# Архитектура конкурентности и паттерны устойчивости в.NET

## Учебные материалы: Неделя 3, День 4

### Углубленная теория и анализ механизмов CLR

## Введение: Смена парадигмы — от синтаксиса к архитектуре

Четвертый день третьей недели обучения знаменует собой фундаментальный переход в мышлении разработчика: от написания асинхронного кода (использования async/await) к проектированию асинхронных систем. Если предыдущие этапы были сосредоточены на механике работы TaskScheduler и машине состояний, то текущий модуль фокусируется на **управляемой конкурентности** (Managed Concurrency) и **архитектурной устойчивости** (Resilience).

В современных высоконагруженных распределенных системах недостаточно просто запустить задачи параллельно. Ключевыми становятся вопросы:

* Как защитить систему от перегрузки собственными запросами (Self-Inflicted DDoS)?
* Как изолировать сбои во внешних сервисах, чтобы они не обрушили все приложение (Cascading Failures)?
* Как корректно остановить тысячи операций без утечек памяти и потери данных?
* Как организовать потокобезопасный обмен данными без блокировок, убивающих производительность?

Данный документ представляет собой исчерпывающее теоретическое руководство, охватывающее внутреннее устройство примитивов синхронизации, теорию очередей в контексте System.Threading.Channels, механику кооперативной отмены и паттерны проектирования Throttling и Bulkhead.

## Глава 1. Продвинутая синхронизация: Демистификация SemaphoreSlim

В мире многопоточности.NET существует множество инструментов синхронизации, но SemaphoreSlim является "рабочей лошадкой" для асинхронных сценариев. Чтобы понять, почему именно он стал стандартом де-факто для ограничения конкурентности, необходимо углубиться в эволюцию примитивов синхронизации и внутреннее устройство CLR.

### 1.1. Эволюция блокировок: Kernel Mode vs User Mode

Для осознанного выбора инструментов необходимо понимать различие между режимом ядра (Kernel Mode) и пользовательским режимом (User Mode).

1. Тяжеловесные примитивы (Kernel Mode):  
   Классический класс System.Threading.Semaphore (без суффикса Slim) является оберткой над объектом ядра операционной системы Windows (Win32 Semaphore object).1
   * **Механизм:** При каждом вызове WaitOne() происходит системный вызов (syscall). Это требует переключения контекста процессора из пользовательского режима в режим ядра. Эта операция дорога: она сбрасывает конвейер инструкций процессора и инвалидирует кэши (TLB, L1/L2), что может занимать тысячи тактов CPU.
   * **Преимущества:** Позволяет синхронизировать потоки между *разными процессами* (именованные семафоры).2
   * **Недостатки:** Неприемлемые накладные расходы для высокочастотных операций внутри одного процесса.
2. Гибридные примитивы (User Mode + Kernel Mode):  
   SemaphoreSlim, Monitor (lock) и ManualResetEventSlim пытаются избежать перехода в ядро, пока это возможно.
   * **Механизм:** Они работают в пользовательском пространстве памяти процесса CLR. При попытке захвата ресурса они сначала используют атомарные инструкции процессора (Interlocked), что на порядки быстрее системных вызовов.1

### 1.2. Анатомия SemaphoreSlim: Почему он "Slim"?

SemaphoreSlim называется "тонким" (slim), потому что он оптимизирован для минимизации потребления ресурсов и времени ожидания.

#### Механизм активного ожидания (SpinWait)

Когда поток вызывает WaitAsync на SemaphoreSlim, и счетчик семафора равен нулю (ресурс занят), поток не сразу "усыпает" (блокируется). Вместо этого запускается эвристический алгоритм **SpinWait**.4

Это критически важная оптимизация. Во многих сценариях ресурс освобождается буквально через несколько наносекунд после того, как другой поток попытался его захватить. Если бы ожидающий поток сразу уходил в глубокий сон (Context Switch), накладные расходы на его "пробуждение" превысили бы время самого ожидания.

Вместо этого SemaphoreSlim выполняет короткий цикл (spinning) — по сути, while(true), сжигая такты CPU в ожидании изменения переменной счетчика. Если ресурс освобождается во время спин-ожидания, захват происходит практически мгновенно, без участия планировщика ОС.5 Только если спин-цикл исчерпан (обычно это несколько микросекунд), поток переходит в полноценное ожидание.

#### Асинхронная природа (WaitAsync)

В отличие от lock (который привязан к потоку — Thread Affinity) и Semaphore (который блокирует поток), SemaphoreSlim предоставляет метод WaitAsync, возвращающий Task.7

Это фундаментальное отличие. WaitAsync не блокирует поток выполнения. Если семафор занят, метод возвращает незавершенную задачу. Поток, вызвавший метод, возвращается в пул потоков (ThreadPool) и может выполнять другую полезную работу. Когда слот в семафоре освобождается, продолжение (continuation) метода планируется к выполнению. Это позволяет серверу с небольшим количеством потоков обслуживать тысячи одновременных запросов, ожидающих доступа к ограниченному ресурсу (например, БД).7

### 1.3. Сравнительный анализ производительности

Исследования показывают существенную разницу в накладных расходах между классическим Semaphore и SemaphoreSlim.

| **Характеристика** | **System.Threading.Semaphore** | **System.Threading.SemaphoreSlim** | **Влияние на архитектуру** |
| --- | --- | --- | --- |
| **Базовая реализация** | Объект ядра ОС (Win32 Handle) | Объект CLR + SpinWait + Lazy Kernel Object | SemaphoreSlim значительно легче для GC и памяти. |
| **Стоимость захвата (свободный)** | ~1000+ тактов (Syscall) | ~10-20 тактов (Interlocked/Spin) | SemaphoreSlim предпочтителен для высоконагруженных циклов. |
| **Межпроцессное взаимодействие** | Поддерживается (Named semaphore) | Не поддерживается (Local only) | Используйте Semaphore только для IPC (Inter-Process Communication). |
| **Поддержка async/await** | Нет (Блокирует поток) | Да (WaitAsync) | SemaphoreSlim — единственный выбор для современного.NET.7 |
| **Потребление ресурсов** | Дескриптор ОС (ограниченный ресурс) | Память в куче (Managed Heap) | Можно создавать тысячи SemaphoreSlim без истощения ресурсов ОС. |

### 1.4. Паттерны применения и Best Practices

#### Ограничение параллелизма (Throttling)

Основной сценарий использования SemaphoreSlim — это Throttling. Если у вас есть список из 10,000 URL для загрузки, запуск Task.Run для каждого из них ("Parallelism") приведет к открытию 10,000 сокетов, исчерпанию портов и падению производительности из-за переключения контекста.

Использование SemaphoreSlim позволяет ограничить степень параллелизма (DOP - Degree of Parallelism) до разумного числа (например, 100).4

#### Гарантированное освобождение (The Finally Block)

Самая критическая ошибка при работе с семафорами — отсутствие гарантии вызова Release. Если в критической секции произойдет исключение, и Release будет пропущен, "разрешение" навсегда потеряется. Со временем счетчик семафора достигнет нуля, и приложение войдет в состояние **Deadlock** (точнее, Resource Starvation).4

**Корректный паттерн:**

C#

await semaphore.WaitAsync(cancellationToken);  
try  
{  
 // Критическая секция: работа с ресурсом  
 await AccessDatabaseAsync();  
}  
finally  
{  
 // Release должен быть вызван ГАРАНТИРОВАННО  
 semaphore.Release();  
}

#### Проблема "Sync over Async"

Никогда не используйте синхронный метод .Wait() или свойство .Result на задаче, возвращаемой semaphore.WaitAsync(). Это классический антипаттерн, который может привести к исчерпанию пула потоков. Всегда используйте await.9

## Глава 2. Паттерн Producer-Consumer и System.Threading.Channels

Если SemaphoreSlim управляет доступом к коду, то System.Threading.Channels управляет потоком данных. Это современная, высокопроизводительная реализация паттерна "Производитель-Потребитель", пришедшая на смену устаревшим подходам.

### 2.1. Почему BlockingCollection устарела?

До появления Channels (в.NET Core 3.0+) стандартом была BlockingCollection<T>. Однако она имеет фундаментальные недостатки для асинхронного мира:

1. **Блокирующая природа:** Методы Take и Add блокируют вызывающий поток. Это приемлемо для выделенных потоков (new Thread()), но губительно для ThreadPool.10
2. **Отсутствие нативной асинхронности:** Хотя можно использовать GetConsumingEnumerable, это не позволяет использовать await внутри цикла потребления так же эффективно, как await foreach.
3. **Производительность:** Бенчмарки показывают, что Channels работают значительно быстрее и создают меньше аллокаций памяти благодаря использованию ValueTask и оптимизированных структур данных.11

### 2.2. Архитектура System.Threading.Channels

Библиотека построена вокруг концепции "трубы" с двумя концами:

* **ChannelWriter:** Сторона для записи (Producer).
* **ChannelReader:** Сторона для чтения (Consumer).

Это разделение интерфейсов (Interface Segregation) позволяет четко разграничить ответственность компонентов: производитель не имеет доступа к методам чтения, и наоборот.13

### 2.3. Bounded vs Unbounded: Управление давлением (Backpressure)

Выбор типа канала определяет поведение системы под нагрузкой.

#### Unbounded Channel (Безграничный)

* **Поведение:** Емкость ограничена только оперативной памятью (Int32.MaxValue).
* **Риск:** Если производитель работает быстрее потребителя (Producer > Consumer), очередь будет бесконечно расти. Это приведет к увеличению потребления памяти (Memory Leak pattern) и, в конечном итоге, к OutOfMemoryException и падению процесса.14
* **Применение:** Только когда гарантировано, что потребитель быстрее, или всплески нагрузки кратковременны.

#### Bounded Channel (Ограниченный)

* **Поведение:** Имеет фиксированную емкость. Это ключевой инструмент для реализации **Backpressure** (Обратного давления).
* **Механизм:** Когда канал заполнен, метод WriteAsync не завершается немедленно. Он возвращает незавершенную задачу, которая завершится только тогда, когда потребитель освободит место в очереди. Это автоматически "притормаживает" производителя, синхронизируя скорость компонентов системы.13
* **FullMode стратегии:**
  + Wait (по умолчанию): Асинхронное ожидание места. Самый безопасный вариант для целостности данных.
  + DropNewest: Новое сообщение отбрасывается.
  + DropOldest: Удаляется самое старое сообщение (актуально для телеметрии, где важны только свежие данные).
  + DropWrite: Запись отменяется, производитель получает уведомление об ошибке.13

### 2.4. Внутренние оптимизации и Lock-Free алгоритмы

Одной из причин высокой производительности Channels является возможность настройки под конкретный сценарий использования через ChannelOptions.

SingleWriter / SingleReader:

Если разработчик знает, что у канала будет только один поток-писатель, он может установить SingleWriter = true.

* **Эффект:** Внутри канала отключаются тяжелые механизмы синхронизации (locks/Interlocked), необходимые для координации нескольких писателей. Код переключается на использование оптимизированных **Lock-Free** алгоритмов (или алгоритмов с минимальной синхронизацией).13
* **Результат:** Существенное снижение накладных расходов на запись/чтение и уменьшение конкуренции за кэш процессора (False Sharing).
* **Опасность:** Если установить SingleWriter = true, но писать из нескольких потоков, поведение не определено (undefined behavior) и данные могут быть повреждены без выбрасывания исключений.16

Синтез знаний:

Использование System.Threading.Channels с Bounded емкостью и корректно настроенными SingleReader/Writer позволяет создавать высокопроизводительные конвейеры обработки данных (Pipelines), которые устойчивы к всплескам нагрузки и эффективно используют ресурсы CPU.15

## Глава 3. Архитектура отмены: Механика CancellationToken

В распределенных системах "право передумать" является не роскошью, а необходимостью. Пользователь может закрыть браузер, или вышестоящий сервис может отменить запрос по таймауту..NET реализует модель **кооперативной отмены** (Cooperative Cancellation).

### 3.1. Философия кооперативной отмены

В отличие от жесткого убийства потока (Thread.Abort — который ныне устарел и не поддерживается в.NET Core), кооперативная отмена требует участия самого выполняемого кода.19

* **Инициатор:** Посылает сигнал через CancellationTokenSource (CTS).
* **Исполнитель:** Периодически проверяет CancellationToken и корректно завершает работу.

Это гарантирует целостность данных: блоки finally выполняются, ресурсы (using) освобождаются, транзакции откатываются.

### 3.2. Soft vs Hard Cancellation

Существует два паттерна реакции на отмену запроса.20

#### Hard Cancellation (Жесткая отмена)

Это стандартный подход для задач, которые не имеют смысла, если результат больше не нужен.

* **Метод:** token.ThrowIfCancellationRequested().
* **Поведение:** Если токен отменен, выбрасывается исключение OperationCanceledException.
* **Эффект:** Поток управления прерывается, стек разматывается (stack unwinding). Задача (Task) переходит в состояние Canceled.
* **Важно:** Никогда не следует "глотать" это исключение (catch (Exception)) внутри библиотеки, если вы не планируете скрыть факт отмены. Вызывающий код должен знать, что задача была именно отменена, а не упала с ошибкой.22

#### Soft Cancellation (Мягкая отмена)

Используется, когда отмена является частью нормальной логики, а не исключением.

* **Метод:** Проверка свойства token.IsCancellationRequested.
* **Поведение:** Цикл или метод корректно завершается и возвращает частичный результат или значение по умолчанию (null, 0, false).
* **Пример:** Фоновая индексация файлов. При отмене можно вернуть список уже проиндексированных файлов, а не выбрасывать ошибку.20

### 3.3. Ловушка LinkedTokenSource и Утечки Памяти

Одной из самых коварных проблем при работе с отменой является использование CancellationTokenSource.CreateLinkedTokenSource.

Этот метод создает новый токен, который срабатывает, если сработал *любой* из входных токенов (например, "отмена пользователем" ИЛИ "глобальный таймаут").

Механизм утечки:

Чтобы связанный токен узнал об отмене родительского, он должен подписаться на событие отмены родителя. Это означает, что LinkedTokenSource регистрирует колбэк (Register) внутри родительского токена.24

* **Сценарий:** У вас есть долгоживущий родительский токен (например, ApplicationStopping токен всего сервиса). На каждый HTTP-запрос вы создаете LinkedTokenSource (связывая его с таймаутом запроса).
* **Проблема:** Родительский токен хранит ссылки на ВСЕ созданные связанные токены через список делегатов обратного вызова. Поскольку родитель живет долго, GC не может собрать "короткоживущие" связанные токены, так как на них есть активная ссылка.
* **Результат:** Массивная утечка памяти (Memory Leak), где тысячи CancellationTokenSource забивают кучу.

**Решение:** CancellationTokenSource реализует IDisposable. Обязательно вызывать Dispose() для связанного CTS. Метод Dispose отписывает колбэк от родительского токена, разрывая связь и позволяя GC собрать мусор.24

C#

// Правильный паттерн: использование using  
using var linkedCts = CancellationTokenSource.CreateLinkedTokenSource(token1, token2);  
await DoWorkAsync(linkedCts.Token);  
// При выходе из блока using вызывается Dispose(), предотвращая утечку

## Глава 4. Паттерны устойчивости: Throttling и Bulkhead

Когда система находится под нагрузкой, задача инженера — обеспечить плавную деградацию (Graceful Degradation), а не катастрофический отказ. Для этого используются паттерны управления нагрузкой.

### 4.1. Throttling (Дросселирование)

Цель: Защитить собственный сервис от перегрузки входящими запросами.

Принцип: Закон Литтла (Little's Law) утверждает, что количество одновременных запросов в системе равно скорости поступления запросов, умноженной на время обработки. Если время обработки растет (БД тормозит), количество запросов растет экспоненциально.

Реализация: Throttling ограничивает количество одновременных операций. В.NET это реализуется через глобальный SemaphoreSlim. Если лимит превышен, новые запросы либо ждут в очереди (с таймаутом), либо немедленно отклоняются (HTTP 429 Too Many Requests) — это называется паттерном Fail Fast.7

Также существуют алгоритмы ограничения частоты (Rate Limiting), такие как **Token Bucket** и **Leaky Bucket**, которые ограничивают не количество одновременных задач, а количество запросов в единицу времени (RPS). В.NET 7+ для этого введен System.Threading.RateLimiting, но SemaphoreSlim остается базовым примитивом для ограничения конкурентности (Concurrency Limiting).28

### 4.2. Bulkhead Isolation (Переборки)

Цель: Защитить сервис от каскадных сбоев (Cascading Failures), вызванных отказом внешних зависимостей.30

Аналогия: Корабль разделен на водонепроницаемые отсеки (Bulkheads). Если пробоина в одном отсеке, вода не затапливает все судно.

Сценарий катастрофы:

Сервис зависит от "Базы Данных А" и "Внешнего API Б". Если "API Б" начинает отвечать очень медленно (latency spike), все потоки приложения (ThreadPool) могут оказаться заняты ожиданием ответа от "API Б". В результате, приложение перестает обрабатывать запросы даже к "Базе Данных А", которая абсолютно здорова. Произошло "заражение" всей системы сбоем одного компонента.32

Решение:

Изолировать ресурсы для каждой зависимости.

* Создать отдельный SemaphoreSlim (или изолированный ThreadPool) для вызовов к "API Б".
* Создать отдельный SemaphoreSlim для вызовов к "БД А".

Теперь, если "API Б" тормозит, исчерпается только его личный семафор. Запросы к нему начнут отваливаться (Fail Fast), но семафор "БД А" останется свободным, и эта часть функционала продолжит работать идеально. Система теряет только часть функциональности (Partial Failure), но остается живой.30

**Таблица: Сравнение стратегий защиты**

| **Характеристика** | **Throttling** | **Bulkhead Isolation** |
| --- | --- | --- |
| **Вектор угрозы** | Высокая входящая нагрузка (DDoS, Marketing spikes). | Отказ или замедление внутренней/внешней зависимости. |
| **Механизм защиты** | Ограничение общего входа (Input Gate). | Сегментация ресурсов (Resource Partitioning). |
| **Инструмент.NET** | Глобальный SemaphoreSlim или RateLimiter. | Множество локальных SemaphoreSlim (по одному на зависимость) или Polly Policies. |
| **Результат при сбое** | Отказ в обслуживании лишних клиентов (429). | Отключение только сбойной фичи, сохранение ядра системы. |

## Глава 5. Практический синтез: Архитектура Smart Downloader

Все изученные концепции объединяются в лабораторной работе дня — создании отказоустойчивого загрузчика данных.

### 5.1. Архитектурный разбор задачи

Необходимо реализовать сервис, который загружает список URL, ограничивая параллелизм (например, не более 5 одновременных загрузок), с возможностью отмены всей операции.

**Компоненты решения:**

1. Ограничитель параллелизма (Throttling):  
   Используем SemaphoreSlim(5). Каждый рабочий поток перед началом загрузки должен получить "билет" через await semaphore.WaitAsync(ct). Передача токена отмены в WaitAsync критически важна: если пользователь нажмет "Стоп", ожидающие потоки должны немедленно выйти из очереди, а не ждать освобождения слота.33
2. Поток данных (Producer-Consumer):  
   Используем Channel.CreateUnbounded<DownloadResult>().
   * **Producers:** Множество задач (Tasks), запускаемых через Task.Run или Select, которые пишут результаты загрузки в канал.
   * **Consumer:** Один цикл await foreach, который читает из канала и сохраняет данные/обновляет UI. Это развязывает UI от сетевых задержек.
3. Обработка ошибок и отмены:  
   Используем блок try-catch-finally внутри каждой задачи загрузки.
   * catch (OperationCanceledException): Корректно обрабатываем отмену, не роняя весь процесс.
   * finally { semaphore.Release(); }: **Железобетонное правило**. Семафор должен быть освобожден даже если сеть упала, диск переполнен или токен отменен. Иначе — утечка слотов и зависание загрузчика.4
4. LinkedTokenSource:  
   Для реализации таймаута на всю пачку загрузок создаем связанный токен: CancellationTokenSource.CreateLinkedTokenSource(userToken, timeoutToken). Не забываем обернуть его в using для предотвращения утечек памяти!

### 5.2. Концептуальный код

C#

public async Task SmartDownloadAsync(List<string> urls, int maxConcurrency, CancellationToken ct)  
{  
 // 1. Создаем ограничитель (Throttling / Bulkhead для сети)  
 using var semaphore = new SemaphoreSlim(maxConcurrency);  
   
 // 2. Создаем канал для результатов  
 var channel = Channel.CreateUnbounded<string>();  
  
 // 3. Запускаем задачи (Producers)  
 var tasks = urls.Select(async url =>   
 {  
 // Входим в семафор. Если отмена сработала, выходим сразу.  
 await semaphore.WaitAsync(ct);  
 try  
 {  
 // Проверка перед долгой операцией  
 ct.ThrowIfCancellationRequested();  
   
 // Сама работа (сетевой запрос)  
 // Передаем токен внутрь HttpClient для прерывания сокета  
 var content = await httpClient.GetStringAsync(url, ct);  
   
 // Пишем в канал  
 await channel.Writer.WriteAsync($"Success: {url}", ct);  
 }  
 catch (OperationCanceledException)  
 {  
 // Логируем отмену ("Soft cancellation" для одной задачи)  
 }  
 catch (Exception ex)  
 {  
 await channel.Writer.WriteAsync($"Error: {url} - {ex.Message}", ct);  
 }  
 finally  
 {  
 // ВСЕГДА освобождаем слот  
 semaphore.Release();  
 }  
 });  
  
 // 4. Паттерн "Завершение канала"  
 // Когда все задачи завершатся (успешно или нет), закрываем канал для записи  
 \_ = Task.WhenAll(tasks).ContinueWith(\_ => channel.Writer.Complete());  
  
 // 5. Потребитель (Consumer)  
 // Читаем результаты по мере их поступления  
 await foreach (var result in channel.Reader.ReadAllAsync(ct))  
 {  
 Console.WriteLine(result);  
 }  
}

Этот подход демонстрирует профессиональный уровень владения асинхронностью: код неблокирующий, ресурсоэффективный (благодаря SemaphoreSlim и Channels), устойчивый к сбоям (изоляция через try-catch) и корректно управляемый (через CancellationToken).

### Заключение

Четвертый день обучения является поворотным моментом. Мы переходим от изучения *инструментов* к изучению *систем*. Понимание того, как SemaphoreSlim экономит такты процессора через SpinWait, почему Channels превосходят коллекции на блокировках, и как Bulkhead спасает микросервисы от цепной реакции — это фундамент для построения надежных Enterprise-приложений на платформе.NET.

#### Источники

1. Semaphore and SemaphoreSlim - .NET - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/threading/semaphore-and-semaphoreslim>
2. Semaphore vs SeamphoreSlim Micro Benchmark | March | 2013 | Articles - CodeWithStu, дата последнего обращения: декабря 3, 2025, <https://im5tu.io/article/2013/03/semaphore-vs-seamphoreslim-micro-benchmark/>
3. How do I choose between Semaphore and SemaphoreSlim? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/4154480/how-do-i-choose-between-semaphore-and-semaphoreslim>
4. SemaphoreSlim Class (System.Threading) - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.threading.semaphoreslim?view=net-10.0>
5. Does SemaphoreSlim's Wait(Int32) method return immediately when passed zero?, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/14574181/does-semaphoreslims-waitint32-method-return-immediately-when-passed-zero>
6. I found a performance bug in SemaphoreSlim : r/dotnet - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/dotnet/comments/2y2avx/i_found_a_performance_bug_in_semaphoreslim/>
7. Efficient Synchronization in C# with SemaphoreSlim - Oleg Kyrylchuk, дата последнего обращения: декабря 3, 2025, <https://okyrylchuk.dev/blog/efficient-synchronization-in-csharp-with-semaphoreslim/>
8. C# Semaphoreslim (How It Works For Developers) - IronPDF, дата последнего обращения: декабря 3, 2025, <https://ironpdf.com/blog/net-help/csharp-semaphoreslim/>
9. SemaphoreSlim in .NET, a practical guide with the rest of the toolbox - DEV Community, дата последнего обращения: декабря 3, 2025, <https://dev.to/stevsharp/semaphoreslim-in-net-a-practical-guide-with-the-rest-of-the-toolbox-1mh7>
10. BlockingCollection vs. Channels in C#: What's the Difference? | by erhan355 - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@erhan355/blockingcollection-vs-channels-in-c-whats-the-difference-8b742fc332f4>
11. Asynchronous Producer Consumer Pattern in .NET (C#) - DotNetCurry.com, дата последнего обращения: декабря 3, 2025, <https://www.dotnetcurry.com/dotnetcore/1509/async-dotnetcore-pattern>
12. Performance Showdown of Producer/Consumer (Job Queues) Implementations in C# .NET, дата последнего обращения: декабря 3, 2025, <https://michaelscodingspot.com/performance-of-producer-consumer/>
13. Channels - .NET - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/channels>
14. How to implement Producer/Consumer with System.Threading.Channels - David Guida, дата последнего обращения: декабря 3, 2025, <https://www.davidguida.net/how-to-implement-producer-consumer-with-system-threading-channels/>
15. Building High-Performance .NET Apps With C# Channels - Anton DevTips, дата последнего обращения: декабря 3, 2025, <https://antondevtips.com/blog/building-high-performance-dotnet-apps-with-csharp-channels>
16. Implement Producer/Consumer patterns using Channel in C# - Saigon Technology, дата последнего обращения: декабря 3, 2025, <https://saigontechnology.com/blog/implement-producerconsumer-patterns-using-channel-in-c/>
17. What is System.Threading.Channels - Concept and Usage | KoderShop, дата последнего обращения: декабря 3, 2025, <https://kodershop.com/blog/technical-tutorials-10/introduction-to-system-threading-channels-359>
18. Job Offloading Pattern with System.Threading.Channels. A way to deal with long-running tasks in .NET - nikiforovall.blog, дата последнего обращения: декабря 3, 2025, <https://nikiforovall.blog/dotnet/async/2024/04/21/job-offloading-pattern.html>
19. CancellationToken: The Complete Technical Guide for .NET Developers - DEV Community, дата последнего обращения: декабря 3, 2025, <https://dev.to/iron-software/-cancellationtoken-the-complete-technical-guide-for-net-developers-1h7p>
20. Why are cancellations handled as exceptions? Aren't they expected in many cases? - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/dotnet/comments/1jlz4v6/why_are_cancellations_handled_as_exceptions_arent/>
21. Best Practice: "Soft Cancellation" with gRPC - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/71092559/best-practice-soft-cancellation-with-grpc>
22. A Deep Dive into C#'s CancellationToken | by Mitesh Shah | Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@mitesh_shah/a-deep-dive-into-c-s-cancellationtoken-44bc7664555f>
23. c# - Elegantly handle task cancellation - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/12633903/elegantly-handle-task-cancellation>
24. Hidden catch when using linked CancellationTokenSource, дата последнего обращения: декабря 3, 2025, <https://lowleveldesign.wordpress.com/2015/11/30/catch-in-cancellationtokensource/>
25. Memory leak from linked cancellation token registrations · Issue #78180 · dotnet/runtime, дата последнего обращения: декабря 3, 2025, <https://github.com/dotnet/runtime/issues/78180>
26. Linked CancellationTokenSource memory leaks | by Tim Lovell-Smith | Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@parchandri/linked-cancellationtokensource-memory-leaks-b5c92e7d2953>
27. Throttling pattern - Azure Architecture Center | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/azure/architecture/patterns/throttling>
28. Rate Limiter using Leaky Bucket Algorithm | by Anil Goyal - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@anil.goyal0057/rate-limiter-using-leaky-bucket-algorithm-ea22b17bace6>
29. Locking in .NET — lock vs System.Threading.Lock vs SemaphoreSlim - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@manobesh1982_54603/locking-in-net-lock-vs-system-threading-lock-vs-semaphoreslim-3d821502e3f6>
30. Bulkhead Pattern for Robust Software Systems in C#, дата последнего обращения: декабря 3, 2025, <https://www.c-sharpcorner.com/article/bulkhead-pattern-for-robust-software-systems-in-c-sharp/>
31. Bulkhead pattern - Azure Architecture Center - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead>
32. Example of bulkheads in .NET micro services - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/55836600/example-of-bulkheads-in-net-micro-services>
33. Semaphore slim to handle throttling per time period - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/67576459/semaphore-slim-to-handle-throttling-per-time-period>