# Глубокий анализ архитектуры безопасности данных и примитивов синхронизации в среде.NET (Теоретический модуль третьего дня)

## 1. Введение: Фундаментальный сдвиг парадигмы к конкурентной безопасности

Переход от однопоточного мышления, характерного для паттерна "Game Loop" или простых последовательных алгоритмов, к конкурентной многопоточной архитектуре требует радикального пересмотра подходов к управлению состоянием памяти.1 Если первые два дня обучения были посвящены механике создания задач и синтаксическому сахару асинхронности, то третий день представляет собой критический этап, посвященный **корректности** и **безопасности** данных.

В многопоточной среде корректность выполнения программы больше не гарантируется простым порядком следования инструкций в исходном коде. В дело вступают физические ограничения аппаратного обеспечения, оптимизации компилятора и алгоритмы планировщика операционной системы. Основная цель данного теоретического модуля — предоставить исчерпывающий анализ механизмов защиты разделяемого состояния (Shared State), разобрать анатомию состояний гонки (Race Conditions) на уровне ассемблера и микроархитектуры процессора, а также детально рассмотреть внутреннее устройство примитивов синхронизации CLR (Common Language Runtime), включая эволюционные изменения, представленные в.NET 9.1

Этот документ предназначен для глубокого изучения теоретической базы, необходимой для выполнения практических лабораторных работ третьего дня, и служит эталонным справочником по низкоуровневой синхронизации в экосистеме.NET.

## 2. Физика повреждения данных: Состояния гонки и атомарность

В основе проблематики многопоточного программирования лежат два фундаментальных понятия: атомарность операций и видимость памяти. Понимание этих концепций требует деконструкции высокоуровневых абстракций языка C# до уровня машинных инструкций и циклов процессора.

### 2.1. Иллюзия атомарности в управляемом коде

Одной из самых распространенных и опасных ментальных ловушек для разработчиков является предположение об атомарности простых арифметических операций. Классический пример — операция инкремента i++. С точки зрения синтаксиса C#, это единое выражение. Однако, семантически и физически, это сложная последовательность действий, известная как **Read-Modify-Write (RMW)**.1

На уровне языка ассемблера (x86/x64) операция i++ для поля класса транслируется в следующую последовательность микроопераций:

1. **Load (Чтение):** Значение переменной загружается из оперативной памяти (или L1/L2 кэша) в регистр процессора (например, MOV EAX, [Address]).
2. **Modify (Модификация):** Процессор выполняет арифметическую операцию над значением в регистре (например, INC EAX).
3. **Store (Запись):** Обновленное значение из регистра записывается обратно в ячейку памяти (MOV [Address], EAX).4

#### Анатомия состояния гонки (Race Condition)

Состояние гонки возникает, когда планировщик операционной системы (OS Scheduler) прерывает выполнение потока посередине этой последовательности RMW.3

Рассмотрим сценарий с двумя потоками (Thread A и Thread B), оперирующими над общей переменной counter = 0:

1. **Thread A** выполняет **Load**: читает 0 в свой регистр.
2. *Контекстное переключение (Context Switch)*: ОС приостанавливает Thread A и передает управление Thread B.5
3. **Thread B** выполняет **Load**: также читает 0 (так как Thread A еще не записал результат).
4. **Thread B** выполняет **Modify** (0 -> 1) и **Store** (записывает 1 в память).
5. *Контекстное переключение*: Управление возвращается Thread A.
6. **Thread A** восстанавливает свое состояние (в его регистре все еще 0), выполняет **Modify** (0 -> 1) и **Store** (записывает 1 в память).

В результате двух инкрементов переменная равна 1 вместо ожидаемых 2. Это явление называется **"Lost Update" (Потерянное обновление)**.6 Без механизмов синхронизации невозможно гарантировать целостность данных даже при простейших операциях.

### 2.2. Гарантии атомарности в спецификации ECMA CLI

Спецификация Common Language Infrastructure (ECMA-335) дает строгие определения того, какие операции чтения и записи являются атомарными по своей природе 7:

* Чтение и запись переменных ссылочных типов (class, interface, delegate) всегда атомарны.
* Чтение и запись примитивных типов данных, размер которых не превышает разрядность процессора (Native Word Size), атомарны. Это bool, char, byte, sbyte, short, ushort, uint, int, float.

**Критическое ограничение:** Для 64-битных типов данных (long, ulong, double) на 32-битных системах атомарность **не гарантируется**. Операция записи long (64 бита) на 32-битном процессоре выполняется как две инструкции записи по 32 бита (MOV DWORD PTR для младшей и старшей частей).8 Если контекстное переключение произойдет между этими двумя инструкциями, другой поток может прочитать **"разорванное значение" (Torn Read)** — комбинацию старших бит нового значения и младших бит старого значения (или наоборот), создавая число, которое никогда логически не существовало в программе.9

### 2.3. Аппаратная реализация атомарности: Interlocked

Для решения проблем RMW и обеспечения атомарности на аппаратном уровне.NET предоставляет класс System.Threading.Interlocked.4 Методы этого класса транслируются JIT-компилятором непосредственно в специальные инструкции процессора с префиксом блокировки.

* **Архитектура x86/x64:** Метод Interlocked.Increment использует инструкцию с префиксом LOCK (например, LOCK XADD или LOCK INC).11 Этот префикс заставляет процессор утвердить сигнал LOCK# на шине управления (в старых архитектурах) или, что более распространено в современных CPU, использовать протокол когерентности кэша (например, MESI) для захвата эксклюзивного владения соответствующей кэш-линией (Cache Line Locking).13 Это гарантирует, что ни одно другое ядро не сможет прочитать или записать в эту область памяти до завершения инструкции.
* **Архитектура ARM:** Традиционно ARM использовала пару инструкций LDREX (Load-Exclusive) и STREX (Store-Exclusive) для реализации атомарности через механизм LL/SC (Load-Link / Store-Conditional).15 Однако, начиная с.NET 9, среда выполнения оптимизирована для использования более новых атомарных инструкций LSE (Large System Extensions), таких как LDADD, которые обеспечивают лучшую масштабируемость на многоядерных серверных процессорах (например, AWS Graviton).16

#### Compare-And-Swap (CAS)

Метод Interlocked.CompareExchange реализует фундаментальный паттерн синхронизации **CAS**, на котором строятся все алгоритмы без блокировок (lock-free).18 Он атомарно сравнивает значение в памяти с ожидаемым и, если они равны, заменяет его новым. В x86 это инструкция CMPXCHG с префиксом LOCK.12 Это позволяет строить оптимистичные алгоритмы, которые не блокируют поток, а повторяют попытку (spin-loop) в случае неудачи обновления.20

## 3. Модели памяти и архитектурные барьеры

Второй критический аспект теории третьего дня — это модели памяти (Memory Models) и переупорядочивание инструкций.1 Если атомарность касается *целостности* записи, то модель памяти определяет *порядок* и *видимость* этих записей между разными процессорами.

### 3.1. Иерархия переупорядочивания (Reordering)

Разработчики часто ошибочно полагают, что код выполняется в том порядке, в котором он написан. В реальности оптимизация происходит на трех уровнях 21:

1. **Компилятор (C# / Roslyn):** Может переупорядочивать независимые инструкции, но делает это редко.
2. **JIT-компилятор (RyuJIT):** Агрессивно оптимизирует код. Например, **Loop Hoisting** может вынести чтение переменной за пределы цикла. Если у вас есть код while (!stop) { }, JIT может преобразовать его в if (!stop) { while (true) { } }, кэшируя значение stop в регистре. Если другой поток изменит stop в памяти, данный поток никогда об этом не узнает и уйдет в бесконечный цикл.21
3. **Процессор (Hardware):** Современные CPU используют суперскалярную архитектуру, конвейеризацию и внеочередное исполнение (Out-of-Order Execution). Записи в память сначала попадают в **Store Buffer** (буфер записи), чтобы процессор не ждал медленной оперативной памяти. Из-за этого запись, выполненная на одном ядре, может стать видимой другим ядрам с задержкой, нарушая последовательную согласованность.21

### 3.2. Ключевое слово volatile и семантика барьеров

Ключевое слово volatile в C# — это инструкция компилятору и среде выполнения отключить определенные оптимизации и установить барьеры памяти (Memory Barriers).25 Оно навязывает семантику **Acquire/Release**:

* **Volatile Write (Release Semantics):** Гарантирует, что все операции записи и чтения, предшествующие этой записи в коде, будут завершены и станут видимыми *до* того, как станет видна сама volatile-запись. Инструкции не могут "перешагнуть вниз" через volatile-запись.26
* **Volatile Read (Acquire Semantics):** Гарантирует, что чтение произойдет *до* любых последующих операций чтения или записи. Инструкции не могут "перешагнуть вверх" через volatile-чтение.26

Важно отметить, что volatile не гарантирует строгой атомарности RMW-операций (как Interlocked), а лишь обеспечивает упорядочивание и свежесть чтения.28

### 3.3. Различия архитектур: x86 vs ARM64

Понимание различий между "сильной" и "слабой" моделями памяти критично для кроссплатформенной разработки.NET.29

#### x86/x64: Сильная модель (Strong Memory Model)

Архитектура Intel/AMD использует модель **TSO (Total Store Order)**. В этой модели:

* Записи (Stores) не переупорядочиваются с другими записями.
* Чтения (Loads) не переупорядочиваются с другими чтениями.
* Единственное возможное переупорядочивание — это Store за которым следует Load (из-за Store Buffer).30  
  В результате, на x86 многие ошибки конкурентности "маскируются" железом. Код может работать корректно даже без volatile (хотя это не гарантируется спецификацией), потому что процессор сам обеспечивает строгий порядок.31

#### ARM64: Слабая модель (Weak/Relaxed Memory Model)

Архитектура ARM (используемая в мобильных устройствах, Apple Silicon, AWS Graviton) позволяет процессору переупорядочивать чтения и записи практически произвольно, если между ними нет явной зависимости по данным. Без использования барьеров памяти (volatile или lock) потоки могут видеть изменения данных в хаотичном порядке.33

**Эволюция в.NET:**

* В ранних версиях.NET на ARM для реализации volatile JIT генерировал инструкцию **DMB (Data Memory Barrier)**. Это "полный барьер", который заставляет процессор сбросить конвейер и дождаться завершения всех операций памяти. Это очень дорогая операция с точки зрения производительности.21
* **Оптимизация в.NET 9:** В.NET 9 JIT-компилятор для ARM64 был существенно улучшен. Вместо тяжелых барьеров DMB, теперь используются специализированные инструкции **LDAR (Load-Acquire)** и **STLR (Store-Release)**.16 Эти инструкции являются "однонаправленными" барьерами: они запрещают переупорядочивание только в одну сторону (вход в критическую секцию), что значительно дешевле и эффективнее, чем полная остановка конвейера.36 Это делает volatile операции на ARM64 в.NET 9 значительно быстрее по сравнению с x64 реализациями, где барьеры часто реализованы через более дорогие инструкции LOCK или MFENCE.35

## 4. Глубокое погружение: Monitor и механизм блокировки

Конструкция lock в C# является синтаксическим сахаром для класса System.Threading.Monitor.1 Внутренняя реализация этого механизма представляет собой гибридную блокировку, оптимизированную для различных сценариев нагрузки.

### 4.1. Внутренняя структура объекта CLR (Object Header)

Каждый ссылочный объект в куче.NET имеет скрытые накладные расходы. Перед данными объекта находятся два машинных слова:

1. **Method Table Pointer:** Указатель на описание типа (Type Metadata).
2. **Object Header (Заголовок объекта):** 4 байта (на 32-бит) или 8 байт (на 64-бит). Это битовая маска, используемая для хранения хэш-кода, битов фиксации GC (pinning) и, что наиболее важно для нас, информации о блокировке.39

### 4.2. Механизм инфляции блокировки (Lock Inflation)

CLR использует адаптивную стратегию блокировки, чтобы избежать дорогих обращений к ядру ОС, когда это возможно. Процесс разделяется на два этапа: "Thin Lock" и "Fat Lock".41

#### Thin Lock (Тонкая блокировка)

Когда поток пытается захватить блокировку (Monitor.Enter) на объекте, который свободен, CLR пытается использовать **Thin Lock**.

* Вся информация о блокировке сохраняется непосредственно в **Object Header**.
* Используется атомарная инструкция CompareExchange для записи идентификатора текущего потока (Thread ID) в заголовок объекта.
* Никакой памяти не выделяется, никакие объекты ядра ОС не создаются.
* Это чрезвычайно быстрая операция (десятки наносекунд), выполняемая полностью в пользовательском режиме (User Mode).43
* Заголовок объекта имеет сложную битовую структуру. Например, младшие биты могут кодировать, является ли блокировка тонкой, есть ли хэш-код, и каков уровень рекурсии (сколько раз один и тот же поток вошел в lock).40

#### Inflation (Инфляция) и Fat Lock (Толстая блокировка)

Инфляция (превращение тонкой блокировки в толстую) происходит в трех случаях:

1. **Конкуренция (Contention):** Если поток пытается захватить блокировку, которая уже удерживается другим потоком, и спиннинг (spinning) не помог.
2. **Хэш-код:** Если у объекта был вызван метод GetHashCode(), хэш-код сохраняется в заголовке объекта. Места для Thread ID больше не остается, так как биты заняты хэш-кодом.
3. **Переполнение рекурсии:** Если счетчик рекурсии переполняет отведенные ему биты в заголовке.40

При инфляции CLR выполняет следующие действия:

1. Выделяет структуру **SyncBlock** из глобальной таблицы SyncBlock Table.
2. В Object Header объекта записывается индекс (SyncBlock Index), указывающий на запись в этой таблице.
3. SyncBlock содержит полноценные примитивы синхронизации ядра ОС (в Windows это Event-объекты или Семафоры), списки ожидания потоков и полный 32-битный хэш-код.41

**Стоимость Fat Lock:** Использование Fat Lock значительно дороже, так как при ожидании (Wait) поток должен перейти в режим ядра (Kernel Mode), что вызывает контекстное переключение (Context Switch), стоящее тысячи процессорных тактов.38

### 4.3. Адаптивный спиннинг (Adaptive Spinning)

Перед тем как "раздуть" блокировку (inflate) и усыпить поток через ядро ОС, Monitor пытается выполнить **адаптивный спиннинг**. Поток некоторое время крутится в цикле (busy-wait), проверяя, не освободилась ли блокировка. Это основано на предположении, что блокировка удерживается недолго, и ожидание в цикле дешевле, чем стоимость переключения контекста.

* Длительность спиннинга динамически подстраивается (адаптируется) в зависимости от количества процессоров и истории успешных захватов на данном объекте.
* На одноядерных машинах спиннинг отключается, так как он бессмысленен (занимающий блокировку поток не может выполняться одновременно со ждущим).40

## 5. System.Threading.Lock: Эволюция в.NET 9

В.NET 9 был представлен новый тип System.Threading.Lock, который призван устранить архитектурные недостатки классического Monitor и блокировок на произвольных объектах (lock(new object())).2

### 5.1. Проблемы классического Monitor

Использование object в качестве объекта синхронизации имеет ряд недостатков:

* **Полиморфизм заголовка:** Так как любой объект может быть заблокирован, CLR вынуждена проверять заголовок *каждого* объекта при доступе, чтобы понять, является ли он блокировкой, хэш-кодом или указателем на SyncBlock. Это накладные расходы на каждое обращение.
* **GC Overhead:** SyncBlock'и являются внешними структурами, которые усложняют работу сборщика мусора (GC), требуя поддержания связей между управляемой кучей и таблицей SyncBlock.47
* **Отсутствие типобезопасности:** Можно случайно заблокировать публично доступный объект (например, this или Type), что может привести к дедлокам, если сторонний код также заблокирует этот объект.38

### 5.2. Архитектура System.Threading.Lock

Новый класс Lock не использует заголовок объекта для хранения состояния синхронизации. Вместо этого вся необходимая информация (Thread ID, счетчики, состояние ожидания) инкапсулирована внутри полей самого класса Lock.2

* **Отсутствие инфляции:** Поскольку класс Lock специализирован для синхронизации, ему не нужно "раздуваться" из заголовка объекта. Он всегда готов к использованию.
* **Ref Struct Scope:** Метод EnterScope() возвращает структуру ref struct Scope, которая реализует паттерн Dispose. Это позволяет использовать конструкцию using для гарантированного освобождения блокировки, при этом избегая аллокации памяти в куче (no heap allocation), что снижает давление на GC.2

### 5.3. Производительность: Monitor vs. Lock

Бенчмарки показывают, что использование System.Threading.Lock в.NET 9 обеспечивает прирост производительности примерно на 25% по сравнению с классическим Monitor.Enter в сценариях без высокой конкуренции.48

Это достигается за счет:

1. Исключения проверок Object Header.
2. Более эффективной реализации спиннинга.
3. Оптимизации JIT-компилятора, который распознает паттерн lock(lockObject) (где lockObject имеет тип System.Threading.Lock) и генерирует оптимизированный код вызова EnterScope вместо вызова Monitor.Enter.2

## 6. SpinLock и ожидание в пользовательском режиме

В ситуациях, когда блокировка удерживается крайне малое время, даже накладные расходы оптимизированного Monitor могут быть избыточными. Здесь на сцену выходит SpinLock.1

### 6.1. Экономика переключения контекста

Стоимость переключения контекста потока (Context Switch) в Windows составляет от 2,000 до 10,000 процессорных тактов.3 Это "чистые потери", когда процессор занят сохранением регистров, переключением таблиц страниц памяти (TLB flush) и загрузкой состояния нового потока, не выполняя полезной работы.

Если ожидание ресурса займет всего 500 тактов, то уход в сон (блокировка ядра) — это экономически невыгодное решение. Дешевле "покрутиться" в цикле (spin) 500 тактов, потратив энергию CPU, но получив ресурс быстрее.50

### 6.2. Внутреннее устройство SpinLock

System.Threading.SpinLock — это структура (value type), что исключает аллокацию в куче. Она использует активное ожидание (busy waiting).

* Внутри SpinLock использует примитив System.Threading.SpinWait. SpinWait реализует сложную логику "вежливого" спиннинга:
  1. Сначала выполняется несколько десятков итераций простого цикла PAUSE (инструкция процессора, подсказывающая, что это спин-луп, чтобы не перегревать конвейер).
  2. Если блокировка не получена, вызывается Thread.Yield() (передача кванта времени другому готовому потоку на том же ядре).
  3. При дальнейшем ожидании вызывается Thread.Sleep(0) и Thread.Sleep(1), форсируя переключение контекста, чтобы избежать 100% загрузки CPU (starvation).51

### 6.3. Сценарии использования и риски

SpinLock следует использовать только для листовых блокировок (leaf-locks) — блокировок, которые защищают очень короткие участки кода и никогда не захватывают другие блокировки внутри себя.45

Главный риск — Priority Inversion (Инверсия приоритетов): Если поток с высоким приоритетом начнет спиннинг в ожидании ресурса, который удерживается потоком с низким приоритетом, и ОС прервет низкоприоритетный поток, высокоприоритетный поток может бесконечно крутиться, сжигая 100% CPU, не давая низкоприоритетному потоку завершить работу и освободить ресурс. Поэтому SpinLock никогда не следует использовать, если критическая секция может быть длинной или содержать вызовы ОС (I/O).50

## 7. Запрет на блокировки в Async: Thread Affinity

Одной из центральных тем третьего дня является несовместимость классических блокировок с асинхронностью async/await. Понимание этого ограничения требует знания концепции **Thread Affinity** (Привязка к потоку).1

### 7.1. Механика Thread Affinity в Monitor

Классический Monitor (и lock) жестко привязан к потоку. Когда поток вызывает Monitor.Enter, CLR записывает ID этого конкретного *управляемого потока* в SyncBlock (или Object Header). Освободить блокировку (Monitor.Exit) может **только** тот же самый поток. Если Exit вызовет другой поток, будет выброшено исключение SynchronizationLockException.38

### 7.2. Разрыв контекста в Async State Machine

Ключевые слова async/await преобразуют метод в конечный автомат (State Machine).

1. Код до первого await выполняется в вызывающем потоке (например, Thread A).
2. При встрече await, если задача не завершена, текущий поток (Thread A) освобождается и возвращается в пул потоков (ThreadPool).
3. Когда асинхронная операция завершается, продолжение (Continuation) кода после await планируется в пуле потоков.
4. Продолжение может быть подхвачено **любым** свободным потоком (например, Thread B).54

Если мы попытаемся использовать lock внутри async метода:

C#

lock (obj) { // Захват блокировки в Thread A  
 await SomeWorkAsync(); // Thread A уходит, continuation планируется  
 //... продолжение выполняется в Thread B  
} // Thread B пытается вызвать Monitor.Exit для блокировки, которой владеет Thread A

Это приведет к краху логики синхронизации. Либо Thread B получит исключение, либо блокировка навсегда останется "висящей", так как Thread A ее никогда не освободит. Именно поэтому компилятор C# выдает ошибку CS1996: "Cannot await in the body of a lock statement".57

### 7.3. Решение: SemaphoreSlim

Для асинхронной синхронизации используется System.Threading.SemaphoreSlim (инициализированный с count=1 для эмуляции мьютекса).

* **Отсутствие Thread Affinity:** Семафор не запоминает ID потока. Это просто счетчик. Любой поток может уменьшить счетчик (Wait) и любой поток может увеличить его (Release).
* **Асинхронное ожидание:** Метод WaitAsync() не блокирует поток. Если семафор занят, он возвращает незавершенную Task. Внутренне SemaphoreSlim регистрирует TaskCompletionSource в связном списке ожидающих. Когда семафор освобождается (Release()), один из ожидающих TaskCompletionSource переводится в завершенное состояние, и соответствующий await продолжает выполнение.59
* **Риск реентерабельности:** Поскольку у SemaphoreSlim нет привязки к потоку, он не поддерживает рекурсивный вход (re-entrancy). Если один и тот же асинхронный поток попытается дважды сделать await semaphore.WaitAsync(), он заблокирует сам себя (Deadlock), так как семафор не знает, что это тот же самый логический поток исполнения.58

## 8. Практическая реализация и диагностика

В рамках практической части дня рассматривается симуляция проблем синхронизации.

### 8.1. Симуляция банковского счета (Race Condition)

Реализация счета с ошибкой гонки демонстрирует нарушение инвариантов при отсутствии атомарности:

C#

// Небезопасный код  
public void Deposit(decimal amount) {  
 \_balance += amount; // RMW-операция: Read, Add, Write  
}

При одновременном вызове Deposit из 10 потоков, итоговый баланс будет меньше суммы всех депозитов из-за перезаписи результатов.

Исправление:

* Для decimal нельзя использовать Interlocked, так как он 128-битный.
* Необходимо использовать lock (или System.Threading.Lock в.NET 9) для создания критической секции, которая обеспечивает как атомарность всей транзакции, так и барьеры памяти для видимости изменений.61

### 8.2. Симуляция и анализ Deadlock

Взаимная блокировка (Deadlock) возникает при нарушении порядка захвата ресурсов.

Сценарий:

* Поток 1: lock(A) -> sleep -> lock(B)
* Поток 2: lock(B) -> sleep -> lock(A)

Оба потока зависнут навсегда.

Диагностика в Visual Studio:

Инструмент Parallel Stacks (Параллельные стеки) в режиме отладки визуализирует граф зависимостей потоков. Окно Threads покажет, что потоки находятся в состоянии WaitSleepJoin с блокировкой на методе Monitor.Enter (или Wait).

В отличие от SQL Server, CLR по умолчанию не имеет встроенного детектора дедлоков, который бы автоматически убивал "жертву". Приложение просто зависнет.62

### Таблица 1: Сравнительный анализ примитивов синхронизации

| **Примитив** | **Механизм реализации** | **Thread Affinity** | **Поддержка Async** | **Рекурсия** | **Накладные расходы (без конкуренции)** |
| --- | --- | --- | --- | --- | --- |
| **Interlocked** | Аппаратный LOCK/CAS | Нет | Нет | Нет | < 10 нс (экстремально быстро) |
| **Monitor** (lock) | Гибридный (Spin + Kernel Wait) | **Да** | Нет | **Да** | ~20-50 нс (Thin Lock) |
| **SpinLock** | Активное ожидание (CPU loop) | Опционально | Нет | Нет | ~10-20 нс (риск загрузки CPU) |
| **SemaphoreSlim** | Счетчик + TaskCompletionSource | Нет | **Да** | Нет | Выше (аллокации Task/Node) |
| **Lock** (.NET 9) | Гибридный (без Object Header) | **Да** | Нет | **Да** | ~15 нс (на 25% быстрее Monitor) |

### Таблица 2: Влияние модели памяти на видимость данных

| **Операция** | **x86/x64 (Strong Model)** | **ARM64 (Weak Model) до.NET 9** | **ARM64 (Weak Model) в.NET 9+** |
| --- | --- | --- | --- |
| **Обычная запись** | Видима почти сразу (Store Buffer) | Может быть задержана/переупорядочена | Может быть задержана/переупорядочена |
| **Volatile запись** | MOV (неявно Release) | DMB (Full Barrier - дорого) | STLR (Store-Release - эффективно) |
| **Volatile чтение** | MOV (неявно Acquire) | DMB (Full Barrier - дорого) | LDAR (Load-Acquire - эффективно) |
| **Стоимость барьера** | Низкая (Hardware TSO) | Высокая (остановка конвейера) | Средняя (One-way barrier) |

**Список источников (Интегрирован в текст):**

* 1 План обучения C#
* 3 Атомарность, RMW, x86 LOCK prefix
* 40 Внутреннее устройство Monitor, Object Header, Thin/Fat Locks
* 2 System.Threading.Lock в.NET 9
* 58 SemaphoreSlim, Async Wait
* 16 Модели памяти ARM64, LDAR/STLR, DMB
* 5 Стоимость переключения контекста и SpinLock

#### Источники

1. План обучения C# третьей недели
2. Lock Class (System.Threading) | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.threading.lock?view=net-10.0>
3. Peeking into assembly code to understand why count++ is not atomic - YouTube, дата последнего обращения: декабря 3, 2025, <https://www.youtube.com/watch?v=kBHd7kn_1EU>
4. Interlocked Class (System.Threading) - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.threading.interlocked?view=net-10.0>
5. What provides spinlocks effectiveness? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/22398909/what-provides-spinlocks-effectiveness>
6. Atomic Operation In C#. Introduction | by Wayne Ye - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@wayneye/atomic-operation-in-c-a40590a4d2a8>
7. Is accessing a variable in C# an atomic operation? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/9666/is-accessing-a-variable-in-c-sharp-an-atomic-operation>
8. What operations are atomic in C#? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/11745440/what-operations-are-atomic-in-c>
9. Atomic vs. Non-Atomic Operations - Preshing on Programming, дата последнего обращения: декабря 3, 2025, <https://preshing.com/20130618/atomic-vs-non-atomic-operations/>
10. Are basic arithmetic operations in C# atomic - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/10530337/are-basic-arithmetic-operations-in-c-sharp-atomic>
11. Some performance implications of CAS operations - Joe Duffy - Blog, дата последнего обращения: декабря 3, 2025, <https://joeduffyblog.com/2009/01/08/some-performance-implications-of-cas-operations/>
12. Compare-and-swap - Wikipedia, дата последнего обращения: декабря 3, 2025, <https://en.wikipedia.org/wiki/Compare-and-swap>
13. Is x86 CMPXCHG atomic, if so why does it need LOCK? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/27837731/is-x86-cmpxchg-atomic-if-so-why-does-it-need-lock>
14. Why is CompareAndSwap instruction considered expensive? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/2972389/why-is-compareandswap-instruction-considered-expensive>
15. You Can Do Any Kind of Atomic Read-Modify-Write Operation - Preshing on Programming, дата последнего обращения: декабря 3, 2025, <https://preshing.com/20150402/you-can-do-any-kind-of-atomic-read-modify-write-operation/>
16. Performance Improvements in .NET 9 - Microsoft Developer Blogs, дата последнего обращения: декабря 3, 2025, <https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-9/>
17. The Future of .NET: Inside the Remarkable Advances of .NET 9 - Arna Softech, дата последнего обращения: декабря 3, 2025, <https://arnasoftech.com/future-of-dotnet9/>
18. Understanding the Interlocked API in C# | by Roman Fairushyn - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@fairushyn/understanding-the-interlocked-api-in-c-49c8fd502ad6>
19. CMPXCHG — Compare and Exchange, дата последнего обращения: декабря 3, 2025, <https://www.felixcloutier.com/x86/cmpxchg>
20. Can CompareExchange be implemented with CompareAndSwap? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/2463969/can-compareexchange-be-implemented-with-compareandswap>
21. C# - The C# Memory Model in Theory and Practice, Part 2 - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/archive/msdn-magazine/2013/january/csharp-the-csharp-memory-model-in-theory-and-practice-part-2>
22. Volatile keyword in C# – memory model explained - igoro.com, дата последнего обращения: декабря 3, 2025, <https://igoro.com/archive/volatile-keyword-in-c-memory-model-explained/>
23. What I do not understand about volatile and Memory-Barrier is - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/21989776/what-i-do-not-understand-about-volatile-and-memory-barrier-is>
24. Memory barriers force cache coherency? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/30958375/memory-barriers-force-cache-coherency>
25. volatile keyword - C# reference - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/volatile>
26. volatile with release/acquire semantics - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/6526432/volatile-with-release-acquire-semantics>
27. Memory Model: preventing store-release and load-acquire reordering - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/16572571/memory-model-preventing-store-release-and-load-acquire-reordering>
28. Reordering of operations around volatile - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/11357439/reordering-of-operations-around-volatile>
29. ARM vs x86; What are the key differences? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/44269853/arm-vs-x86-what-are-the-key-differences>
30. Memory Barriers - Nadeem Afana's Blog, дата последнего обращения: декабря 3, 2025, <https://afana.me/archive/2015/07/10/memory-barriers-in-dot-net.aspx/>
31. CLR - .NET Development for ARM Processors - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/archive/msdn-magazine/2012/august/clr-net-development-for-arm-processors>
32. “Strong” and “weak” hardware memory models - Herb Sutter, дата последнего обращения: декабря 3, 2025, <https://herbsutter.com/2012/08/02/strong-and-weak-hardware-memory-models/>
33. Examining ARM vs X86 Memory Models with Rust - Nick Wilcox's Coding Blog, дата последнего обращения: декабря 3, 2025, <https://www.nickwilcox.com/blog/arm_vs_x86_memory_model/>
34. Arm64 Performance Improvements in .NET 7 - Microsoft Developer Blogs, дата последнего обращения: декабря 3, 2025, <https://devblogs.microsoft.com/dotnet/arm64-performance-improvements-in-dotnet-7/>
35. Significant Performance Disparity Between Arm64 and x64 Write Barriers #106051 - GitHub, дата последнего обращения: декабря 3, 2025, <https://github.com/dotnet/runtime/issues/106051>
36. Load-Acquire and Store-Release instructions - Arm Developer, дата последнего обращения: декабря 3, 2025, <https://developer.arm.com/documentation/102336/0100/Load-Acquire-and-Store-Release-instructions>
37. What's new in .NET 10 runtime - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-10/runtime>
38. System.Threading.Monitor class - .NET - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-threading-monitor>
39. Deep Dive Into Monitor Class as Hybrid Synchronization Construct in .NET, дата последнего обращения: декабря 3, 2025, <https://dev.to/rasulhsn/deep-dive-into-monitor-class-as-hybrid-synchronization-construct-in-net-cn8>
40. Managed object internals, Part 2. Object header layout and the cost of locking - Microsoft Developer Blogs, дата последнего обращения: декабря 3, 2025, <https://devblogs.microsoft.com/premier-developer/managed-object-internals-part-2-object-header-layout-and-the-cost-of-locking/>
41. CLR monitors and sync blocks - Joe Duffy - Blog, дата последнего обращения: декабря 3, 2025, <https://joeduffyblog.com/2007/06/24/clr-monitors-and-sync-blocks/>
42. (PDF) Fine-grained Adaptive Biased Locking - ResearchGate, дата последнего обращения: декабря 3, 2025, <https://www.researchgate.net/publication/252065957_Fine-grained_Adaptive_Biased_Locking>
43. .NET Core Concepts (Synchronization) | Marian Todorov - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@meriffa/net-core-concepts-synchronization-0dfffa50c5dc>
44. Object header get complicated | My Coding Place, дата последнего обращения: декабря 3, 2025, <https://mycodingplace.wordpress.com/2018/01/10/object-header-get-complicated/>
45. SpinLock in C#. In which type of algorithm SpinLock is a better choice against Monitor? [duplicate] - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/14642215/spinlock-in-c-in-which-type-of-algorithm-spinlock-is-a-better-choice-against-m>
46. `System.Threading.Lock` in .NET 9 — A Modern, Safer Locking Mechanism, дата последнего обращения: декабря 3, 2025, <https://dev.to/cristiansifuentes/systemthreadinglock-in-net-9-a-modern-safer-locking-mechanism-7n2>
47. NET: why store Sync Block in every object? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/40346893/net-why-store-sync-block-in-every-object>
48. Locking with .NET 9.0's System.Threading.Lock, even on older frameworks - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/csharp/comments/1f6ari2/locking_with_net_90s_systemthreadinglock_even_on/>
49. For which specific use-case is the new `System.Threading.Lock` in .Net 9 introduced?, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/79188426/for-which-specific-use-case-is-the-new-system-threading-lock-in-net-9-introdu>
50. How is spinlock different from polling? - Software Engineering Stack Exchange, дата последнего обращения: декабря 3, 2025, <https://softwareengineering.stackexchange.com/questions/301947/how-is-spinlock-different-from-polling>
51. SpinWait - .NET - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/threading/spinwait>
52. Spinlock implementation reasoning - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/49232620/spinlock-implementation-reasoning>
53. Spinlock Class in Threading C#, дата последнего обращения: декабря 3, 2025, <https://www.c-sharpcorner.com/UploadFile/1d42da/spinlock-class-in-threading-C-Sharp/>
54. Understanding Async, Avoiding Deadlocks in C# | by Eke Péter | Rubrikk Group - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/rubrikkgroup/understanding-async-avoiding-deadlocks-e41f8f2c6f5d>
55. Locking behaviour when dealing with async/await [closed] - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/77469336/locking-behaviour-when-dealing-with-async-await>
56. For async in C#, how exactly are tasks passed onto other threads? : r/csharp - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/csharp/comments/1l3o0ky/for_async_in_c_how_exactly_are_tasks_passed_onto/>
57. The lock statement - synchronize access to shared resources - C# reference, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/lock>
58. Locking & async/await - Rock Solid Knowledge, дата последнего обращения: декабря 3, 2025, <https://www.rocksolidknowledge.com/articles/locking-asyncawait>
59. Fun with the Spiral of Death - Marc Gravell, дата последнего обращения: декабря 3, 2025, <https://blog.marcgravell.com/2019/02/fun-with-spiral-of-death.html>
60. SemaphoreSlim.WaitAsync Method (System.Threading) | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.threading.semaphoreslim.waitasync?view=net-10.0>
61. what is the reasoning behind volatile semantics in Java and C# - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/11353274/what-is-the-reasoning-behind-volatile-semantics-in-java-and-c-sharp>
62. Deadlocks Guide - SQL Server - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-deadlocks-guide?view=sql-server-ver17>