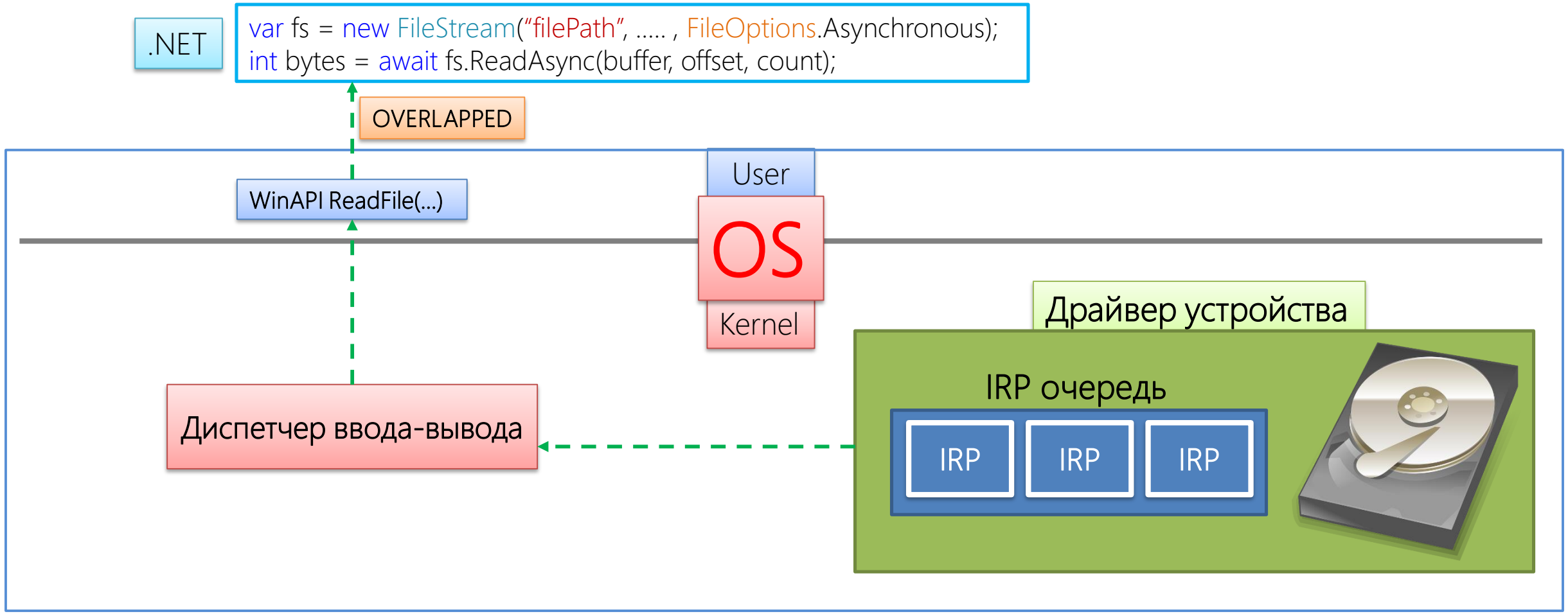
C# Асинхронное программирование

Пример асинхронной работы с файловой системой



C# Асинхронное программирование

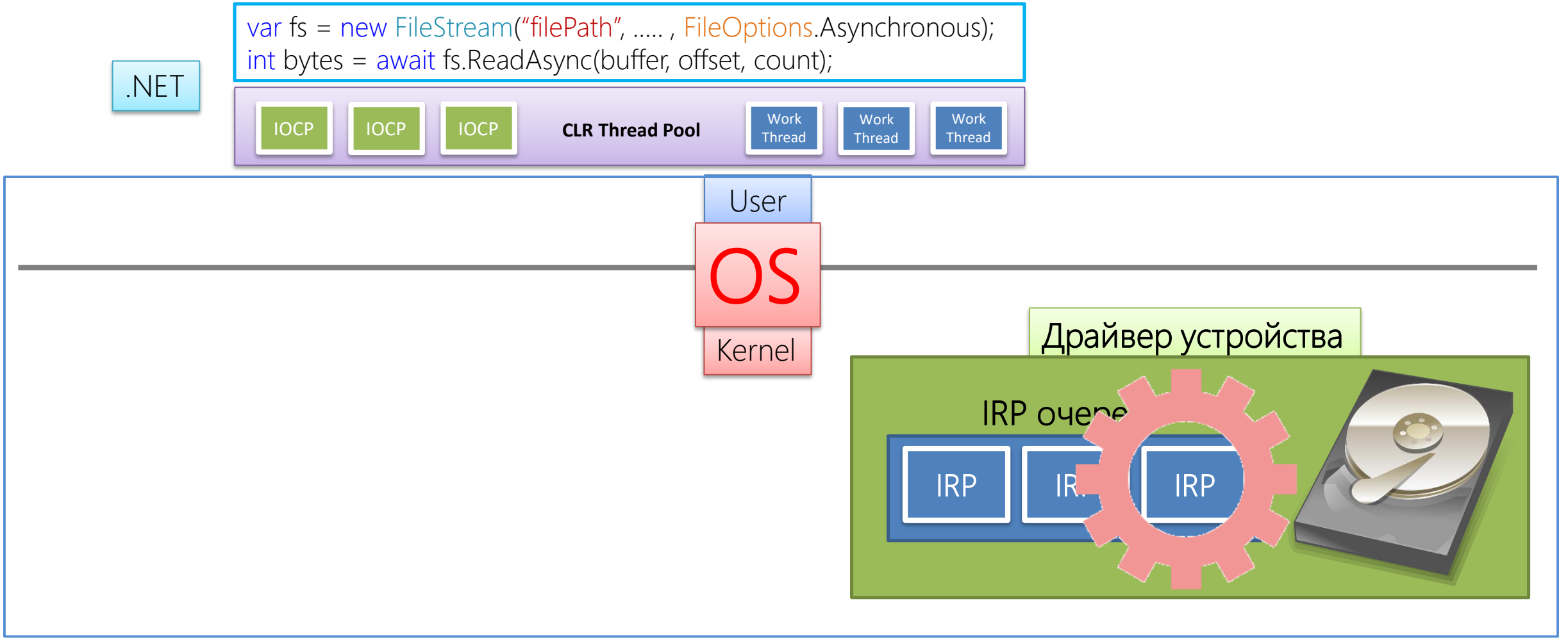
Пример асинхронной работы с файловой системой



C# Асинхронное программирование

Пример асинхронной работы с файловой системой

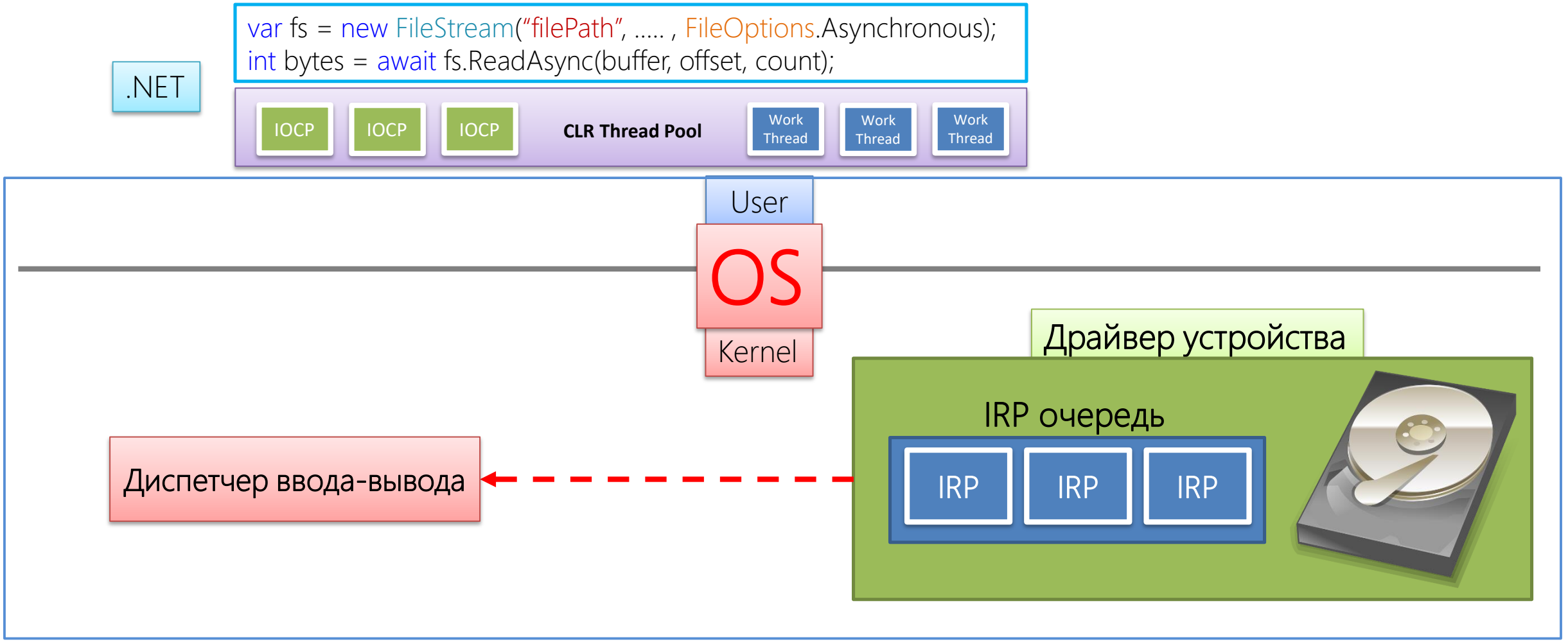
```
var fs = new FileStream("filePath", ..... , FileOptions.Asynchronous);
int bytes = await fs.ReadAsync(buffer, offset, count);
```



C# Асинхронное программирование

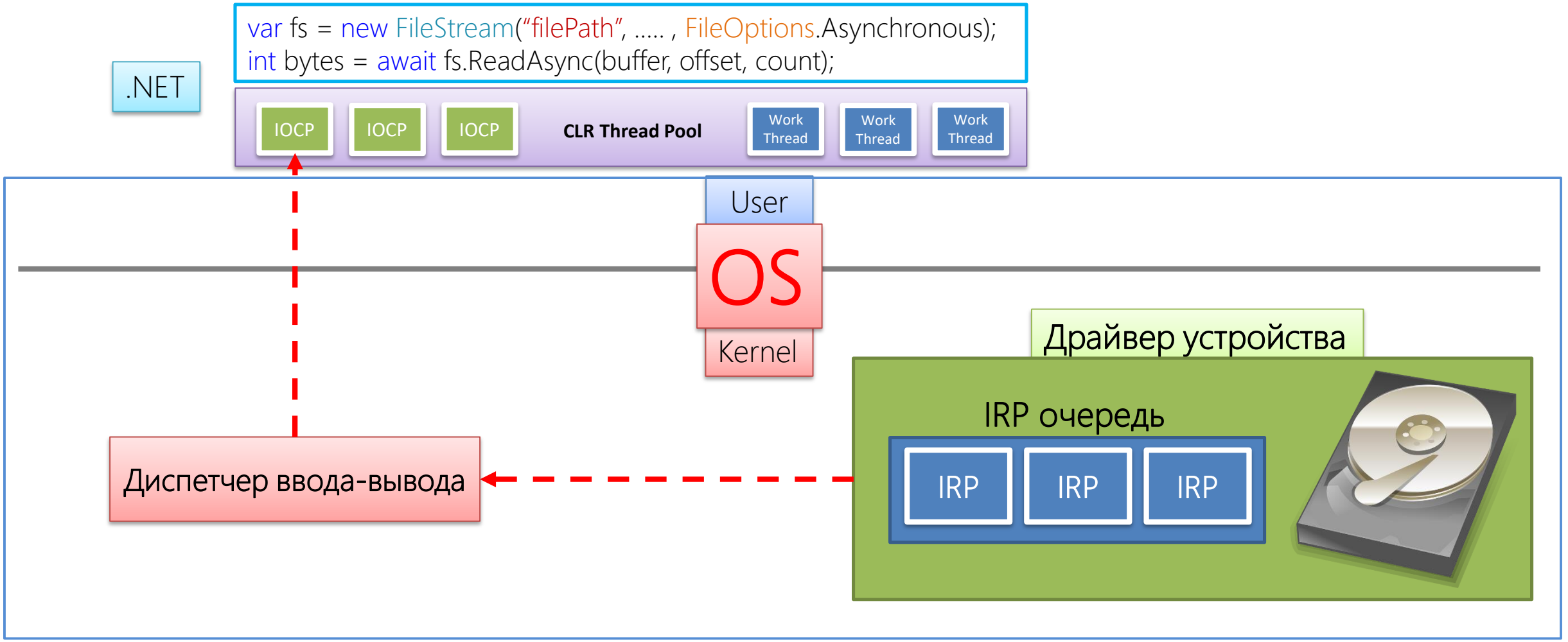
Пример асинхронной работы с файловой системой

```
var fs = new FileStream("filePath", ..... , FileOptions.Asynchronous);
int bytes = await fs.ReadAsync(buffer, offset, count);
```



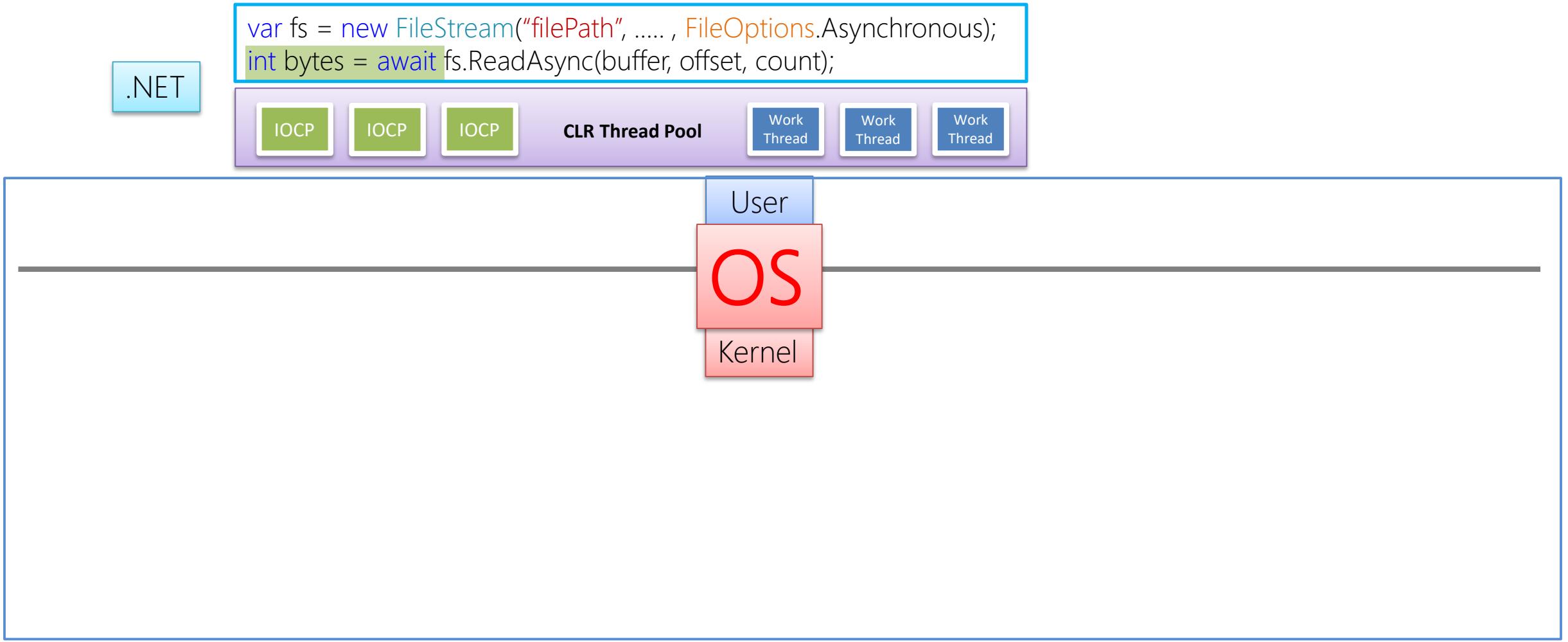
C# Асинхронное программирование

Пример асинхронной работы с файловой системой



C# Асинхронное программирование

Пример асинхронной работы с файловой системой



Асинхронные операции ввода-вывода

Отличия работы синхронного ввода-вывода от асинхронного:

- При запуске синхронной операции ввода-вывода, поток переходит в состояние ожидания до тех пор, пока не получит сигнал о завершении работы операции ввода-вывода.

- При запуске асинхронной операции ввода-вывода, поток продолжает обрабатывать другие операции, пока ядро не отправит сигнал потоку, указывая, что асинхронная операция ввода-вывода завершена.

Асинхронность vs Многопоточность

После появления задач и ключевых слов async await использование асинхронности стало очень простым. Поэтому, программисты начали широко использовать асинхронность. Но, зачастую, люди считают, что асинхронность – это обязательно участие потока (Thread).

Асинхронность не означает, что вы используете поток для выполнения операции. Асинхронное выполнение может происходить без участия потока. Это асинхронный ввод-вывод.

Асинхронность в большинстве случаев как раз и означает асинхронные операции ввода-вывода, именно для этого она и была введена - чтобы использовать неблокирующие операции без участия потоков. Но, при этом, вам никто не запрещает использовать механизмы async await для CPU операций, пользуясь потоками.

Определение «Асинхронность»

Зная о разнице между работой операций для потоков и для ввода-вывода, мы можем сформировать определение асинхронности, которое действительно ей соответствует.

Асинхронность – это неблокирующее выполнение кода.

Асинхронные шаблоны программирования

В платформе .NET существует 3 асинхронных шаблона программирования:

1. APM – Asynchronous Programming Model *(**устарел**, не рекомендуется к применению)*
2. EAP – Event-based Asynchronous Pattern *(**устарел**, не рекомендуется к применению)*
3. TAP – Task-based Asynchronous Pattern *(**используется**, рекомендуется к применению)*

Шаблон APM

Asynchronous Programming Model – шаблон асинхронного программирования, основанный на интерфейсе IAsyncResult и методах BeginXXX и EndXXX.

Для запуска асинхронной операции используется метод BeginXXX. После его вызова приложение может продолжить выполнение инструкций в вызывающем потоке, в это же время асинхронная операция выполняется в другом потоке.

Для завершения асинхронной операции используется метод EndXXX. Он отдает результаты операций.

Шаблон APM поддерживает методы обратного вызова для обработки результатов, не возвращаясь в основной поток. Поэтому, каждый BeginXXX метод всегда принимает два дополнительных параметра:

- делегат, который представляет метод обратного вызова;
- состояние для делегата.

Все делегаты поддерживают асинхронный шаблон APM. Это доступно с помощью методов **BeginInvoke** и **EndInvoke**.

Шаблон EAP

Event-based Asynchronous Pattern – шаблон асинхронного программирования, основанный на событиях.

Для обеспечения шаблона EAP тип должен иметь метод XXXAsync и соответствующее событие XXXCompleted.

Потребитель должен создать обработчик события и подписать его на событие XXXCompleted. После, потребитель вызывает асинхронный метод XXXAsync. По завершении его работы он вызовет событие XXXCompleted, которое вызовет все обработчики события, подписанные на него.

Для передачи асинхронных результатов, исключений, состояния, используются классы, производные от EventArgs. При завершении асинхронной операции в обработчик события будет отдан этот экземпляр в качестве параметра.

Необязательно: *могут поддерживать отмену, отчет о прогрессе или дополнительные результаты для каждого асинхронного метода.*

Шаблон TAP

Task-based Asynchronous Pattern – шаблон асинхронного программирования, основанный на задачах.

Он основан на типах из библиотеки **TPL (Task Parallel Library)** Task и Task<TResult>, и ключевых словах async await.

Для работы с шаблоном TAP создают асинхронные методы, которые возвращают задачу. Асинхронные методы имеют суффикс Async или TaskAsync в названии. Могут использовать ключевые слова async await для повышения абстракции и упрощения асинхронного программирования.

Преимущества использования шаблона TAP:

- Простая инициализация и завершение асинхронной операции.
- Удобный способ получения возвращаемого значения асинхронной операции.
- Получение исключения, возникшего в асинхронной операции для его обработки.
- Просмотр состояния асинхронной операции.
- Поддержка отмены выполнения (Необязательно).
- Продолжения задач (Task Continuations/async await).
- Планирование выполнения асинхронной операции.
- Поддержка прогресса операции (Необязательно).

Задача – Task (Promise)

Задача (Task/Task<TResult>) – единица параллельной обработки, написанная по шаблону «Promise».

Она используется одновременно и как высокоуровневая обертка над работой с **потоками**, и как обертка над работой с **асинхронными операциями ввода-вывода**.

Это возможно благодаря продолжениям (Continuations) или ключевым словам async await.

Асинхронные шаблоны (сравнение в использовании)

Как выглядел метод для загрузки содержимого страницы в разных асинхронных шаблонах:

1) APM (Asynchronous Programming Model):

```
private void APM_Click(object sender, RoutedEventArgs e)
{
        var webRequest = WebRequest.Create("http://microsoft.com/");
        webRequest.BeginGetResponse((iAsyncResult) =>
        {
                var webResponse = webRequest.EndGetResponse(iAsyncResult);
                var reader = new StreamReader(webResponse.GetResponseStream());
                Dispatcher.Invoke(() => txtAPMResult.Text = reader.ReadToEnd());
        }, null);
}
```

2) EAP (Event-based Asynchronous Pattern):

```
private void EAP_Click(object sender, RoutedEventArgs e)
{
        WebClient webClient = new WebClient();
        webClient.DownloadStringCompleted += WebClient_DownloadCompleted;
        webClient.DownloadStringAsync(new Uri("http://microsoft.com"));
}

private void WebClient_DownloadCompleted(object sender, DownloadStringCompletedEventArgs e)
{
        txtEAPResult.Text = e.Result;
}
```

3) TAP (Task-based Asynchronous Pattern):

```
private async void TAP_Click(object sender, RoutedEventArgs e)
{
        var result = await new HttpClient().GetStringAsync("http://microsoft.com/");
        txtTAPResult.Text = result;
}
```

Превращение APM и EAP в TAP

Если у вас будут проблемы с использованием старых API, которые работают на шаблоне **APM** или **EAP**, вы можете с помощью специального типа переписать их в **TAP**.

Для этого используют класс TaskCompletionSource. Он позволяет создавать задачи-марионетки.

Асинхронный шаблон **APM** имеет настолько много API, что под него были созданы готовые методы для быстрого преобразования в асинхронный шаблон **TAP**. Это методы FromAsync из фабрики задач.

TaskCompletionSource<TResult>

TaskCompletionSource<TResult> - создает асинхронные операций в виде задач.

Он порождает задачу-марионетку, состояние и завершение которой контролируется с помощью методов класса TaskCompletionSource. При этом, для потребителей взаимодействие будет выглядеть как работа с обычной задачей, они не почувствуют разницы.

Этот класс можно использовать для создания своего типа асинхронных операций.

Task.Delay

Task.**Delay** – статический метод, который создает задачу и делает ее завершенной через указанный интервал времени. Время можно указать в миллисекундах (тип int) или в промежутке (тип TimeSpan).

Используется для задержки выполнения асинхронной задачи.

Является асинхронным вариантом метода Thread.**Sleep**(), поддерживается оператором await через возврат задачи.

Метод Delay поддерживает отмену через CancellationToken.

Task.WhenAll

Task.**WhenAll** – статический метод, который ожидает завершение всех переданных задач. Метод возвращает задачу или задачу с массивом результатов выполнившихся задач типа TResult.

Имеет 4 перегруженных варианта:

- `Task WhenAll(params Task[] tasks);`
- `Task WhenAll(IEnumerable<Task> tasks);`
- `Task<TResult[]> WhenAll(params Task<TResult>[] tasks);`
- `Task<TResult[]> WhenAll(IEnumerable<Task<TResult>> tasks);`

Если переданный массив или коллекция не содержит задач, результирующая задача получит состояние «успешно завершена» (RanToCompletion) еще до того, как она вернется пользователю.

Task.WhenAll

Состояние результирующей задачи:

- Если хоть одна из переданных задач будет завершена в состоянии Failed, то результирующая задача тоже будет провалена. Свойство Exception результирующей задачи будет содержать набор всех исключений из каждой задачи, где оно произошло.
- Если все задачи не были провалены (Failed), но хоть одна из них была отменена, то результирующая задача получит состояние Canceled.
- Если все задачи не были провалены (Failed) и не были отменены (Canceled), то результирующая задача получит состояние RanToCompletion (успешно завершена).

Task.WhenAny

Task.**WhenAny** – статический метод, который ожидает завершение одной, только первой задачи из переданных в параметрах. По завершении возвращает задачу или задачу с результатом типа TResult первой выполнившейся задачи.

Имеет 4 перегруженных варианта:

- Task WhenAny(params Task[] tasks);
- Task WhenAny(IEnumerable<Task> tasks);
- Task<Task<TResult>> WhenAny(params Task<TResult>[] tasks);
- Task<Task<TResult>> WhenAny(IEnumerable<Task<TResult>> tasks).

Task.WhenAny

Состояние результирующей задачи:

- Результирующая задача будет возвращена после первого завершения любой из переданных задач.
- Результирующая задача всегда будет завершена в состоянии RanToCompletion с установленным результатом, который был получен от первой завершившейся задачи.
- Такое поведение будет даже если первая завершенная задача завершилась с ошибкой (Failed) или отменой (Cancelled).

Ограничения в использовании оператора await

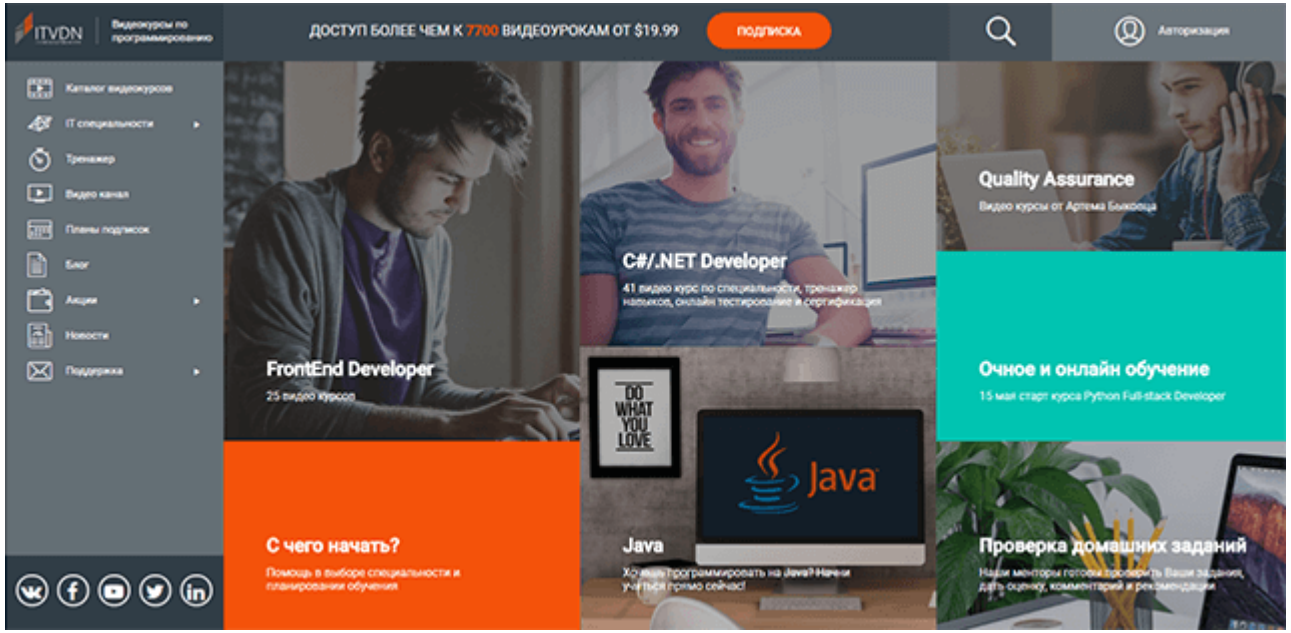
Оператор await нельзя использовать в следующих ситуациях:

- В теле синхронного метода, лямбда-выражения, анонимного метода
- В блоке оператора lock
- В выражении запроса (LINQ)
- В небезопасном (unsafe) контексте
- Нельзя создавать экземпляры ref struct типов в асинхронных методах.
- В блоке catch (НАЧИНАЯ С ВЕРСИИ ЯЗЫКА C# 6.0 - РАЗРЕШЕНО ИСПОЛЬЗОВАТЬ)
- В блоке finally (НАЧИНАЯ С ВЕРСИИ ЯЗЫКА C# 6.0 - РАЗРЕШЕНО ИСПОЛЬЗОВАТЬ)
- В методе Main (НАЧИНАЯ С ВЕРСИИ ЯЗЫКА C# 7.1 – РАЗРЕШЕНО ИСПОЛЬЗОВАТЬ)

Q&A

Смотрите наши уроки в видео формате

ITVDN.com



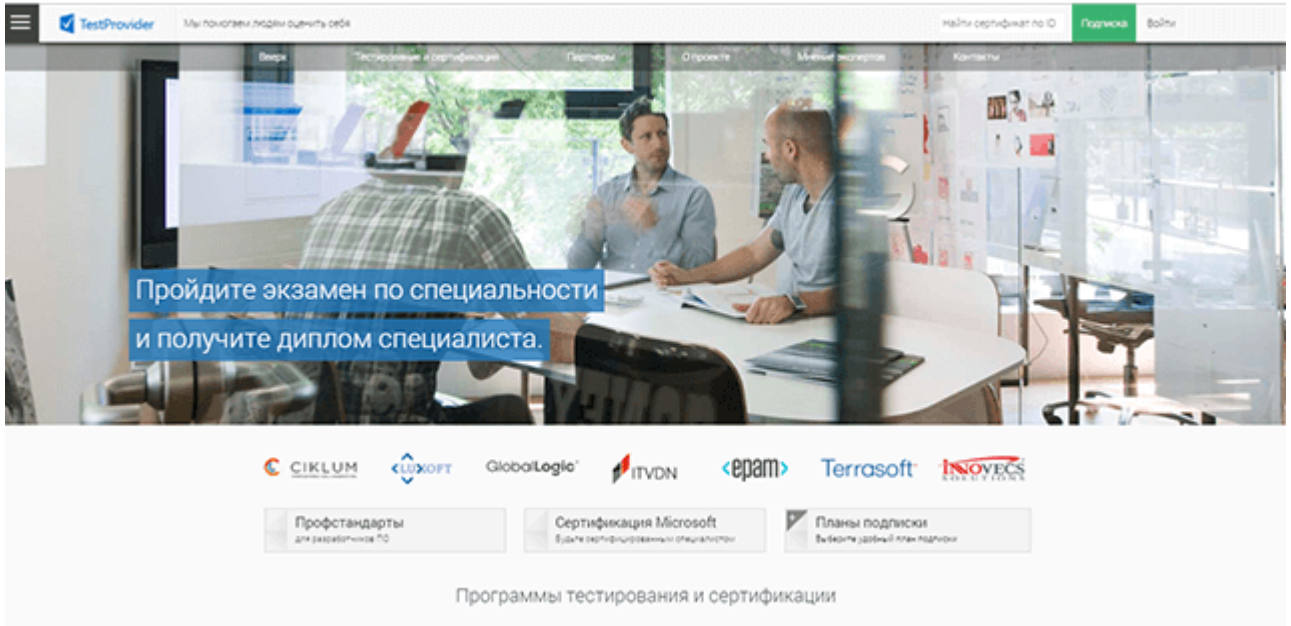
Посмотрите этот урок в видео формате на образовательном портале ITVDN.com для закрепления пройденного материала.

Курсы записаны сертифицированными тренерами, которые работают в учебном центре CyberBionic Systematics и другими высококвалифицированными разработчиками.

ITVDN
IT VIDEO DEVELOPERS NETWORK

Проверка знаний

TestProvider.com



TestProvider – это online сервис проверки знаний по информационным технологиям. С его помощью Вы можете оценить Ваш уровень и выявить слабые места. Он будет полезен как в процессе изучения технологии, так и для общей оценки знаний IT специалиста.

После каждого урока проходите тестирование для проверки знаний на TestProvider.com

Успешное прохождение финального тестирования позволит Вам получить соответствующий Сертификат.

TestProvider

Информационный видеосервис для разработчиков программного обеспечения

ITVDN

IT VIDEO DEVELOPERS NETWORK

TestProvider PROMETRIC TEST CENTER CyberBionic systematics Microsoft Partner Silver Learning Windows Azure Cloud Partner EBA European Business Association