# Архитектурные примитивы высоконадежного логирования и управления файловыми операциями в корпоративных системах.NET

## 1. Введение: Эпистемология персистентности в распределенных средах

Фундаментальный переход от разработки клиентских приложений реального времени, таких как игровые движки Unity, к проектированию высоконагруженных серверных систем на платформе.NET Enterprise, требует радикального пересмотра подходов к подсистемам ввода-вывода (I/O). В детерминированном игровом цикле (Game Loop) состояние приложения часто является мутабельным, локальным и жестко привязанным к частоте кадров; доступ к диску рассматривается как блокирующая операция, допустимая лишь в моменты загрузки сцен, и состояние часто не требует мгновенной, транзакционной фиксации на носителе.1 Напротив, архитектура современного бэкенда на базе ASP.NET Core диктует требования абсолютной асинхронности, отсутствия состояния (statelessness) и способности обрабатывать тысячи конкурентных запросов в секунду, сохраняя при этом строгие гарантии долговечности данных (durability).

Данный отчет представляет собой исчерпывающее теоретическое руководство и справочный материал для пятого дня пятой недели программы обучения, сфокусированный на теме «Ротация логов и атомарные операции». Документ синтезирует механические различия между ядром Windows NT и ядром Linux (в частности, в аспектах файловых блокировок и управления инодами), анализирует иерархию памяти, участвующую в сбросе буферов (flushing), и детально рассматривает паттерны асинхронной координации, необходимые для развязки (decoupling) задержек ввода-вывода и пропускной способности приложения.

Критическим императивом данной предметной области является надежность. В распределенной системе лог-файл часто является единственным артефактом истины (single source of truth) в момент сбоя. Следовательно, механизмы, используемые для записи, ротации и сохранения этих логов, должны быть устойчивы к перебоям питания, аварийному завершению процессов и конкуренции за ресурсы. Анализ, проведенный на основе предоставленных материалов, показывает, что поверхностного использования классов File или StreamWriter недостаточно для уровня Senior Backend Developer. Требуется глубокое понимание того, как среда выполнения (CLR) взаимодействует с драйверами файловой системы, как работают алгоритмы атомарного переименования и какие стратегии обеспечивают целостность данных при высокой конкурентной нагрузке.

## 2. Механика файлового ввода-вывода: От пространства пользователя к пространству ядра

Понимание производительности и надежности операций логирования начинается с осознания стоимости пересечения границы между пользовательским кодом (managed code) и ядром операционной системы. Каждая операция записи в лог инициирует сложную цепочку событий, проходящую через множество слоев абстракции.

### 2.1. Иерархия буферизации и гарантии сохранности данных

Распространенным заблуждением среди разработчиков является мнение, что успешное выполнение метода stream.WriteAsync() или File.AppendAllText() гарантирует физическое сохранение данных на диске. В действительности данные проходят через глубокую иерархию буферов, каждый из которых вносит свою задержку и свои риски потери информации.1

1. **Буфер приложения (User Space):** При использовании высокоуровневых библиотек логирования или структур вроде StringBuilder, строки накапливаются в управляемой куче (Managed Heap). Вызов метода записи лишь копирует данные в массив байтов внутри объекта потока. Это операция в оперативной памяти, происходящая за наносекунды.
2. **Внутренний буфер FileStream (Runtime Buffer):** Класс FileStream в.NET по умолчанию использует внутренний буфер размером 4 КБ (4096 байт).1 Это сделано для минимизации количества системных вызовов. Данные накапливаются здесь до тех пор, пока буфер не заполнится или не будет вызван метод Flush(). Если приложение аварийно завершится (крах процесса) до сброса этого буфера, данные будут безвозвратно утеряны.
3. **Страничный кэш ОС (Kernel Page Cache):** Когда FileStream сбрасывает свой буфер (через системный вызов write в Linux или WriteFile в Windows), данные копируются в пространство ядра, в страничный кэш (Page Cache).
   * *Второй порядок понимания:* На этом этапе операционная система сообщает приложению об успешном завершении операции записи. Однако данные все еще находятся в волатильной (энергозависимой) оперативной памяти. Ядро ОС берет на себя ответственность за их "ленивую" запись на физический диск (write-back caching).2 В Linux этот механизм оптимизирован для производительности: запись в память происходит мгновенно, но физический сброс может быть отложен на секунды.
4. **Аппаратный кэш диска (Disk Write Cache):** Современные SSD и HDD оснащены собственным DRAM-кэшем (от 64 МБ до нескольких ГБ) для переупорядочивания операций записи и оптимизации износа (wear leveling). Даже когда ОС отправляет данные контроллеру диска, они могут задержаться в этом аппаратном кэше.
5. **Физический носитель (Platter/NAND):** Только когда магнитная головка меняет полярность домена на пластине или заряд плавающего затвора фиксируется в ячейке NAND-памяти, данные становятся по-настоящему персистентными.

### 2.2. Семантика Flush() против Flush(true)

В контексте разработки надежных систем логирования критически важно различать методы сброса буферов, так как они обеспечивают разные уровни консистентности.3

* **stream.Flush() (без параметров):** Данный метод очищает *внутренний буфер.NET* и передает данные в *страничный кэш ОС*. Это обеспечивает так называемую "Согласованность приложения" (Application Consistency). Если приложение упадет, данные останутся в памяти ОС и будут записаны позже. Этого достаточно для большинства сценариев прикладного логирования, так как это быстрая операция.
* **stream.Flush(true):** Этот метод инициирует принудительную фиксацию данных на диске, обеспечивая "Согласованность при сбое" (Crash Consistency).
  + **В Windows:** Вызывается API FlushFileBuffers. Это заставляет ОС сбросить страничный кэш для данного файлового дескриптора *и* отправляет команду контроллеру диска на очистку аппаратного кэша.5
  + **В Linux:** Вызывается системный вызов fsync (или fdatasync), который блокирует поток до тех пор, пока устройство хранения не подтвердит физическую запись данных.6

**Влияние на производительность:** Использование Flush(true) является чрезвычайно дорогостоящей операцией. Если запись в страничный кэш занимает микросекунды, то fsync может занимать миллисекунды, так как требует физического взаимодействия с медленным носителем. На вращающихся дисках (HDD) это может ограничить скорость записи до 100-200 операций в секунду. Поэтому в высоконагруженных системах Flush(true) применяется только для журналов транзакций (WAL) в базах данных или критически важных аудиторских записей, но не для обычных текстовых логов.5

## 3. Теоретические основы атомарности в файловых системах

Для реализации корректной ротации логов — процесса архивации текущего файла и создания нового без потери данных — необходимо глубокое понимание концепции атомарности на уровне файловой системы.

### 3.1. Определение атомарной операции

В контексте баз данных (ACID) атомарность гарантирует, что транзакция либо выполняется целиком, либо не выполняется вовсе. В файловых системах атомарность означает неделимость операции. Наблюдатели системы (другие потоки или процессы) могут видеть систему либо в состоянии "до операции", либо в состоянии "после", но никогда в промежуточном, несогласованном состоянии.9

Примером неатомарной операции является перезапись файла путем его открытия, очистки содержимого и записи новых байтов. Если питание пропадет в середине этого процесса, файл окажется поврежденным (содержащим частичные данные). Атомарный подход подразумевает создание нового файла во временной директории и последующий вызов системной функции замены (rename/replace), которая подменяет указатели в файловой системе за одну инструкцию.

### 3.2. Атомарность метаданных против атомарности данных

Важно различать атомарность операций над *метаданными* и долговечность *данных*.

* **Атомарность метаданных:** Операции вроде rename (в Linux) или MoveFile (в Windows) изменяют структуру директорий. На журналируемых файловых системах (NTFS, ext4, XFS) эти операции являются атомарными. Журнал файловой системы записывает намерение выполнить перемещение, гарантируя, что даже при сбое операция будет либо завершена, либо откатана при восстановлении.10
* **Долговечность данных:** Хотя само *переименование* атомарно, *содержимое* файла может все еще находиться в кэше. Атомарность гарантирует, что имя файла указывает на правильную структуру данных (иноду), но не гарантирует, что сами данные записаны на магнитный диск.

### 3.3. Ограничения границ томов (Volume Boundary)

Фундаментальным ограничением атомарных перемещений является граница логического раздела (тома). Атомарное переименование возможно только в пределах одной файловой системы (mount point).

Перемещение файла из C:\Logs на D:\Archives или из /var/log в /mnt/backup не является обновлением метаданных. Это операция "Копирование + Удаление" (Copy-then-Delete).

Последовательность Copy-Delete не является атомарной. Сбой после копирования, но до удаления, приводит к дублированию данных. Сбой во время копирования приводит к появлению частичного файла. Для ротации логов это означает, что архивация (смена имени current.log на archive.log) должна происходить строго на том же диске, где файл был создан. Перенос в долгосрочное хранилище должен выполняться отдельным асинхронным процессом.11

## 4. Кроссплатформенные модели блокировок и конкуренции

Одной из сложнейших задач при переходе на.NET Core является различие в моделях блокировки файлов между Windows и Linux. Алгоритм ротации, идеально работающий на рабочей станции разработчика (Windows), часто приводит к исключениям IOException или потере данных при развертывании в Docker-контейнере на базе Linux.

### 4.1. Модель Windows: Обязательная блокировка (Mandatory Locking)

Windows использует модель **обязательной блокировки**. Если процесс открывает файл без явного разрешения совместного доступа, ядро ОС накладывает блокировку. Ни один другой процесс не может читать, писать или удалять этот файл, пока дескриптор не будет закрыт.2

* **Проблема "File In Use":** Это классическая ошибка в Windows-среде. Строгое соблюдение блокировок означает, что писатель имеет эксклюзивный контроль, но это создает хрупкость. Если антивирус, индексер или агент сбора логов (Filebeat) откроет файл для чтения, попытка приложения выполнить ротацию (переименование) будет отклонена с ошибкой доступа UnauthorizedAccessException или IOException.13

### 4.2. Модель Linux: Консультативная блокировка и Иноды

Linux (и POSIX-системы в целом) использует **консультативную блокировку** (advisory locking) и систему инодов (inodes) с подсчетом ссылок.

* **Иноды (Inodes):** В Linux имя файла — это всего лишь жесткая ссылка (hard link) на инод (index node), структуру данных, описывающую файл и расположение его блоков на диске.
  + **Unlink (Удаление):** Когда выполняется команда удаления файла (unlink), удаляется запись в директории (имя). Однако, если какой-либо процесс держит этот файл открытым, счетчик ссылок инода остается положительным. Файл продолжает существовать на диске, и процесс может продолжать писать в него. Пространство освобождается только тогда, когда закрывается последний файловый дескриптор.14
  + **Rename (Переименование):** Переименование открытого файла просто меняет имя, указывающее на инод. Процесс, удерживающий дескриптор, продолжает писать в тот же самый физический файл (тот же инод), даже если его имя изменилось. Это делает ротацию логов в Linux тривиальной: можно переименовать app.log в app.log.1 прямо во время записи. Приложение продолжит писать в то, что теперь называется app.log.1, пока само не решит закрыть и открыть файл заново.15

### 4.3. Глубокий анализ перечисления System.IO.FileShare

Перечисление FileShare управляет тем, как FileStream взаимодействует с этими механизмами ОС. Понимание FileShare обязательно для реализации паттернов конкурентного логирования.16

| **Флаг FileShare** | **Поведение в Windows** | **Поведение в Linux** | **Сценарий использования** |
| --- | --- | --- | --- |
| **None** | Эксклюзивный доступ. Никто другой не может даже читать. | Запрашивает эксклюзивную блокировку (flock LOCK\_EX). | Базы данных, файлы состояния. |
| **Read** | Другие могут читать, но не писать. | Запрашивает разделяемую блокировку (flock LOCK\_SH). | Конфигурационные файлы. |
| **ReadWrite** | Полный доступ для всех. Опасно без синхронизации. | Консультативно. Разрешает все. | Общие логи (требует мьютексов). |
| **Delete** | **Критично:** Разрешает переименование/удаление открытого файла. | По умолчанию в Linux (unlink/rename разрешены всегда), но влияет на flock. | **Обязательно для ротации логов.** |

Императив FileShare.Delete:

В Windows, чтобы эмулировать поведение Linux и позволить ротацию файла, в который идет запись, необходимо открывать файл с флагами FileShare.Read | FileShare.Write | FileShare.Delete.

Если флаг FileShare.Delete опущен, вызов File.Move выбросит исключение, так как файл занят процессом записи. Устанавливая этот флаг, приложение сообщает ядру Windows: "Я пишу в этот файл, но если кто-то (включая меня самого в другом потоке) захочет его переименовать или удалить — разрешаю".

Нюанс: В Windows, даже с FileShare.Delete, файл не исчезает с диска окончательно, пока дескриптор не закрыт. Он переходит в состояние "delete pending", имя удаляется из пространства имен, предотвращая новые открытия, но данные остаются доступными текущему владельцу дескриптора.12

### 4.4. Изменения в.NET 6+ для Unix

До версии.NET 6, реализация FileStream в Unix пыталась эмулировать строгие блокировки Windows, выбрасывая исключения, если блокировку flock не удавалось получить. Это приводило к проблемам совместимости. В.NET 6 поведение было смягчено для соответствия стандартам POSIX. В частности, открытие файла с FileAccess.Read теперь пытается получить разделяемую блокировку, но не падает с фатальной ошибкой, если это невозможно, полагаясь на консультативную природу механизма. Это изменение уменьшает количество "ложных срабатываний" при блокировках, но требует от разработчиков более тщательной реализации синхронизации на уровне приложения.19

## 5. Паттерн "Производитель-Потребитель" и асинхронное логирование

Прямая запись в файл из потока обработки HTTP-запроса (например, внутри Контроллера) является архитектурным анти-паттерном. Это жестко связывает латентность (latency) ответа пользователю со скоростью работы дисковой подсистемы. Если диск "подвиснет" (stall) или очередь записи переполнится, поток из пула (ThreadPool) заблокируется, что приведет к исчерпанию пула потоков (Thread Starvation) и отказу в обслуживании.1

### 5.1. Архитектура разделения ответственности

Стандартом индустрии для высоконагруженных систем является паттерн **Producer-Consumer** (Производитель-Потребитель).

* **Производители (Producers):** Потоки обработки запросов формируют лог-сообщение (структуру или строку) и помещают его в быструю потокобезопасную очередь в оперативной памяти. Эта операция занимает микросекунды и не зависит от диска.
* **Потребитель (Consumer):** Единственный выделенный фоновый поток (или Task) вычитывает сообщения из очереди и выполняет тяжелые операции ввода-вывода (открытие файла, запись, сброс буфера, ротация). Это обеспечивает сериализацию доступа к файлу, устраняя необходимость в сложных блокировках файла между потоками.

### 5.2. Эволюция коллекций: BlockingCollection vs System.Threading.Channels

Исторически в.NET использовалась BlockingCollection<T>, обертывающая ConcurrentQueue<T>. Однако она предоставляет синхронный блокирующий API (Take()), который может занимать поток в ожидании данных.

System.Threading.Channels (введенные в.NET Core 3.0) являются более совершенным примитивом для этой задачи.21

* **Асинхронность:** Каналы (Channels) спроектированы для работы с async/await. Метод Reader.ReadAsync() позволяет циклу потребителя освобождать поток, когда очередь пуста, возвращая его в ThreadPool для полезной работы, вместо того чтобы блокировать его в ожидании.23
* **Обратное давление (Backpressure) и Bounded Channels:** Если диск работает медленно, а логи генерируются быстрее, чем записываются, очередь будет расти до тех пор, пока не закончится память (OOM). Ограниченные каналы (Bounded Channel) решают эту проблему через механизм обратного давления. Когда канал заполнен, попытка записи Writer.WriteAsync() может либо асинхронно ожидать (замедляя производителей и, следовательно, входящий трафик), либо отбрасывать сообщения.21
  + *Стратегия:* Для логирования обычно используется стратегия BoundedChannelFullMode.DropWrite (или DropOldest). В критических ситуациях лучше потерять часть логов, чем полностью остановить обработку бизнес-транзакций (платежей, заказов) из-за переполнения буферов логирования.24

### 5.3. Zero-Allocation: Оптимизация памяти

Для минимизации нагрузки на сборщик мусора (Garbage Collection), современные библиотеки логирования (например, Serilog, NLog, Microsoft.Extensions.Logging) используют System.Buffers и Span<T>. Вместо создания новой строки string для каждого сообщения (что вызывает аллокацию в куче), продвинутые реализации используют интерфейс IBufferWriter<byte> для форматирования сообщения (добавления метки времени, уровня лога) прямо в переиспользуемый буфер байтов, который затем передается в файловый поток. Использование System.Threading.Channels с передачей структур или сегментов памяти (Memory<T>) вместо строк является ключевым элементом оптимизации.1

## 6. Алгоритмы и стратегии ротации логов

Ротация логов — это процесс управления ростом файлов для предотвращения переполнения дискового пространства. Существует две основные стратегии: **основанная на размере** (Size-Based) и **основанная на времени** (Time-Based).

### 6.1. Стандартный алгоритм ротации (Rename-Create)

Это наиболее надежный паттерн для приложений.NET, использующий атомарность File.Move, описанную в разделе 3.

**Алгоритм:**

1. **Проверка условия:** Перед записью проверить, превышает ли размер current.log заданный лимит (например, 10 МБ).
2. **Синхронизация:** Убедиться, что запись не идет (гарантируется паттерном Producer-Consumer, так как писатель один).
3. **Закрытие потока:** Поток FileStream, пишущий в current.log, должен быть сброшен (Flush) и закрыт (Dispose). Это освобождает файловый дескриптор. На Windows это критически важно даже с FileShare.Delete для гарантии завершения всех операций.5
4. **Сдвиг архивов (Rotation Shift):** Выполнение каскадного переименования старых файлов.
   * Удалить app.log.5 (самый старый, если лимит 5 файлов).
   * Переместить app.log.4 -> app.log.5.
   * ...
   * Переместить app.log.1 -> app.log.2.
   * Переместить current.log -> app.log.1.
   * *Примечание:* Использование File.Move(src, dest, overwrite: true), доступного в.NET Core 3.0+, упрощает этот процесс, позволяя атомарно заменять целевой файл, хотя сдвиг обычно требует последовательного переименования.25
5. **Создание нового:** Создать новый FileStream для current.log.
6. **Возобновление записи:** Записать сообщение, вызвавшее ротацию, в новый файл.

**Анализ надежности:** Критическим моментом является Шаг 4. File.Move атомарен. Если приложение упадет во время шага 4, у вас может остаться app.log и app.log.1, но файл не будет поврежден. Логика сдвига при следующем запуске должна корректно обработать существующие файлы.

### 6.2. Анти-паттерн "Copy-Truncate"

Некоторые внешние инструменты (например, старые конфигурации logrotate в Linux) используют стратегию Copy-Truncate:

1. Копировать current.log в current.log.1.
2. Урезать (truncate) current.log до 0 байт.  
   Почему это опасно: Между копированием и урезанием существует временной зазор. Любые логи, записанные приложением в этот промежуток, будут потеряны (они попадут в файл после копирования, но будут уничтожены урезанием).27 В высоконагруженных приложениях.NET это гарантированно приводит к потере данных. Этот паттерн не рекомендуется к реализации внутри приложения.

### 6.3. Внешняя ротация (Linux logrotate)

В среде Linux часто делегируют ротацию демону logrotate.

* **Механизм:** logrotate переименовывает файл (app.log -> app.log.1).
* **Проблема сигнализации:** Поскольку Linux сохраняет инод открытым,.NET приложение продолжает писать в app.log.1 (старый инод), даже если имя файла изменилось.
* **Решение:** Приложение должно слушать Unix-сигнал (обычно SIGHUP). Когда logrotate перемещает файл, он посылает сигнал SIGHUP процессу.NET. Приложение должно перехватить этот сигнал, закрыть дескриптор файла и открыть его заново (reopen), что приведет к созданию нового файла app.log.27

## 7. Детальный план реализации: День 5

На основе представленной теоретической базы, ниже приведен детальный план реализации модуля RotationLogger.

### 7.1. Цель

Разработать потокобезопасный класс RotationLogger, который:

1. Принимает сообщения логов из множества потоков без блокировок вызывающего кода.
2. Пишет в файл, используя буферизированный FileStream.
3. Атомарно ротирует файлы при достижении лимита размера (например, 1 МБ).
4. Использует семантику FileShare.Delete для совместимости с Windows.
5. Реализует корректное завершение работы (Graceful Shutdown) для сброса очереди.

### 7.2. Ключевые компоненты реализации

#### A. Канал (Буфер)

Использовать Channel.CreateBounded<LogEntry> с емкостью, например, 1000 элементов.

* **Опция SingleReader = true:** Оптимизация, так как у нас только один фоновый поток-писатель.28
* **Опция AllowSynchronousContinuations = false:** Принудительное использование асинхронности, чтобы продолжения не захватывали потоки производителей.
* **Режим заполнения (FullMode):** DropOldest (приоритет стабильности системы над полнотой логов при экстремальной нагрузке) или Wait (если потеря логов недопустима).

#### B. Фоновый Писатель (Потребитель)

Фоновый сервис (IHostedService или Task.Run), выполняющий бесконечный цикл чтения:

C#

while (await reader.WaitToReadAsync()) {  
 while (reader.TryRead(out var log)) {  
 // 1. Проверка критериев ротации  
 if (currentFileSize > limit) await RotateLogAsync();  
   
 // 2. Запись в буфер  
 await fileStream.WriteAsync(log.Data);  
   
 // 3. Обновление счетчика размера  
 currentFileSize += log.Data.Length;  
 }  
 // 4. Оппортунистический сброс (Flush), когда очередь пуста  
 await fileStream.FlushAsync();   
}

*Инсайт:* Вызов FlushAsync() после каждой строки убивает производительность. Сброс только тогда, когда канал пуст (или по таймеру), обеспечивает наилучший баланс между скоростью и надежностью.8

#### C. Логика ротации (Atomic Move)

Метод RotateLogAsync должен выполнять следующую последовательность:

1. **Закрытие потока:** await fileStream.DisposeAsync(). Это обязательно для освобождения дескриптора.
2. **Политика повторных попыток (Polly):** Обернуть операцию перемещения в Retry Policy. На Windows антивирусы могут удерживать файл миллисекунды после закрытия дескриптора. Цикл повторов с экспоненциальной задержкой (Exponential Backoff) обязателен для предотвращения сбоев ротации.30
3. **Перемещение:**  
   C#  
   File.Move(currentPath, archivePath, overwrite: true);   
   // Использует MoveFileEx с флагом REPLACE\_EXISTING на Windows
4. **Переоткрытие:** Создать новый FileStream для оригинального пути. Убедиться, что флаги FileShare.Read и FileShare.Delete установлены, чтобы внешние утилиты (например, tail -f) могли читать файл во время записи.17

### 7.3. Кроссплатформенная конфигурация

Реализация должна корректно обрабатывать пути:

* Использовать Path.Combine и Path.DirectorySeparatorChar.
* Всегда использовать FileShare.ReadWrite | FileShare.Delete. На Linux это "разрешающее" поведение (permissive), на Windows — "включающее" (enabling functionality).
* Использовать Environment.NewLine или явный \n в зависимости от формата логов (для JSON-логов предпочтителен \n).

## 8. Заключение

Реализация системы ротации логов является микрокосмом более широких проблем бэкенд-инжиниринга. Она требует синтеза знаний о **ядре операционной системы** (иноды, файловые дескрипторы, страничный кэш), **паттернах конкурентности** (Producer-Consumer, Backpressure, Channels) и **инженерии надежности** (атомарные операции, идемпотентность, retry-политики).

Отказ от синхронного File.AppendAllText в пользу асинхронных, буферизированных и атомарных примитивов, предоставляемых пространствами имен System.IO и System.Threading.Channels в.NET, позволяет создавать системы, которые не только производительны, но и "механически симпатичны" (mechanically sympathetic) операционной системе. Этот фундамент необходим для масштабирования приложений, где поток логов может соперничать с бизнес-трафиком, и где целостность диагностических данных является первой линией обороны при инцидентах. Строгое применение флагов FileShare.Delete и понимание стоимости Flush(true) гарантирует, что система останется надежной как в среде разработки на Windows, так и в продакшн-контейнерах на Linux.

#### Источники

1. План обучения работе с файлами и конфигурацией
2. Internals of Windows and Linux File Systems | by Keyur Ramoliya | The Deep Hub - Medium, дата последнего обращения: декабря 4, 2025, <https://medium.com/thedeephub/internals-of-windows-and-linux-file-systems-498296e5e81f>
3. What are the differences between FileStream.Flush(), FlushAsync(), and Dispose()?, дата последнего обращения: декабря 4, 2025, <https://www.mindstick.com/interview/34113/what-are-the-differences-between-filestream-flush-flushasync-and-dispose>
4. FileStream.Flush Method (System.IO) | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.filestream.flush?view=net-10.0>
5. What's the difference between FileStream.Flush() and FileStream.Flush(True)? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/4921498/whats-the-difference-between-filestream-flush-and-filestream-flushtrue>
6. FileStream.Flush(true) doesn't flush device buffer on macOS, while it does so on Windows · Issue #28444 · dotnet/runtime - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/issues/28444>
7. Does FileStream.FlushAsync() ensure file is written to disk? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/78216282/does-filestream-flushasync-ensure-file-is-written-to-disk>
8. c# - StreamWriter flush() call performance impact - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/3300291/streamwriter-flush-call-performance-impact>
9. Is an atomic file rename (with overwrite) possible on Windows? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/167414/is-an-atomic-file-rename-with-overwrite-possible-on-windows>
10. c# - File.Move atomic operation - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/15274861/file-move-atomic-operation>
11. How to move a directory in C# .NET in a single atomic operation - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/683369/how-to-move-a-directory-in-c-sharp-net-in-a-single-atomic-operation>
12. Can using FileShare.Delete cause a UnauthorizedAccessException? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/19875329/can-using-fileshare-delete-cause-a-unauthorizedaccessexception>
13. File.Move (overwrite) unreliable because FileShare.Delete access of target file ignored · Issue #114230 · dotnet/runtime - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/issues/114230>
14. What is Linux doing differently that allows me to remove/replace files where Windows would complain the file is currently in use?, дата последнего обращения: декабря 4, 2025, <https://unix.stackexchange.com/questions/49299/what-is-linux-doing-differently-that-allows-me-to-remove-replace-files-where-win>
15. rewrite existing file so that it gets replaced by new version atomically, only once fully written, дата последнего обращения: декабря 4, 2025, <https://unix.stackexchange.com/questions/24395/rewrite-existing-file-so-that-it-gets-replaced-by-new-version-atomically-only-o>
16. System.IO.FileStream FileAccess vs FileShare - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/25097773/system-io-filestream-fileaccess-vs-fileshare>
17. FileShare Enum (System.IO) | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.fileshare?view=net-10.0>
18. Did the behaviour of deleted files open with FileShare.Delete change on Windows?, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/60424732/did-the-behaviour-of-deleted-files-open-with-fileshare-delete-change-on-windows>
19. Breaking change: FileStream locks files with shared lock on Unix - .NET | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/core/compatibility/core-libraries/6.0/filestream-file-locks-unix>
20. Platform-dependent FileStream permissions behavior? · Issue #24432 · dotnet/runtime, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/issues/24432>
21. Channels - .NET - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/channels>
22. Worth considering system.threading.channels? · Issue #63 · serilog/serilog-sinks-async, дата последнего обращения: декабря 4, 2025, <https://github.com/serilog/serilog-sinks-async/issues/63>
23. Building High-Performance .NET Apps With C# Channels - Anton DevTips, дата последнего обращения: декабря 4, 2025, <https://antondevtips.com/blog/building-high-performance-dotnet-apps-with-csharp-channels>
24. A Concise Guide to Asynchronous Data Flow with C# Channels - DEV Community, дата последнего обращения: декабря 4, 2025, <https://dev.to/petersaktor/a-concise-guide-to-asynchronous-data-flow-with-c-channels-45af>
25. File.Move Method (System.IO) | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.file.move?view=net-10.0>
26. c# - File.Move Does Not Work - File Already Exists - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/5920882/file-move-does-not-work-file-already-exists>
27. Changing LogRotate's default way of rotating the log files - Server Fault, дата последнего обращения: декабря 4, 2025, <https://serverfault.com/questions/594798/changing-logrotates-default-way-of-rotating-the-log-files>
28. ChannelOptions.SingleReader Property (System.Threading.Channels) | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.threading.channels.channeloptions.singlereader?view=net-10.0>
29. When should System.Threading.Channels be preferred to ConcurrentQueue?, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/76809859/when-should-system-threading-channels-be-preferred-to-concurrentqueue>
30. Is there an os-independent way to atomically overwrite a file? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/30385225/is-there-an-os-independent-way-to-atomically-overwrite-a-file>
31. Scaling Connections with BlockingCollection
32. Prevent deletion of a file on Linux - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/57702889/prevent-deletion-of-a-file-on-linux>