# Архитектурная трансформация и управление данными: Глубокий анализ подсистем ввода-вывода и асинхронности в среде.NET Enterprise

## 1. Введение: Смена парадигмы ввода-вывода в современных серверных архитектурах

Переход от разработки интерактивных приложений реального времени, таких как игровые движки (в частности, Unity), к проектированию высоконагруженных распределенных систем на платформе Enterprise.NET требует не просто смены инструментария, но фундаментальной перестройки ментальных моделей архитектора. В среде Unity разработчик оперирует в рамках жестко детерминированного игрового цикла (Game Loop), где доступ к файловой системе часто рассматривается как допустимая, пусть и дорогая, операция, синхронизированная с жизненным циклом кадра. В противоположность этому, архитектура современного бэкенда на базе ASP.NET Core диктует требования абсолютной асинхронности, отсутствия состояния (statelessness) и способности обрабатывать тысячи конкурентных запросов ввода-вывода (I/O) без блокировки вычислительных ресурсов.1

В контексте пятой недели программы обучения, сфокусированной на "Работе с файловой системой и конфигурацией", первый день является критически важным этапом, закладывающим теоретический фундамент. Анализ учебного плана показывает, что поверхностного знания методов класса File или синтаксического сахара async/await категорически недостаточно для специалиста уровня Senior Backend Developer. Требуется глубокое понимание механики взаимодействия Common Language Runtime (CLR) с ядром операционной системы (Kernel Space), принципов управления буферами памяти во избежание фрагментации кучи, а также низкоуровневых алгоритмов планирования потоков, предотвращающих истощение пула (ThreadPool Starvation).1

Настоящий отчет представляет собой исчерпывающий технический анализ теоретических основ первого дня. Он деконструирует механику системных вызовов, исследует эволюцию подсистем ввода-вывода от устаревших моделей к современным io\_uring и I/O Completion Ports, а также детально рассматривает стратегии буферизации, оптимизированные под работу Garbage Collector (GC). Цель документа — сформировать у читателя целостное представление о "физике" процессов ввода-вывода, позволяющее принимать обоснованные архитектурные решения при проектировании высокопроизводительных систем хранения и обработки данных.

### 1.1. От однопоточного детерминизма к стохастической конкурентности

Фундаментальное различие между клиентской (игровой) и серверной разработкой кроется в модели управления временем и ресурсами. В игровой разработке основной поток (Main Thread) является священным ресурсом, который должен завершить кадр за 16.6 мс для обеспечения 60 FPS. В этой парадигме I/O операции часто выносятся за скобки игрового процесса (загрузочные экраны) или строго контролируются.

В серверной разработке концепция "основного потока" размывается. Вместо этого мы имеем дело с Пулом Потоков (ThreadPool) — сложным механизмом управления рабочими единицами, который обслуживает тысячи конкурентных запросов. Любая блокировка потока пула в ожидании завершения дисковой операции (I/O Bound) приводит к явлению, известному как ThreadPool Starvation. Когда рабочие потоки блокируются синхронным ожиданием, диспетчер пула вынужден создавать новые потоки (Hill Climbing heuristic), что увеличивает накладные расходы на переключение контекста и потребление памяти, в конечном итоге приводя к деградации сервиса.2

## 2. Механика файлового ввода-вывода: От User Space к Kernel Space

Понимание производительности файловых операций невозможно без детального анализа стоимости взаимодействия между приложением и операционной системой. В современной архитектуре фон-неймановских машин и операционных систем с защищенным режимом (protected mode), прямой доступ к аппаратному обеспечению (диску) из пользовательского кода запрещен для обеспечения безопасности и стабильности системы.

### 2.1. Граница защиты: User Mode и Kernel Mode

Процессор выполняет код в различных режимах привилегий, часто называемых кольцами защиты (protection rings). Прикладной код.NET выполняется в кольце с наименьшими привилегиями — Ring 3 (User Mode), тогда как драйверы устройств и ядро операционной системы работают в Ring 0 (Kernel Mode). Это разделение обеспечивает изоляцию процессов: сбой в пользовательской программе не может обрушить всю систему, так как она не имеет прямого доступа к критическим структурам данных ядра.4

Каждая операция чтения или записи файла, инициируемая через высокоуровневый класс FileStream, неизбежно влечет за собой системный вызов (syscall). Это не просто вызов функции, а сложная процедура смены режима процессора, которая включает в себя сохранение состояния текущего потока (регистров, стека), переключение таблиц страниц памяти и передачу управления диспетчеру системных вызовов ядра.6

### 2.2. Стоимость системного вызова и переключения контекста

Переключение контекста между User Mode и Kernel Mode является одной из самых дорогостоящих операций в современных вычислениях. Измерения показывают, что "чистое" переключение занимает от 1000 до 1500 процессорных тактов.8 Однако реальная стоимость в условиях современной загруженной системы значительно выше из-за косвенных эффектов, связанных с работой кэш-памяти процессора.

#### Загрязнение кэша (Cache Pollution)

При переходе в ядро кэш-линии процессора (L1, L2) заполняются кодом и данными ядра, вытесняя данные приложения. Когда управление возвращается в User Mode, процессор сталкивается с серией промахов кэша (cache misses), что заставляет его простаивать сотни тактов в ожидании подгрузки данных из оперативной памяти.9 В сценариях с интенсивным вводом-выводом этот эффект может деградировать производительность на порядки сильнее, чем сами вычисления.

#### Влияние уязвимостей спекулятивного выполнения

Ситуация усугубилась с открытием уязвимостей класса Spectre и Meltdown. Программные заплатки, такие как KPTI (Kernel Page Table Isolation) в Linux, требуют полной подмены таблиц страниц памяти при каждом входе и выходе из ядра, чтобы предотвратить чтение памяти ядра из пользовательского пространства. Это увеличило накладные расходы на системные вызовы на 30% и более, делая каждую операцию ввода-вывода еще более дорогой.11

### 2.3. Цепочка обработки прерываний: ISR и DPC

Когда запрос на чтение файла достигает драйвера файловой системы (например, ntfs.sys или xfs), он транслируется в команды контроллеру диска (NVMe, SATA). Диск выполняет операцию физически, что занимает миллисекунды — вечность по меркам процессора. Чтобы не блокировать CPU, используется механизм прерываний.

1. **Interrupt Service Routine (ISR):** По завершении операции диск генерирует аппаратное прерывание. Процессор приостанавливает текущую работу и вызывает ISR. ISR работает на высоком уровне приоритета (DIRQL) и должна выполняться максимально быстро. Она лишь сохраняет критические данные и планирует дальнейшую обработку.13
2. **Deferred Procedure Call (DPC):** Основная логика завершения ввода-вывода (копирование данных в буфер пользователя, уведомление процесса) выполняется в отложенном вызове процедуры (DPC), который работает на более низком уровне приоритета (DISPATCH\_LEVEL). Это позволяет системе оставаться отзывчивой к новым прерываниям.14

Эта многоступенчатая архитектура (User Code -> Syscall -> Driver -> Hardware -> Interrupt -> ISR -> DPC -> User Code) объясняет, почему синхронный ввод-вывод неэффективен: он заставляет поток ждать завершения всей этой цепочки. Асинхронный ввод-вывод освобождает поток сразу после отправки команды драйверу.

## 3. Архитектура асинхронности: Windows IOCP против Linux io\_uring

В экосистеме.NET класс FileStream является абстракцией, скрывающей под собой кардинально различные механизмы асинхронности в зависимости от операционной системы. Понимание этих различий критично для диагностики проблем производительности и корректной настройки окружения.

### 3.1. Windows: I/O Completion Ports (IOCP)

В операционной системе Windows асинхронный ввод-вывод (известный как Overlapped I/O) является "гражданином первого класса". Механизм портов завершения ввода-вывода (I/O Completion Ports - IOCP) служит фундаментом для высокопроизводительных серверов, таких как IIS и SQL Server, и полностью интегрирован в ThreadPool.NET.16

#### Механика работы IOCP

1. **Инициация:** При создании FileStream с флагом FileOptions.Asynchronous (или useAsync: true), операционная система открывает файл с флагом FILE\_FLAG\_OVERLAPPED. Хендл файла ассоциируется с портом завершения.17
2. **Выполнение:** При вызове ReadAsync, драйвер немедленно возвращает статус, и если операция не завершена мгновенно (данные не в кэше), возвращается код ERROR\_IO\_PENDING. Поток возвращается в пул.
3. **Завершение:** Когда DPC драйвера завершает работу, он помещает пакет завершения (completion packet) в очередь IOCP.
4. **Пробуждение:** Потоки пула.NET (специальные I/O completion threads) ожидают на порту. Свободный поток "просыпается", забирает результат и выполняет продолжение задачи (Task Continuation).18

Такая модель обеспечивает идеальную масштабируемость: количество потоков не зависит от количества одновременных операций ввода-вывода, а зависит только от количества ядер CPU, необходимых для обработки результатов.

### 3.2. Linux: Эволюция от epoll к io\_uring

Ситуация в Linux исторически была сложнее. Традиционные механизмы select, poll и epoll отлично работали для сетевых сокетов, но плохо подходили для дискового ввода-вывода. POSIX AIO (Asynchronous I/O) имел множество ограничений и часто сваливался в блокирующий режим, что делало его ненадежным для высоконагруженных систем.21

#### Революция io\_uring

С появлением ядер Linux серии 5.x и внедрением подсистемы io\_uring, ситуация радикально изменилась. io\_uring представляет собой пару кольцевых буферов (Ring Buffers) в разделяемой памяти между пространством пользователя и ядром:

* **Submission Queue (SQ):** Приложение помещает запросы на ввод-вывод в эту очередь.
* **Completion Queue (CQ):** Ядро помещает результаты в эту очередь.21

Главное преимущество io\_uring — возможность работы в режиме пулинга (polling mode), что позволяет выполнять операции ввода-вывода вообще без системных вызовов (zero-syscall overhead). В этом режиме ядро периодически проверяет SQ на наличие новых задач, а приложение проверяет CQ на наличие результатов. Это устраняет дорогостоящие переключения контекста и делает Linux I/O конкурентоспособным или даже превосходящим Windows IOCP.23

### 3.3. Реализация в.NET: Стратегии FileStream

Начиная с.NET 6, реализация FileStream была полностью переписана. Была введена внутренняя абстракция FileStreamStrategy, которая выбирает оптимальную реализацию в рантайме:

* **Windows:** Используется AsyncWindowsFileStreamStrategy, базирующаяся на Overlapped I/O и IOCP.
* **Linux:** Используется UnixFileStreamStrategy. В более новых версиях планируется и частично внедряется поддержка io\_uring через IoUringStrategy, если ядро ОС это поддерживает.24

Важно отметить, что вся логика буферизации была вынесена в отдельную обертку BufferedFileStreamStrategy. Это означает, что низкоуровневые стратегии (Windows/Unix) могут фокусироваться исключительно на взаимодействии с ОС, не заботясь о пользовательских буферах.25

## 4. Оптимизация буферизации: Битва за память и кэш LOH

Выбор правильного размера буфера — это баланс между количеством системных вызовов и нагрузкой на сборщик мусора (GC). Стандартный размер буфера в 4 КБ (4096 байт) является историческим наследием и субоптимален для современных задач копирования больших файлов.1

### 4.1. Large Object Heap (LOH) и фрагментация

В.NET управление памятью разделено на поколения. Объекты размером 85,000 байт и более (порог LOH) аллоцируются в специальной куче больших объектов (Large Object Heap).26

* **Проблема:** Память в LOH не подвергается компактному сжатию (дефрагментации) при обычных сборках мусора (Gen 2 GC), так как перемещение больших блоков памяти слишком затратно. Это приводит к фрагментации адресного пространства. Если приложение часто создает и выбрасывает буферы размером > 85 КБ (например, 128 КБ), LOH быстро заполняется "дырами", что может привести к OutOfMemoryException даже при наличии свободной физической памяти.
* **Решение:** Использование буферов размером **81,920 байт** (80 КБ) является индустриальным стандартом для.NET. Этот размер достаточно велик, чтобы минимизировать количество системных вызовов, но при этом остается ниже порога LOH (85 КБ), попадая в Gen 0/1/2, где сборщик мусора эффективно управляет памятью и устраняет фрагментацию.1

### 4.2. Таблица сравнительной эффективности размеров буфера

Ниже представлена таблица, анализирующая влияние размера буфера на различные аспекты системы при копировании больших файлов.

| **Размер буфера** | **Влияние на Syscalls** | **Нагрузка на GC** | **Эффективность кэша CPU** | **Сценарий использования** |
| --- | --- | --- | --- | --- |
| **4 КБ (Default)** | Экстремально высокое (много переключений) | Низкая (SOH) | Высокая (помещается в L1) | Случайный доступ (Random Access), мелкие файлы. |
| **64 КБ - 80 КБ** | Оптимальное (баланс) | Низкая (SOH) | Средняя (L2/L3) | Последовательное чтение/запись (Stream I/O). Рекомендованный стандарт. |
| **1 МБ+** | Низкое | Высокая (LOH) | Низкая (вымывает кэш) | Работа с гигабайтными файлами. Требует ArrayPool или RecyclableMemoryStream. |
| **0 (No Buffer)** | 1 к 1 (прямой маппинг) | N/A | Зависит от приложения | Direct I/O, базы данных. Требует выравнивания по сектору диска. |

### 4.3. Флаг FileOptions.SequentialScan

При последовательном чтении файлов критически важно использовать флаг FileOptions.SequentialScan. Этот флаг транслируется в FILE\_FLAG\_SEQUENTIAL\_SCAN в Windows, давая подсказку (hint) менеджеру кэша операционной системы. ОС, видя этот флаг, активирует агрессивный алгоритм упреждающего чтения (read-ahead), загружая данные в оперативную память до того, как приложение их запросит. Это позволяет превратить дисковый I/O в операции доступа к памяти (Memory Mapped I/O), что дает прирост производительности на порядок.1

## 5. Блокировки и конкурентный доступ: FileShare deep dive

Одной из самых частых причин сбоев в распределенных системах является нарушение доступа к файлам (Sharing Violation). В отличие от баз данных с их транзакциями и уровнями изоляции, файловая система предлагает довольно примитивные, но строгие механизмы блокировок, поведение которых фундаментально различается между Windows и Linux.

### 5.1. Mandatory vs Advisory Locking

Это различие часто становится источником багов при портировании.NET приложений на Linux (Docker/Kubernetes).

* **Windows (Mandatory Locking):** Блокировки являются обязательными. Если процесс открыл файл с флагом FileShare.None (эксклюзивный доступ), ядро ОС физически запретит любому другому процессу (даже с правами администратора) читать или писать в этот файл. Попытка доступа вызовет исключение IOException (Sharing Violation).28 Флаги FileShare в.NET транслируются напрямую в параметры dwShareMode функции Win32 API CreateFile.30
* **Linux (Advisory Locking):** Традиционно в Unix-системах блокировки носят рекомендательный характер (advisory). Процессы могут использовать системные вызовы flock или fcntl, чтобы "договориться" о блокировке. Однако, если сторонний процесс (например, текстовый редактор или утилита cat) захочет прочитать файл, ядро его не остановит, так как эти утилиты обычно не проверяют наличие рекомендательных блокировок. В.NET на Linux реализация FileShare пытается эмулировать поведение Windows через flock, но это работает только между процессами, которые "играют по правилам" (используют тот же механизм блокировки).32

### 5.2. Проблема Kernel Lock Contention

Даже если используется FileShare.ReadWrite, позволяющий множественный доступ, возникает проблема конкуренции на уровне структур ядра. Файловая система (например, NTFS) использует внутренние примитивы синхронизации (FCB locks — File Control Block locks, ERESOURCE) для защиты целостности метаданных файла (размера, времени модификации, списка кластеров).

При высокой конкуренции (сотни потоков пытаются писать в один лог-файл) потоки начинают проводить больше времени в ожидании спинлоков ядра, чем выполняя запись. Это явление называется Kernel Lock Contention. Диагностика такой проблемы требует использования инструментов уровня ядра (ETW, Performance Counters) и часто проявляется как высокая загрузка CPU в режиме ядра (System Time) при низкой пропускной способности диска.34

## 6. Практическая реализация: Асинхронный копировщик

Задание первого дня требует реализации утилиты копирования файлов. Анализ стандартных инструментов и проектирование собственного решения требуют учета всех вышеописанных теоретических аспектов.

### 6.1. Деконструкция File.Copy

Стандартный метод System.IO.File.Copy является удобной абстракцией, но он имеет критический недостаток для интерактивных или высоконагруженных приложений — он блокирующий.

* **В Windows:** Метод вызывает Win32 API функцию CopyFile (или CopyFileEx). Эта функция выполняется в контексте вызывающего потока и не возвращает управление до полного завершения копирования. Для файла размером 10 ГБ это может означать "зависание" потока на минуты.36
* **В Linux:** Реализация.NET (начиная с Core 3.1) пытается использовать системный вызов copy\_file\_range.37 Этот вызов чрезвычайно эффективен, так как позволяет выполнить копирование данных внутри ядра (in-kernel copy) или даже внутри дискового массива (copy offload), минуя пересылку данных в User Space. Однако API File.Copy остается синхронным.

Для реализации требований задания (прогресс-бар, отмена через CancellationToken) необходимо отказаться от File.Copy в пользу ручного цикла чтения-записи.

### 6.2. Реализация паттерна Async Copy

Корректная реализация асинхронного копирования должна использовать FileStream с явно заданными параметрами для активации IOCP/io\_uring.

**Ключевые требования к реализации:**

1. **Буфер:** Использовать буфер 80 КБ (81920 байт) для баланса между syscalls и GC.
2. **Флаги:** Использовать FileOptions.Asynchronous и FileOptions.SequentialScan.
3. **Управление памятью:** Использовать ArrayPool<byte>.Shared.Rent(81920) для аренды буфера, чтобы избежать аллокаций в куче при каждом вызове.
4. **Цикл:**  
   C#  
   // Псевдокод асинхронного цикла  
   byte buffer = ArrayPool<byte>.Shared.Rent(bufferSize);  
   try {  
    int bytesRead;  
    while ((bytesRead = await sourceStream.ReadAsync(buffer, token)) > 0) {  
    await destinationStream.WriteAsync(buffer.AsMemory(0, bytesRead), token);  
    progress?.Report(bytesRead);  
    }  
   } finally {  
    ArrayPool<byte>.Shared.Return(buffer);  
   }

Этот подход обеспечивает полный контроль над процессом, позволяет прервать операцию в любой момент (через token) и информировать пользователя о прогрессе, не блокируя поток пула.39

### 6.3. Контекст синхронизации и ConfigureAwait

В задании указано создание консольной утилиты. Важно понимать различие в поведении await в разных типах приложений.

* **Консоль и Worker Service:** SynchronizationContext.Current по умолчанию равен null. Это означает, что продолжение асинхронного метода (код после await) будет выполнено на произвольном потоке пула (ThreadPool Thread). Вызов ConfigureAwait(false) в данном случае технически избыточен, так как поведение не изменится.41
* **UI (WPF/WinForms) и Legacy ASP.NET:** Существует SynchronizationContext, привязанный к конкретному потоку (UI Thread или Request Thread). Без ConfigureAwait(false) продолжение будет пытаться вернуться в этот контекст, что может вызвать дедлок (если контекст заблокирован ожиданием) или снижение производительности.

Несмотря на то, что в консольном приложении это не критично, использование ConfigureAwait(false) в библиотечном коде (например, в классе FileCopier, который может быть переиспользован в WPF) является строгим требованием профессиональной гигиены кода. Это предотвращает неявный захват контекста и гарантирует предсказуемое поведение компонента в любой среде.43

## 7. Управление процессами и паттерн Envdir (Пропедевтика)

Хотя основной фокус первого дня — потоки, понимание взаимодействия процессов с файловой системой подготавливает почву для следующих дней.

### 7.1. Наследование дескрипторов

При запуске дочерних процессов через System.Diagnostics.Process, они могут наследовать открытые дескрипторы (handles) файлов родительского процесса, если это явно не запрещено. Это частая причина "фантомных" блокировок файлов: родительский процесс закрыл файл, но не может его удалить, так как дочерний процесс (о существовании которого разработчик мог забыть) удерживает унаследованный хендл. Понимание флагов наследования (Inheritance Flags) в Win32 API и их проекции в.NET важно для написания надежных инструментов управления процессами.27

### 7.2. Паттерн Envdir

Паттерн envdir, пришедший из мира Unix (daemontools), демонстрирует использование файловой системы как универсального API конфигурации. Каждый файл в директории соответствует переменной окружения: имя файла — ключу, содержимое — значению. При старте процесса приложение считывает эту директорию и формирует блок переменных окружения (Environment Block) для нового процесса.

В.NET работа с переменными окружения процесса осуществляется через ProcessStartInfo.Environment. Важно помнить, что изменение System.Environment влияет только на текущий процесс, тогда как для дочернего необходимо модифицировать коллекцию StartInfo до вызова Process.Start.8

## 8. Заключение

Анализ теоретических основ первого дня пятой недели демонстрирует, что переход к Enterprise-разработке требует глубокого погружения в системное программирование. Эффективная работа с вводом-выводом — это не просто вызов методов API, а оркестрация взаимодействия между Managed Code, CLR, драйверами ядра и аппаратным обеспечением.

Освоение асинхронного ввода-вывода (IOCP/io\_uring), понимание стоимости системных вызовов, грамотное управление буферизацией (избегание LOH, использование ArrayPool) и знание тонкостей блокировок файловой системы (Mandatory vs Advisory) являются фундаментом для построения высокопроизводительных, масштабируемых и надежных серверных приложений. Эти компетенции отличают разработчика, способного писать код, работающий "по инерции", от инженера, создающего системы, выдерживающие экстремальные нагрузки реального мира.

#### Источники

1. План обучения работе с файлами и конфигурацией
2. Debug ThreadPool Starvation - .NET - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/core/diagnostics/debug-threadpool-starvation>
3. Diagnosing thread pool exhaustion issues in .NET Core apps - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/shows/on-dotnet/diagnosing-thread-pool-exhaustion-issues-in-net-core-apps>
4. User space and kernel space - Wikipedia, дата последнего обращения: декабря 4, 2025, <https://en.wikipedia.org/wiki/User_space_and_kernel_space>
5. What is the difference between the kernel space and the user space? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/5957570/what-is-the-difference-between-the-kernel-space-and-the-user-space>
6. What is difference between User space and Kernel space? - Unix & Linux Stack Exchange, дата последнего обращения: декабря 4, 2025, <https://unix.stackexchange.com/questions/87625/what-is-difference-between-user-space-and-kernel-space>
7. Architecting Containers Part 1: Why Understanding User Space vs. Kernel Space Matters, дата последнего обращения: декабря 4, 2025, <https://www.redhat.com/en/blog/architecting-containers-part-1-why-understanding-user-space-vs-kernel-space-matters>
8. What's the cost (in cycles) to switch between Windows Kernel and User mode?, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/1368061/whats-the-cost-in-cycles-to-switch-between-windows-kernel-and-user-mode>
9. Context Switching Performance — User vs Kernel modes | by Gowtham Tamilarasan, дата последнего обращения: декабря 4, 2025, <https://medium.com/@gtamilarasan/context-switching-performance-user-vs-kernel-modes-0984c61092bc>
10. Infographics: Operation Costs in CPU Clock Cycles - IT Hare on Soft.ware, дата последнего обращения: декабря 4, 2025, <http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/>
11. The Cost of Avoiding a Meltdown - ScyllaDB, дата последнего обращения: декабря 4, 2025, <https://www.scylladb.com/2018/01/07/cost-of-avoiding-a-meltdown/>
12. Meltdown and Spectre Performance - SUSE, дата последнего обращения: декабря 4, 2025, <https://www.suse.com/c/meltdown-spectre-performance/>
13. Servicing an Interrupt - Windows drivers - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/windows-hardware/drivers/wdf/servicing-an-interrupt>
14. Understanding the Windows I/O System | Microsoft Press Store, дата последнего обращения: декабря 4, 2025, <https://www.microsoftpressstore.com/articles/article.aspx?p=2201309&seqNum=3>
15. Understanding Deferred Procedure Calls (DPCs) for Windows Vulnerability Research & Exploit Development | by WaterBucket | Medium, дата последнего обращения: декабря 4, 2025, <https://medium.com/@WaterBucket/understanding-deferred-procedure-calls-dpcs-in-windows-ecd138292883>
16. I/O Completion Ports - Win32 apps - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/windows/win32/fileio/i-o-completion-ports>
17. win32file.CreateFile, дата последнего обращения: декабря 4, 2025, <https://mhammond.github.io/pywin32/win32file__CreateFile_meth.html>
18. How do Completion Port Threads of the Thread Pool behave during async I/O in .NET / .NET Core? - Codemia, дата последнего обращения: декабря 4, 2025, <https://codemia.io/knowledge-hub/path/how_do_completion_port_threads_of_the_thread_pool_behave_during_async_io_in_net__net_core>
19. Asynchronous I/O in C#: I/O Completion Ports - A sea of code, дата последнего обращения: декабря 4, 2025, <https://dschenkelman.github.io/2013/10/29/asynchronous-io-in-c-io-completion-ports/>
20. How do Completion Port Threads of the Thread Pool behave during async I/O in .NET / .NET Core? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/58057749/how-do-completion-port-threads-of-the-thread-pool-behave-during-async-i-o-in-ne>
21. How io-uring implementation is different from AIO? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/65075339/how-io-uring-implementation-is-different-from-aio>
22. The rapid growth of io\_uring - LWN.net, дата последнего обращения: декабря 4, 2025, <https://lwn.net/Articles/810414/>
23. IoRing vs. io\_uring: a comparison of Windows and Linux implementations, дата последнего обращения: декабря 4, 2025, <https://windows-internals.com/ioring-vs-io_uring-a-comparison-of-windows-and-linux-implementations/>
24. Implement io\_uring support for FileStream · Issue #51985 · dotnet/runtime - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/issues/51985>
25. runtime/src/libraries/System.Private.CoreLib/src/System/IO/Strategies/BufferedFileStreamStrategy.cs at main · dotnet/runtime · GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/IO/Strategies/BufferedFileStreamStrategy.cs>
26. Large object heap (LOH) on Windows - .NET | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/large-object-heap>
27. FileStream performance guidelines · dotnet runtime · Discussion #74405 - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/discussions/74405>
28. File locking in windows & linux [closed] - Super User, дата последнего обращения: декабря 4, 2025, <https://superuser.com/questions/1711476/file-locking-in-windows-linux>
29. CreateFile, дата последнего обращения: декабря 4, 2025, <https://www.cs.rpi.edu/academics/courses/fall09/os/win32/CreateFile.html>
30. CreateFileA function (fileapi.h) - Win32 apps | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea>
31. Creating and Opening Files - Win32 apps | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/windows/win32/fileio/creating-and-opening-files>
32. Platform-dependent FileStream permissions behavior? · Issue #24432 · dotnet/runtime, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/issues/24432>
33. Unable To Read Locked File In Linux · Issue #34126 · dotnet/runtime - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/issues/34126>
34. Hang observed due to NTFS!NtfsAcquireFileForCcFlush - OSR Developer Community, дата последнего обращения: декабря 4, 2025, <https://community.osr.com/t/hang-observed-due-to-ntfs-ntfsacquirefileforccflush/58338>
35. inode lock contention | Bobby Durrett's DBA Blog, дата последнего обращения: декабря 4, 2025, <https://www.bobbydurrettdba.com/2013/05/10/inode-lock-contention/>
36. Provide a File.CopyAsync method · Issue #20697 · dotnet/runtime - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/corefx/issues/17306>
37. Linux syscalls: advantage of copy\_file\_range over sendfile?, дата последнего обращения: декабря 4, 2025, <https://unix.stackexchange.com/questions/771238/linux-syscalls-advantage-of-copy-file-range-over-sendfile>
38. copy\_file\_range(2) - Linux manual page - man7.org, дата последнего обращения: декабря 4, 2025, <https://man7.org/linux/man-pages/man2/copy_file_range.2.html>
39. FileStream.CopyToAsync(Stream, Int32, CancellationToken) Method (System.IO), дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.filestream.copytoasync?view=net-9.0>
40. Asynchronous File Copy/Move in C# - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/882686/asynchronous-file-copy-move-in-c-sharp>
41. What are the SynchronizationContext differences between Console, Windows Forms, WPF?, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/69184874/what-are-the-synchronizationcontext-differences-between-console-windows-forms>
42. Why the default SynchronizationContext is not captured in a Console App? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/52686862/why-the-default-synchronizationcontext-is-not-captured-in-a-console-app>
43. ConfigureAwait and SynchronizationContext in C# - GeeksforGeeks, дата последнего обращения: декабря 4, 2025, <https://www.geeksforgeeks.org/c-sharp/synchronizationcontext-and-configureawait-in-c-sharp/>