Автор: Малышева Александра, Ольга Черникова
Лектор: Калишенко Евгений
Курс по параллельному программированию
СпбАУ, 2017 год, 3 курс, 6 семестр, 302 группа

Замечание:
БОльшая часть лекции будет на доске. Сегодня мы рассмотрим щепетильную тему
Потому что на ней можно убедить в неправильном решении, ошибившись в одной строке.

Барьеры памяти

Рассмотрим простой пример: есть две функции запускаемые в двух потоках

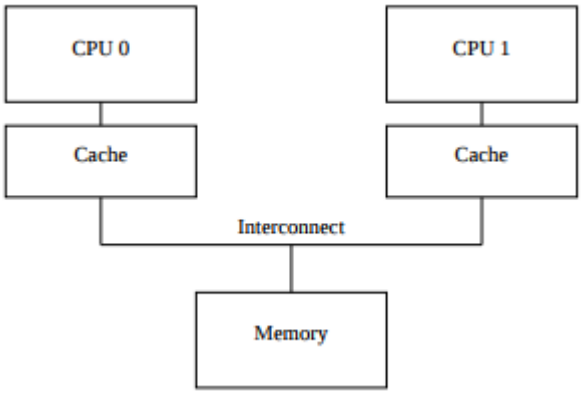
int data
bool ready
f() {
data = 42
ready = true
}
- готовим данные и говорим, что данные готовы
	g(){
	if (ready)
	assert(data = 42);
	}
	- если данные готовы, то проверяем данные

Есть вероятность, что код упадет:
Кода обновление данных произошло в разных кешах для разных функций.
Способ борьбы с этим: нужно поставить валатайл

Замечание:
1. Современные компиляторы на наших компьютерах не упадут на этих функциях.
2. Также есть reordering на уровне компилятора: javac, gcc мы по поводу него сегодня не говорим

Проблема:
Такой код упасть может (не на ноутбуке, но на телефоне), то есть там, где есть (или нет?) reordering. Будем пытаться решить эту проблему.

Как устроены современные процессоры:



Пусть у нас есть CPU0 CPU1, у них есть КЭШ (НУО будем работать только с первым уровнем КЭШ)

Замечание:
Любые ассемблерные процедуры работают с КЭШ у процесса нет возможности работать с памятью минуя КЭШ

Первый способ решение - **volatile**

Автор: Малышева Александра, Ольга Черникова
Лектор: Калишенко Евгений
Курс по параллельному программированию
СпбАУ, 2017 год, 3 курс, 6 семестр, 302 группа

Итак, мы хотели решить проблему с помощью volatile, что делает volatile:
В C++ он гарантирует, что ваша переменная не кэшируется в регистрах
и при любом обращение к переменной - он будет спрашиваться и брать из КЭШ

Рассмотрим как это работает:

1. Пусть у нас есть ячейка в памяти
2. Ядро говорит хочу ячейку
3. Ячейка переходит в КЭШ, а потом используется в CPU
4. Также делает второе ядро

Проблема: возможно рассинхронизация КЭШ. Таким образом volatile от этого вообще не спасает.

Хотим:
Систему, такую чтобы ядро понимало, что ячейка памяти в КЭШ менялась, и меняло бы соответствующую ячейку в памяти.

Второй способ решение - **протокол поддержки когерентности кэша процессоров**

Замечание:

1. Обычно рассматривают на учебном протоколе, но его сейчас, конечно, не используют: MESI
2. Операционная система планирует в терминах потока, то есть для процесса совершенно не важно: поток какого процесса сейчас выполняется.

Для нас важен случай когда выполняется потоки одного процессора на разных ядрах
Рассмотрим пример:

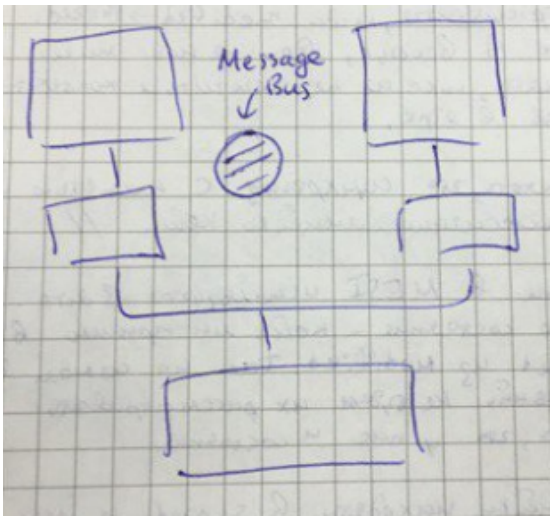
1. Поток на ядре 0 говорит, что хочет переменную data
2. data копируется в КЭШ 0
   (мы копируем в линейку КЭШ = 64 байта и в ней может быть несколько переменных)
3. Тогда ядро 0 уже работает с переменной с помощью обращения с КЭШ 0
4. Если второе ядро также копирует и модифицирует переменную дата - возникнет проблема

Как же разрешить конфликт:
MESI - к линейки добавляются +2 бита
кодируя состояние линейки:

- **Invalidate** - линейки, в которых нет данных. Изначально линейка КЭШ процессора находиться в состоянии **I**, если в ней никогда ничего не было записано.
- **Exclusive** - данное ядро является монопольным владельцем данной линейки памяти.
- **Modify** - данные были изменены именно этим ядром.
  E->M - то процессор не изменяет данные в памяти, изменения остаются в КЭШ
- **Shared** - если линейка КЭШ разделяется больше, чем одним ядром, и не одно ядро его ее не изменило, то оно переходит в состояние Shared: данные

находятся больше чем у одно ядра в КЭШ
и для того, чтобы изменить данные - нужно всех оповестить

Теперь рассмотрим как все это работает:



Пусть **MsgBuf** - брокер обмена сообщениями между ядрами.
Вы кидаете сообщение брокеру, и он передает всем, кому считает нужным

Пример: сообщение read

1. CPU0 - считала data
2. Теперь CPU1 - тоже захотело считать data и шлет сообщение read брокеру
3. Брокер: если ни одно ядро нет копии data - то он просто считывает data, но так как CPU0 уже считывало - то но и ответит за эти данные: возьмет и передаст свое общение.
4. И в итоге и CPU0 CPU1 знают, что они оба владеют этими данными
5. Таким образом оба ядра понимают, что эта ячейка shared

N.B.

1. Если есть два ядра с состоянием Shared, то мы ждем ответа от любого из них
2. Брокер реализован на аппаратном уровне!)
   И мы скатываемся не на общую память, а на обмен сообщениями. Но программном на уровне абстракции - легче считать, что мы работаем с общей памятью.
3. Долгое время модели памяти процессоров были закрыты, потом поняли, что бывают ошибки, и надо открыть

Когда все это сбрасывается в память:

1. Если мы были в состоянии modify ->Shared - нужно сбросить данные об изменениях ячейки на ядро.
2. Можно сбрасывать в момент переключения контекста на одном из ядре:
   (Это меняется для того, чтобы если поток засыпает, в тот момент, когда он просыпался - он был уверен в состоянии кэша)
3. У нас нет инструкции сбросить весь КЭШ, просто чисто на аппаратном уровне.

Автор: Малышева Александра, Ольга Черникова
Лектор: Калишенко Евгений
Курс по параллельному программированию
СпбАУ, 2017 год, 3 курс, 6 семестр, 302 группа

Проблема:
Если состояние линейки было Modified, то в какое состояние нужно перевести, когда у нас просят read, так чтобы запомнить, что у нам надо на память сбросить??
(MESI эту проблему не решает.)

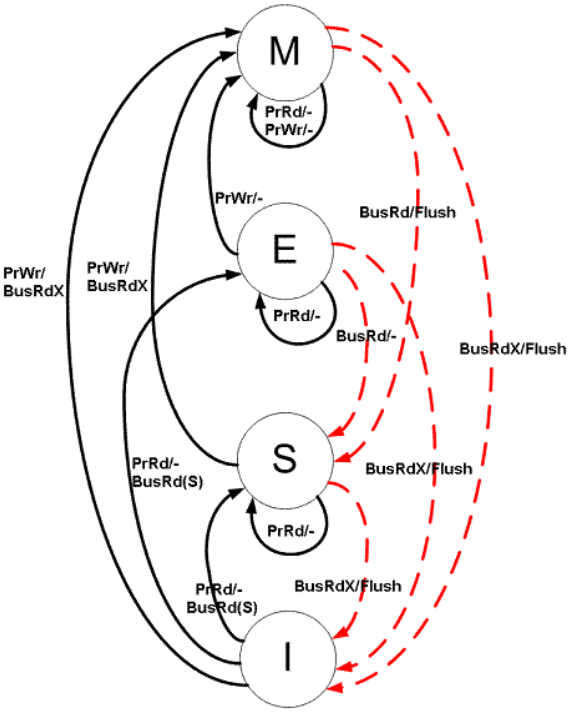
Еще раз о том, какое состояние куда переходит:

Processor Requests to Cache includes the following operations:

1. PrRd: The processor requests to **read** a Cache block.
2. PrWr: The processor requests to **write** a Cache block

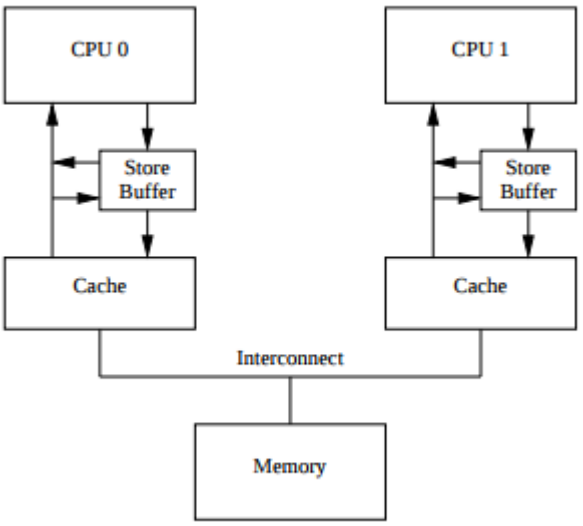
Bus side requests are the following:

1. BusRd: Snooped request that indicates there is a **read** request to a Cache block made by another processor
2. BusRdX: Snooped request that indicates there is a read request to a Cache block made by another processor which **doesn't already have the block.**
3. BusUpgr: Snooped request that indicates that there is a write request to a Cache block made by another processor but that processor already has that **Cache block resident in its Cache**.
4. Flush: Snooped request that indicates that an entire cache block is written back to the main memory by another processor.
5. FlushOpt: Snooped request that indicates that an entire cache block is posted on the bus in order to supply it to another processor(Cache to Cache transfers).



Ускорение с помощью буфера обмена

Важный момент: когда ядро что-то меняет, оно должно дождаться ответа от shared: да, я поменял, ок



<u>Мотивация к изменению:</u>
Но это было бы не производительно, поэтому производители процессоров так не делают!)

<u>Решение</u>: Вводим дополнительный буфер обмена - **store buffer**

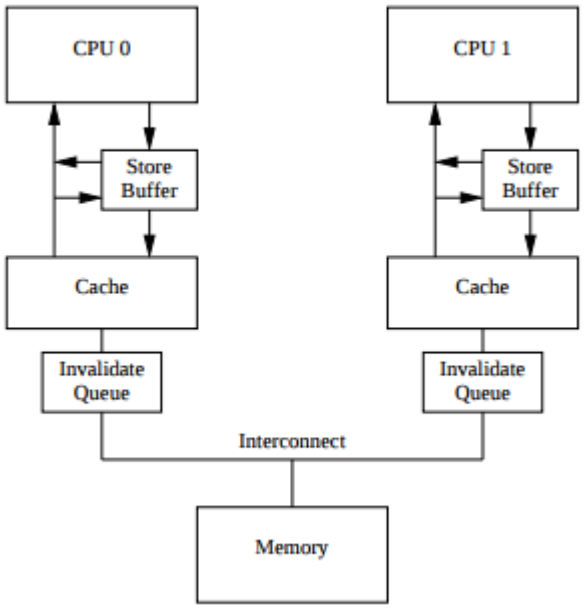
N.B.:
Если два ядра захотели поменять одновременно: то они оба шлют сообщения брокеру, и брокер шлет

ответ “у тебя получилось” только одному из ядер.

Реализация:

Мы, оптимистично предполагаем, что нам ответят, что все хорошо, туда складываем сообщения модификации линек процессора, которые являются shared “я меняю линеечку кэша”, и пока все остальные не ответили, что все поменяли - линейка не уйдет из **store buffer.**

Ускорение с помощью Invalidate queue



Важный момент:Когда мы меняем Shared данные - мы кладем в **store buffer** сообщение об изменениях и посылаем второму ядру сообщение об инвалидации данных. (На самом деле процессор должен был бы остановить все свои операции и инвалидировать данные, и лишь потом продолжить.)

Мотивация к изменению: Это медленно!

Решение:
Поэтомы мы вводим очередную очередь: **Invalidate queue** - в ней складываются сообщения об инвалидациях

Реализация

Когда же посылать запрос о том, что мы поменяли все:
Честно было бы так: когда положили в очередь, подождали, и выполнили - тогда и ответить.
Реально это не так: когда кинули в очередь - тогда сразу и ответилиы

N.B.: Конечно тут уже сто способов прийти к когерентному состоянию

Заключение

Почему это называется протоколом когерентности кэш и почему все это работает.

- То есть как с этим жить?

Вы можете сказать процессору:
Я хочу, чтобы все изменения были правда со всеми согласованы.

1. То есть говорим:
   Пока не очистишь **store buffer**- не работай.
   Для x86 архитектуры: **smp wrb()**;
   N.B.: ассемблерные команды могут влиять только на ваше ядро
2. И еще можем сказать
   Пожалуйста, примени все изменения. которые накапали тебе в очередь,
   То есть пока не очистишь весь **Invalidate queue** - не работает
   Для x86 архитектуры:**smb rub()**;

- И СОБСТВЕННО ЭТО И ЕСТЬ БАРЬЕРЫ ПАМЯТИ

Пример ошибки видимости

Теперь по шагам разберем пример, при котором возникает ошибка видимости, чтобы понять что это такое.

f() {
  a = 1;
  b = 1;
}
	bar() {
	  while (b == 0) continue;
	  assert (a == 1);
	}

На сайте, где выкладываются материалы есть ссылка на статью, где подробно рассматривается пример и другие хорошие примеры.

Начальные условия: функции выполняются в разных потоках. a и b в кеше первого ядра и a в кеше второго. Нормальное допущение. Например их кто-то заранее считал.

#	Операция в первом потоке	Операция во втором	a в 1	b в 1	a в 2	b в 2
0			s	e	s	x
1	a = 1; Будет использоваться store buffer И a идет в store buffer. Должно послать запрос другим ядрам на инвалидацию read invalidate инвалидировать и еще прочитать откуда-нибудь. В одной линейке хеша, может быть не только		s	e	s	x

	переменная a.					
2		Второе ядро начинает выполнять цикл while. while(). Отправляет запрос на чтение b, потому что b не находится в кеше процессора.	s	e	s	x
3	b = 1; Первый поток все еще думает, что он монополист, поэтому все спокойно меняет.		s	m	s	x
4	пришел запрос read от запроса второго. Переходит в состояние shared для b. Отправляет сообщение, что вот оно shared.		s	s	s	x
5		b попадает в кеш.	s	s	s	s
6		Выходим из цикла.	s	s	s	s
7		assert(a == 1) падает!!!	s	s	s	s
8		Принимает и отвечает на запрос read invalidate. Но поздно.	s	s	s	s
9	Сбрасываем store buffer. Потому что получили ответ на invalidate.		s	s	s	s

Представлена последовательность действий, при которой read invalidate опоздал и мы считали не то значение переменной, причем мы про нее знали, что она shared.

Оценка на проявление этой ошибки примерно такая, если запустить стрессится этот код, то он будет падать 1 раз примерно на 10 000.

Если в Java сделать volatile то появятся read и write барьеры и такой проблемы уже не будет. И то после Java 1.5. Производители процессоров, когда вводили store buffer,

invalidate queue не сразу воспроизводили подобные ситуации и не сразу предоставили API доступ к барьерам памяти в принципе. Сейчас есть две такие абстракции, одна называется барьеры памяти, другая - модели памяти. Это немного разные вещи, одна из другой вытекает. Сейчас мы шли снизу и сейчас мы поднялись на более высокий уровень, чтобы понять, как это все отражается на реальном коде. Для того чтобы договорится между разными архитектурами и чтобы разные архитектуры единым образом предоставляли свои гарантии на модель памяти как-то во вне. Придумали абстракцию барьеры памяти в некотором теоретическом смысле.

Барьеры памяти

У нас возможны две операции - store и load. То есть операции записи и операции чтения. И есть 4 вида барьеров.

LoadLoad	LoadStore
StoreLoad	StoreStore

Барьер типа XY, где X/Y это либо load либо store, это барьер который гарантирует выполнение всех X операций до все Y операций.

Барьер памяти говорит о том, что все операции чтения до этого барьера для текущего ядра будут гарантированно выполнены до всех операций чтения после барьера.

Пример.
b == 2
LoadLoad барьер
c == 3

Тогда гарантируется, что чтение b будет раньше чтения c. То есть чтение b завершиться раньше начала чтения c.

LoadStore - все операции чтения до барьера будут гарантированно выполнены до операций записи после этого барьера.

Проще воспринимать, что барьер не пропускает левый операции вниз, и не пропускает правые операции наверх.

Редкая архитектура скажет, что у нее есть ассемблерная инструкция, которая соответствует каждому их 4 видов барьеров. Обычно барьеры объединяются в группы, одна из наиболее распространенных групп барьеров. Это (LoadStore и StoreStrore) и (LoadLoad и LoadStore). (LoadLoad и LoadStore) - называется **acquire**. (LoadStore и StoreStrore) - называется **release.**

Почему так придумали их объединить. (LoadLoad и LoadStore) это означает, что любые операции будут выполнены гарантировано после операций чтения, если есть этот барьер. Если сделать release, то операции запираются с другой стороны (LoadStore и StoreStrore), любые операции будут выполнены до операций записи снизу. Таким образом, если будут применяться в коде два барьера acquire и release, то в некотором роде будет захватываться и освобождаться как бы примитив синхронизации. Все строчки кода, которые будут находится между этими барьерами - никуда не расползутся, не вверх ни вниз. При этом допускается что какая-нибудь операция допустим записи сверху переедет внутрь ограничения. Но это нас не сильно волнует, главное что из блока никакие операции не выйдут.

Наборы этих барьеров или один из этих барьеров реализуется на железе по-разному. Бывает железо, которое что-то гарантирует из коробки. Например поддерживает отдельную ассемблерную инструкцию для конкретного барьера. Пусть не поддерживает ни для какого отдельно, но зато поддерживает для двойки барьеров описанных выше. Главное чтобы железо сказало, что оно это делает и делает в терминах этих барьеров.

If (ready) assert(data == 42);	data = 42; ready = true;

Был следующий пример, когда все падало. Если добавить некоторые барьеры, то все будет хорошо.

If (ready)
LoadLoad
  assert(data == 42);
	data = 42;
	StoreStore
	ready = true;

Догадаться что нужен барьер LoadLoad достаточно просто, здесь кроме чтения нет никаких других операций. Барьер что первая операция чтения будет выполнена до второй операции чтения. Нужно мыслить в терминах видимости переменных. Процессор увидит изменения в переменных, которые читались раньше до того, как он увидит изменения в переменных после барьера. Во втором случае нужно поставить StoreStore. Наше ядро увидит изменения ready после того, как увидит изменение data.

Фактически на самой известной нам архитектуре барьер LoadLoad превращается в операцию сброса invalidate queue.

Пока мы обошлись барьерами LoadLoad StoreStore, сейчас будет пример, когда недостаточно этих барьеров. Есть два потока, которые разделяют две булевские переменные.

bool x = true; bool y = true;

Автор: Малышева Александра, Ольга Черникова
Лектор: Калишенко Евгений
Курс по параллельному программированию
СпбАУ, 2017 год, 3 курс, 6 семестр, 302 группа

x = false;
StoreLoad
assert(y);
	y = false;
	StoreLoad
	assert(x);

Штатным поведением считаем тогда, когда код упадет. Какая-та из функций всегда должна упасть, если сначала срабатывает первая функция, то вторая точно упадет и наоборот.

При неиспользование барьеров памяти этот код может не упасть. Здесь нужно поставить барьеры StoreLoad, поскольку до только store, а после только load.

Для того, чтобы производители разных процессоров и разных архитектур говорили нам, что они гарантируют, в что они не гарантируют, что у них есть и как оно у них есть. Существует 4 разных модели памяти, которые говорят об отсутствии или наличие тех или иных барьеров памяти по умолчанию с точки зрения архитектуры.

1. Sequential consistency; - самая жесткая модель памяти. Эта модель гарантирует применение всех возможных барьеров после каждой операции чтения и записи. Это означает, что барьеры фактически не нужны. При этом для того чтобы поддержать эту модель памяти и вообще работать в терминах модели памяти в современных архитектурах вы можете использовать почти все перечисленные модели памяти. x86 не sequential consistency.
2. Strong-ordered; Эта модель память, которая из коробки гарантирует acquire/release семантику. То есть все операции чтения записи по умолчанию являются acquire/release. Это еще называется TSO total strong order. К таким архитектурам как раз относится известная вам x86. То есть здесь уже можно забить на почти все барьеры, кроме storeLoad, то есть какие-то перестановки такая семантика все-таки разрешает и ошибка как показано выше может быть. И более того она такие перестановки достаточно часто использует.
3. Weak-order. В этом случае возможны все перестановки. Сюда относится архитектура ARMv7. Тут есть единственная гарантия, что если есть код вида

   int* ptr = ….
   if (ptr ...)

   То зависимые по данным инструкции переставляться не будут. Все остальное с точки зрения reordering здесь разрешено. Отчасти поэтому он является достаточно производительным процессором. Это было осознанное решение использовать

	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache/Pipeline?
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64				Y				
ARMv7-A/R	Y	Y	Y	Y	Y	Y	y	Y
IA64	Y	Y	Y	Y	Y	Y		Y
(PA-RISC)	Y	Y	Y	Y				
PA-RISC CPUs								
POWER™	Y	Y	Y	Y	Y	Y		Y
(SPARC RMO)	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86				Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries®				Y				Y

максимально спекулятивное поведение процессора, чтобы выжимать из него максимум производительности.

4. Super weak, где разрешены перестановки любого вида. Оно уже не работает… В первой строчке в таблице.

Вот табличка для разного вида архитектур. Нам будет интересен IA64. Здесь сверху прописаны все возможные перестановки которые используется данным процессором. IA64 использует практически все, кроме Dependent loads. X86 использует stores reordering. ARM использует практически все, там где маленькая буква у, можно считать что ее нет. Здесь указано, что каждый конкретный процессор делает. Не так важно понимать, что делает каждый конкретный процессор. По умолчанию, когда вы пишите код вы должны не полагаться ни на что. Вы должны понимать, что ваш код может быть исполнен на каком-нибудь ARM, на котором ничего не заработает. Для этого у нас есть при программировании на достаточно высокоуровневых языках, простые способы добиваться всего этого дела.

Высокоуровневая работа с барьерами памяти

Например, если говорить о плюсах, то именно такие модели именно в таких названиях моделей памяти можно увидеть в перечисление, когда вы работаете с atomic, atomic только в 11 плюсах появились.

По умолчанию у вас используется memory load sequential consistency. Есть memory-order acquire, есть memory order release. То есть acquire/release семантика будет поддерживаться на уровне операции чтения или операции записи, если вы понимаете что делаете.

Есть вспомнить пример, где мы явно ставили барьер loadLoad и loadStore, если переписать его из предположения что мы пишем на плюсах и на atomic, то правильно с точки зрения производительности будет его написать примерно следующим образом. Ready теперь атомик, вызываем метод store и далее нужно передать какую-то модель памяти. С load точно так же. Фактически в случае чтения мы используем acquire семантику, в случае с записью release.

if (ready.load(acquire)) {
  assert(data == 42)
}
	data = 42;
	ready.store(true, release)

Мораль, если код будет работать на x86 не факт что он будет работать при другой архитектуре процессора.
Всегда есть возможность передать memory order relax, то есть расслабиться и ничего не применять. Это достаточно хорошо с точки зрения производительности и если мы

считаем что нам в данный момент все равно на видимость этой переменной с точки зрения других процессоров, то имеет смысл это использовать.

На Java слово volatile гарантирует корректное использование моделей памяти при работе на любой архитектуре. Volatile на java как раз таки спасает от всех этих барьеров. Он обеспечивает acquire/release семантику когда это нужно бесплатно. И на большинстве машин она и так есть и обеспечивает не задумываться на счет store load барьера. То есть на джаве было достаточно объявить переменные volatile. На плюсах только atomic, до 11 стандарта только руками.

Вы разрабатываете высокопроизводительное приложение и так получается, что на линуксе на серверной машине приложение призвано обрабатывать видео в реальном времени и осуществлять слежение за некоторыми объектами с достаточно высокой точностью. Бывает так что через сутки работа или через двое суток ваше видео просто перестает идти. Система зависла. Причем не все система, операционка, она работает. Как вы будете искать где ошибка?

Обычно, если что-то зависло логично подключиться к процессу и посмотреть что в нем там есть. GDB там подключится. Как только вы пытаетесь подключиться к процессу у вас все сново начинает работать.

После того как вы подключились к strace вот то что выдал strace…

Там мы видим странную ошибку eagain было два часа потрачено, чтобы понять связана ошибка с этим или нет. Нет не связана.

Далее была программа проанализирована, убраны все примитивы синхронизации, wait И notify. Оставлены только некоторые надежные фреймворки.

Единственное что осталось это queue, цикл обработки событий.

В целом, в случае memory reorder если поток подумал, что данные еще не готовы, то теоретически возможно бесконечное засыпания потока, а если к нему подключится дебагером происходит явный вызов барьеров памяти, потому что происходит явная смена контекста процессора.

Все барьеры памяти срабатывают без нашего ведома в трех случаях

- При захвате или освобождение примитивов синхронизации.
- При переключение контекста потока.
- Когда происходит всякие сигналы.

Оказался баг в ядре линукса и могло зависнуть любое приложение, которое имело более одного потока. В том числе джавовский поток. Причем проявлялось это только на очень высокопроизводительной системы.

Исправление этой ошибки заняло всего пару строк. Там в futex дефолтного барьера никогда не было и когда увеличили производительность в одном месте забыли поставить комплиментарный барьер. Причем забыли только в одном потоке.

С барьерами памяти сейчас работать очень не просто.