Java Memory Model

Александр Божнюк 371 группа

Что такое модель памяти и для чего она нужна?

- JMM - Это часть спецификации языка Java.
- Она отвечает на конкретный вопрос: “Какое значение будет получено в результате операции чтения?”
- JMM специфицирует гарантии, которые должна давать JVM, про то, когда записи в переменные могут становиться видимыми другими потоками.
- В однопоточных программах все довольно просто, однако в многопоточной среде появляются проблемы.

Атомарность

- **Атомарность** гарантирует, что любой поток в любой момент времени считает либо значение по умолчанию, либо значение уже записанное ранее. Никакого мусора.
- **JMM: Чтения/записи атомарны для всего, кроме long и double**
- long и double являются 64 битными (все остальное 32 битное), не все машины поддерживают атомарные чтение и запись значений по 64 бит (однако таких меньшинство).
- **JMM: volatile long и volatile double атомарны**
- **Запись и чтение ссылок всегда атомарно**

Word Tearing

Потребность: независимость операций над независимыми элементами (элементы в массиве, поля).

```
T[] as = new T[...]; as[1] = as[2] = V0;
```

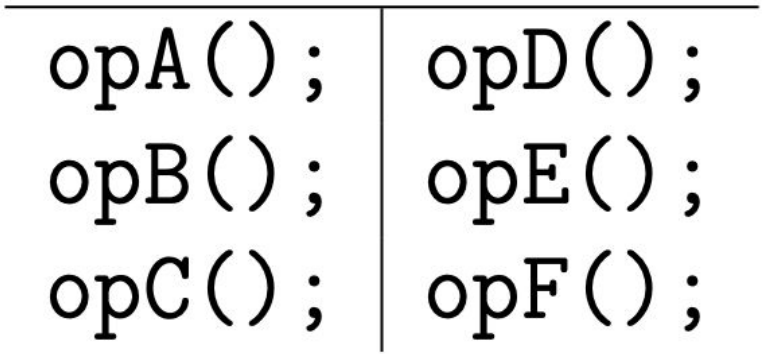
`as[1] = V1;`	`as[2] = V1;`	
`<term>`	`<term>`	`<join both>`
		`T r1 = as[1];`
		`T r2 = as[2];`
		`assert (r1 == r2)`

Word tearing

- **JMM: Word tearing запрещен**
- Как атомарно записать 1-битный boolean, если атомарно можно записать N ($N \geq 8$) бит?
- Если железо умеет адресовать минимум N бит, значит, минимальный размер базового типа в реализации тоже разумно сделать N бит.
- Большинство процессоров способны адресовать минимум по 1 байту.

Параллельность и анализ программ

Потребность: нормально анализировать многопоточные программы.



Удобно думать, что операции исполняются по порядку, иногда переключаясь на другой поток

Sequential Consistency

(Лампорт, 1979): «Результат любого исполнения не отличим от случая, когда все операции на всех процессорах исполняются в некотором последовательном порядке, и операции на конкретном процессоре исполняются в порядке, обозначенном программой»

Reordering

Проблема: компилятор способен переставлять инструкции по своему усмотрению в целях оптимизации. Вот пример:

```
int a = 0, b = 0;
---------------------
r1 = a;
r2 = b;
```

```
int a = 0, b = 0;
---------------------
r2 = b;
r1 = a;
```

Казалось бы, просто поменяли местами два чтения.

Reordering

```
int a = 0, b = 0;
r1 = a;  | b = 2;
r2 = b;  | a = 1;
```

```
int a = 0, b = 0;
r2 = b;  |
         | b = 2;
         | a = 1;
r1 = a;  |
```

В первой версии программы в (r1, r2) могли быть (*, 2) или (0, *), где * - что-то, так как при SC “r2 = b” либо “a = 1” всегда выполняется последним.

В новой версии может получиться новый ответ - (r1, r2) = (1, 0).

SC сломалось.

Sequential Consistency - это конечно хорошо, но…

- Нельзя предсказать, какие оптимизации сломают SC, а какие нет. А вариаций большое множество. Причем эти оптимизации могут зависеть от архитектуры.
- Можно пытаться ставить барьеры памяти на каждую инструкцию, однако это будет стоить производительности.
- Поэтому модель решили немного **ослабить**. Дали возможность переупорядочивать инструкции, но не всегда и не везде.
- JMM дает определенные формальные правила, согласно которым можно понять, какие результаты разрешены языком.

Конфликт и гонка

- Доступы в память **конфликтуют**, если они работают с одним местом в памяти и хотя бы один из этих доступов – запись
- Программа **содержит гонку (data race)**, если два доступа конфликтуют и происходят одновременно (т.е. из нескольких потоков, не связаны синхронизацией)
- Программа с гонками нам дает непредсказуемые результаты

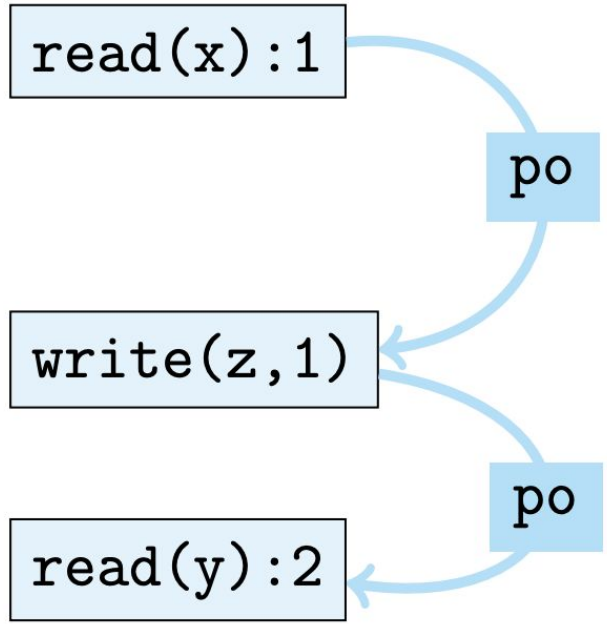
Немного формализма.

- JMM определяет, какие **результаты** разрешены языком
- JMM: программа определяет **действие (action).** С каждым действием ассоциируется значение. К примеру, read(x):3 значит, что было прочитано “3” из “x”.
- Действия связываются в **исполнения (execution)**, благодаря чему можем говорить о **порядках исполнения (execution order).**
- Если валидное исполнение дает какой-то результат, то этот результат **разрешен**
- **Суть:** у нас есть множество всевозможных исполнений, и в нем мы ищем то исполнение, которое оправдывает конкретный результат программы.

Program Order

- **Program Order (PO)** связывает действия внутри одного потока.
- **Intra-thread consistency:** исполнение в PO совместимо с действиями с изолированным исполнением в потоке.

```
if (x == 2) {
  y = 1;
} else {
  z = 1;
}
r1 = y;
```



Synchronization Actions

В слабой модели мы упорядочиваем не все операции, а какие-то избранные.

Есть набор действий синхронизаций:

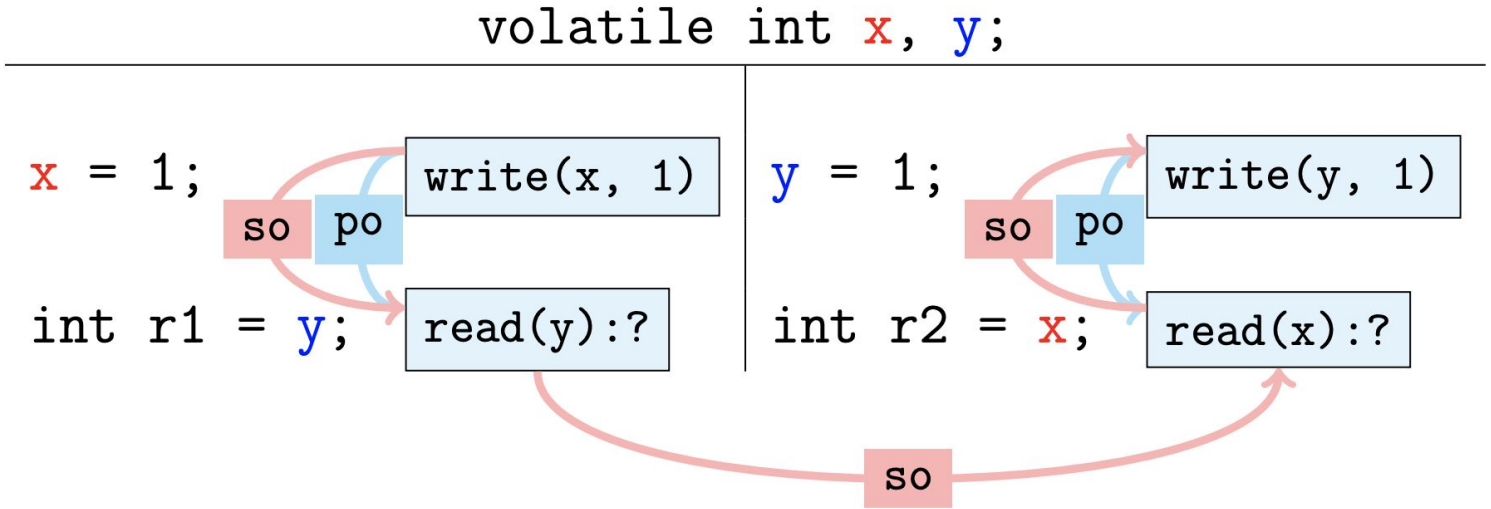
- Volatile read. A volatile read of a variable.
- Volatile write. A volatile write of a variable.
- Lock. Locking a monitor
- Unlock. Unlocking a monitor.
- Первое и последнее действие в потоке
- Действия, обнаруживающие прерывания потока (Thread.join(), Thread.isInterrupted() …)

Synchronization Order

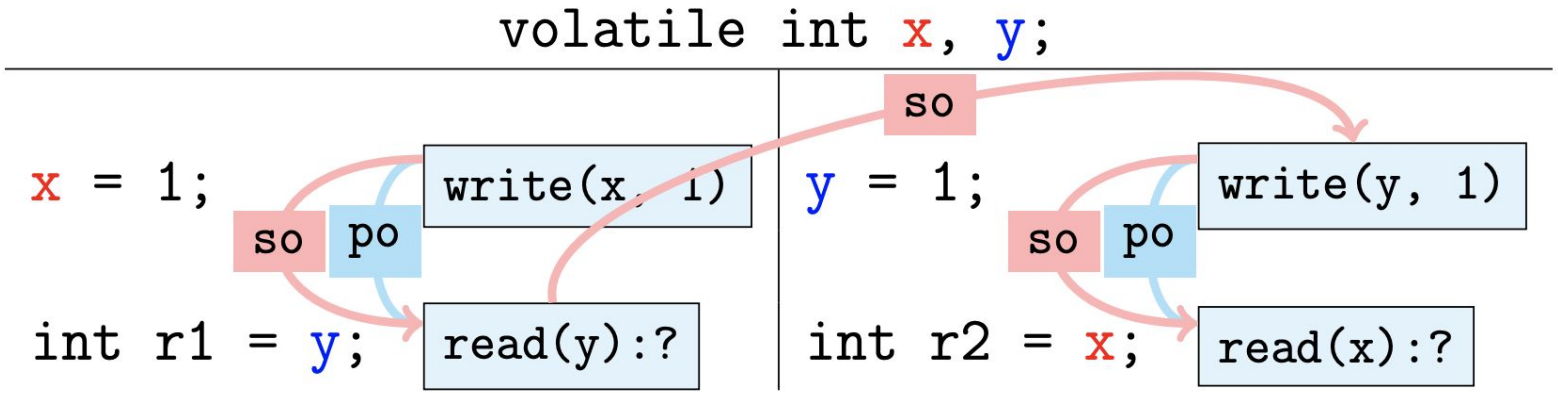
Действия синхронизации образуют **Synchronization Order** - порядок синхронизации.

Для одной программы можно построить множество SO, однако при этом есть правило - **SO должно быть согласовано с PO**. Исполнение ниже отбрасываем!

SO Consistency: все чтения в SO видят последние записи в SO

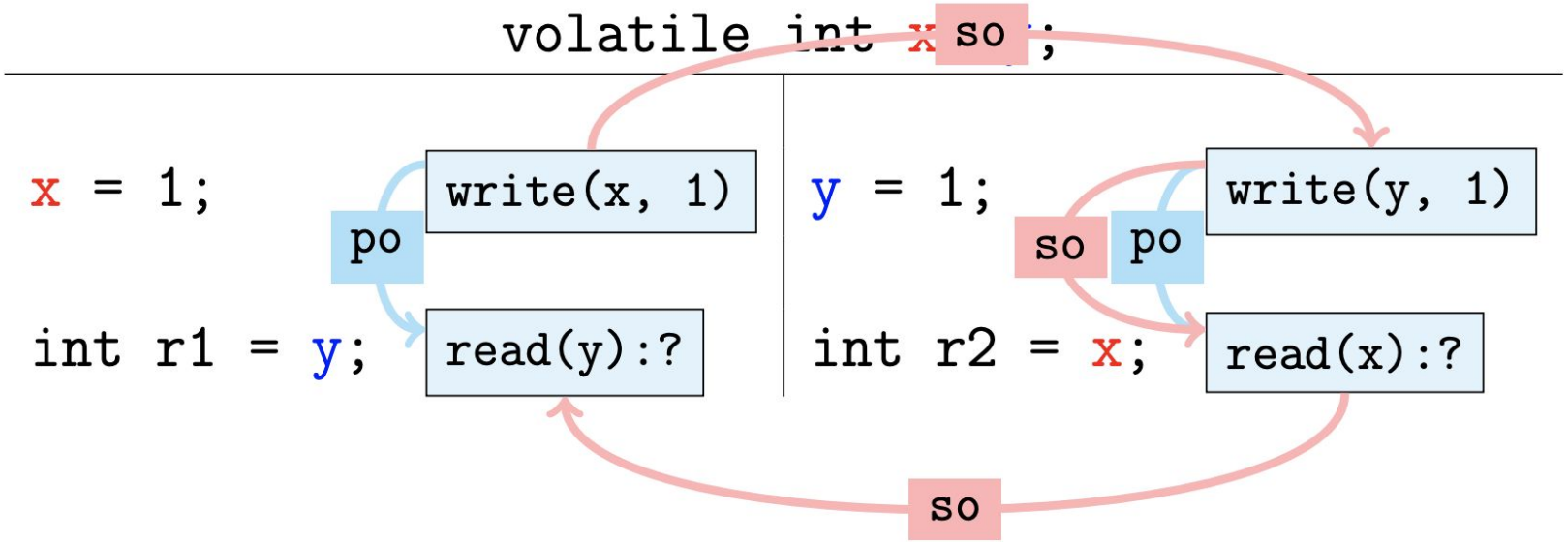


Synchronization Order



А вот это исполнение не отбрасываем. В результате (r1, r2) = (0, 1)

Synchronization Order



Этот вариант исполнения тоже подходит. (r1, r2) = (1, 1)

Synchronization Order и Sequential Consistency

- Заметим, что SO является SC. Благодаря консистентности с PO.
- Получаем, что последнее действие в SO будет последним в PO.

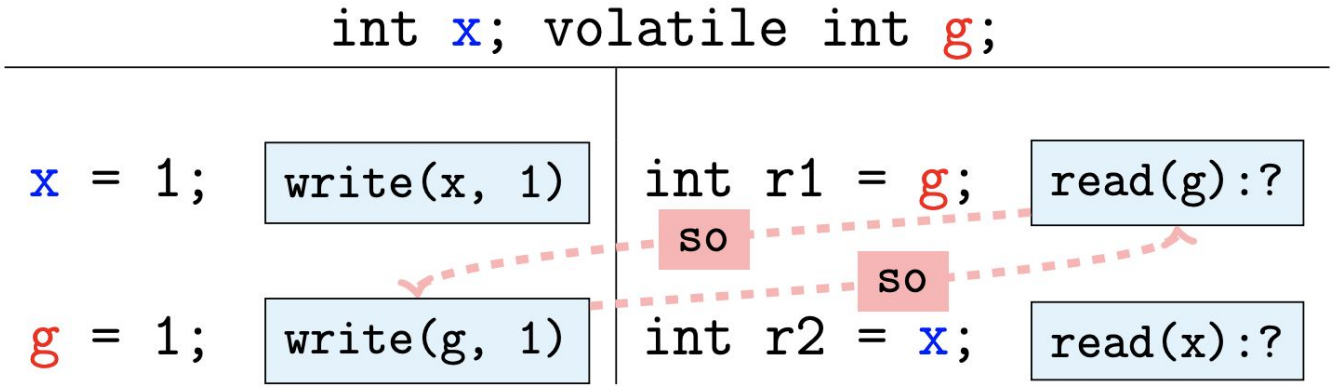
```
volatile int x, y;
```

`x = 1;`	`y = 1;`
`int r1 = y;`	`int r2 = x;`

Получается, модель с SO гарантирует, что в программе выше не может быть результата (r1, r2) = (0, 0)

Проблемы Synchronization Order

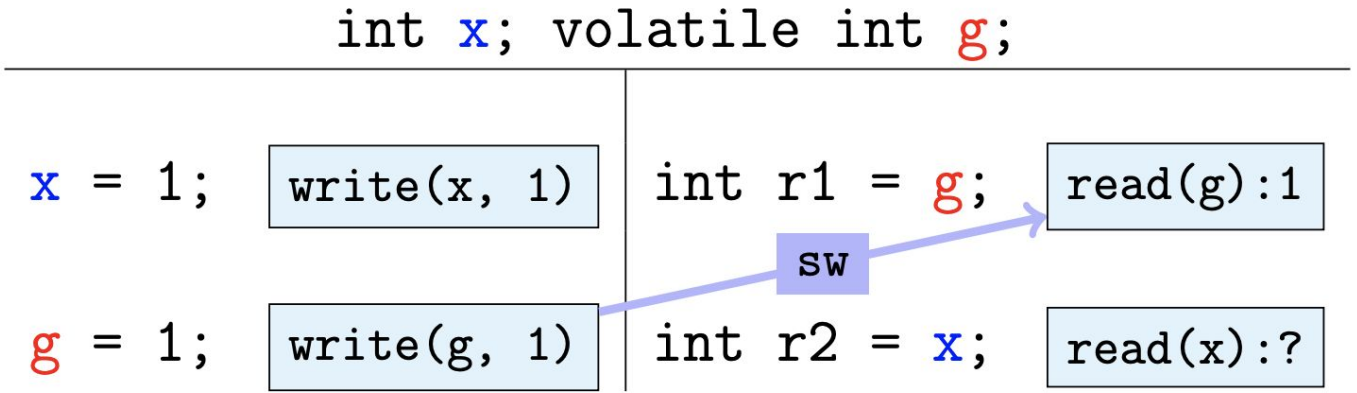
- Synchronization Order образуется только из действий синхронизации (SA). Но не все действия SA, и на них правил не накладывается!
- Все действия делать SA нельзя, мы не сможем делать оптимизации!
- Нужно что-то более ослабленное не для SA операций.



Запрещен ли результат (1, 0)? Мы упорядочили только g, а значит либо read(g):0 либо read(g):1. Ничего не мешает считать из x 0 или 1, вне зависимости от read(g):?

Synchronizes-With Order (SW)

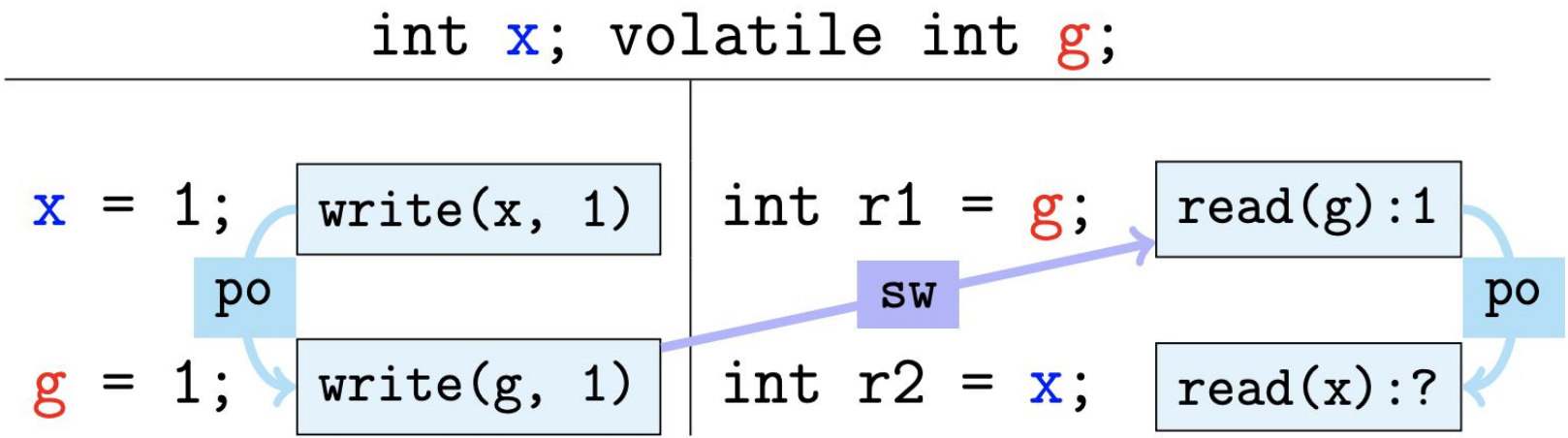
- SO умеет связывать действия между потоками, однако он слишком строг в плане ограничений (можно не упорядочивать ВСЕ SA)
- **SW** - Это подпорядок SO. Только мы теперь упорядочиваем конкретные read, write, unlock M, lock M etc. Если одно SA “видит” другое, то между ними есть SW.
- Получили мостик между потоками



Synchronizes-With Order (SW)

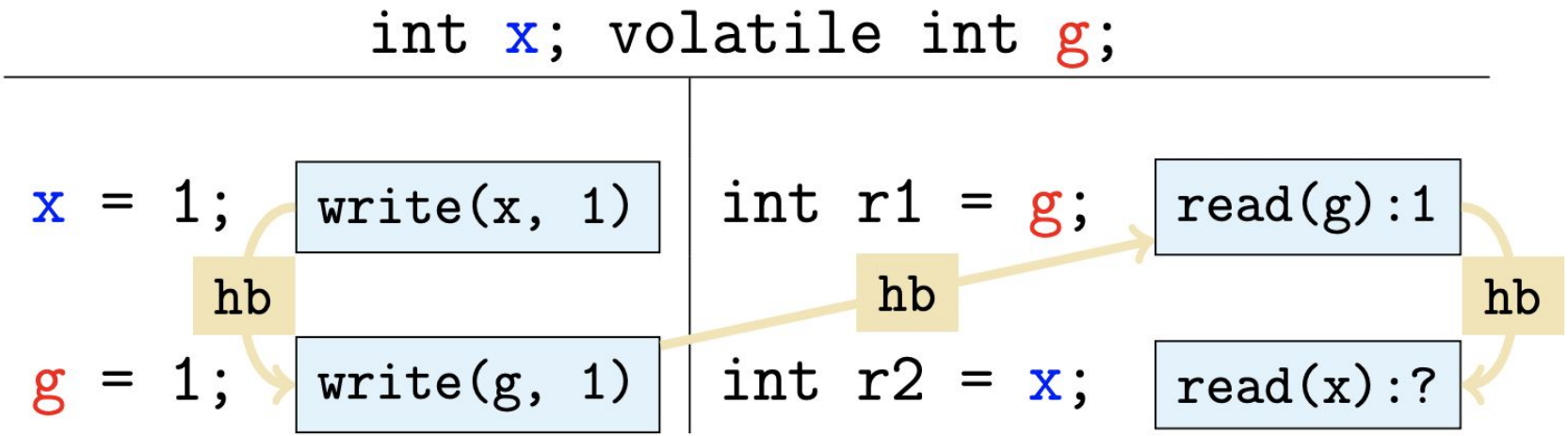
- An unlock action on monitor m **synchronizes-with** all **subsequent** lock actions on m (where "subsequent" is defined according to the synchronization order).
- A write to a volatile variable v **synchronizes-with** all **subsequent** reads of v by any thread (where "subsequent" is defined according to the synchronization order).
- An action that starts a thread **synchronizes-with** the first action in the thread it starts.
- The write of the default value (zero, false, or null) to each variable **synchronizes-with** the first action in every thread.
- The final action in a thread T1 **synchronizes-with** any action in another thread T2 that detects that T1 has terminated. (T2 вызывает T1.join() или T1.isAlive())
- If thread T1 interrupts thread T2, the interrupt by T1 **synchronizes-with** any point where any other thread (including T2) determines that T2 has been interrupted (by having an InterruptedException thrown or by invoking Thread.interrupted or Thread.isInterrupted)

Вернемся к примеру: достроим Program Order



Получили связь между действиями внутри потока

Объединим и замкнем - Happens-Before



Если объединить SW и PO, а потом взять транзитивное замыкание - получим **Happens-Before**

Happens-Before

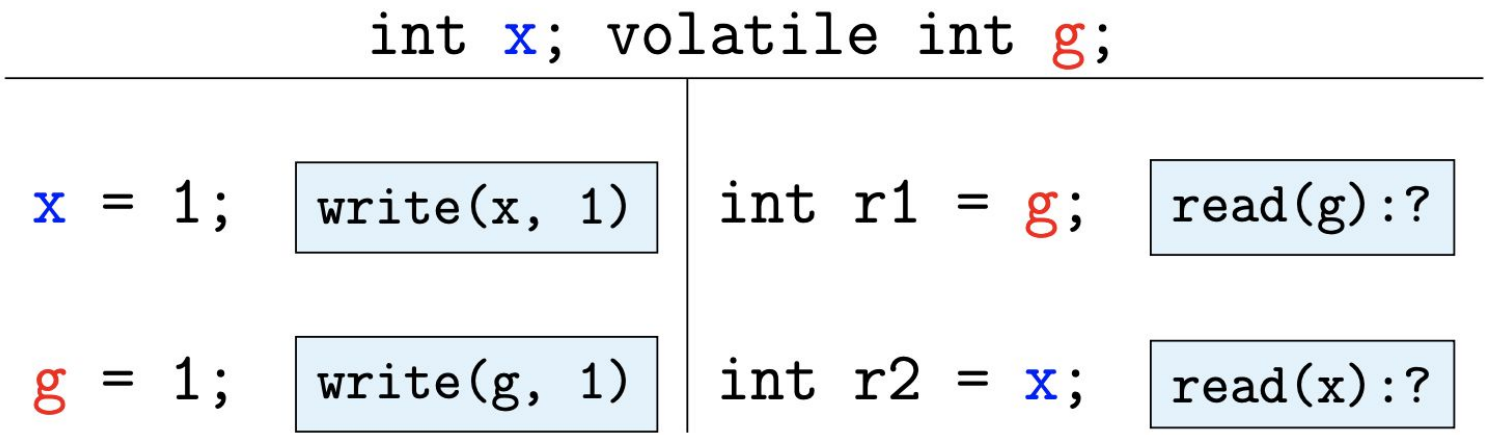
- If x and y are actions of the same thread and x comes before y in program order, then **hb(x, y)**.
- There is a happens-before edge from the end of a constructor of an object to the start of a finalizer (finalize method) for that object.
- If an action x **synchronizes-with** a following action y, then we also have hb(x, y).
- If hb(x, y) and hb(y, z), then hb(x, z)

Happens-Before

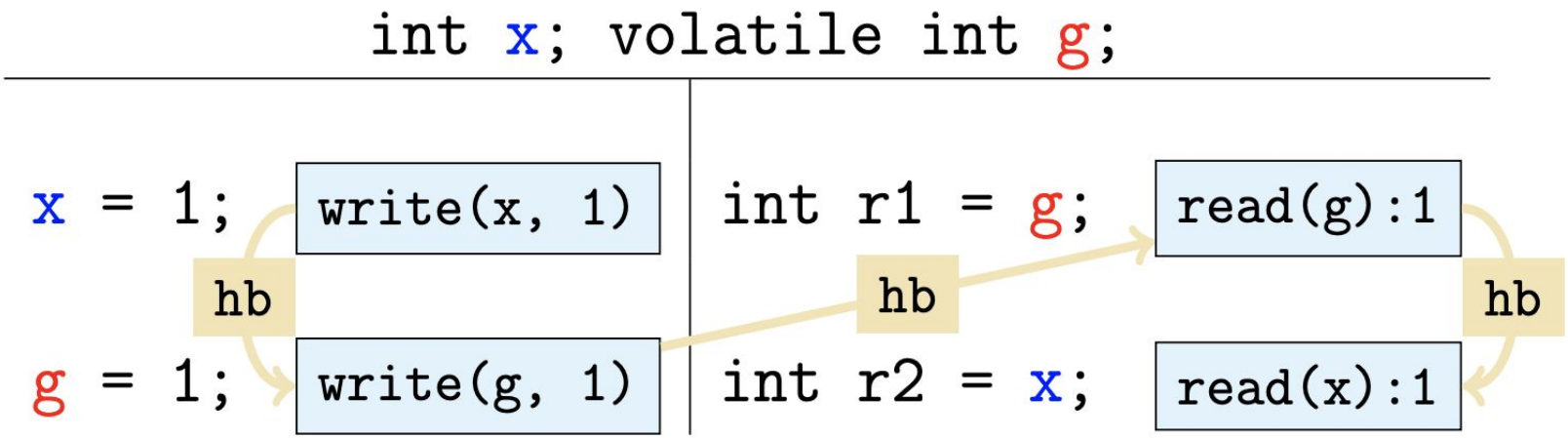
- An unlock on a monitor **happens-before** every subsequent lock on that monitor. (move out of synchronized block **happens-before** move in synchronized block on that monitor)
- A write to a volatile field **happens-before** every subsequent read of that field.
- A call to start() on a thread **happens-before** any actions in the started thread.
- All actions in a thread **happen-before** any other thread successfully returns from a join() on that thread.
- The default initialization of any object **happens-before** any other actions (other than default-writes) of a program.

Happens-Before Consistency

- HB приносит новое правило для чтений:
- **Чтения видят либо последнюю запись через Happens-Before, либо что-то еще через состояние гонки.**
- Посмотрим на пример ниже

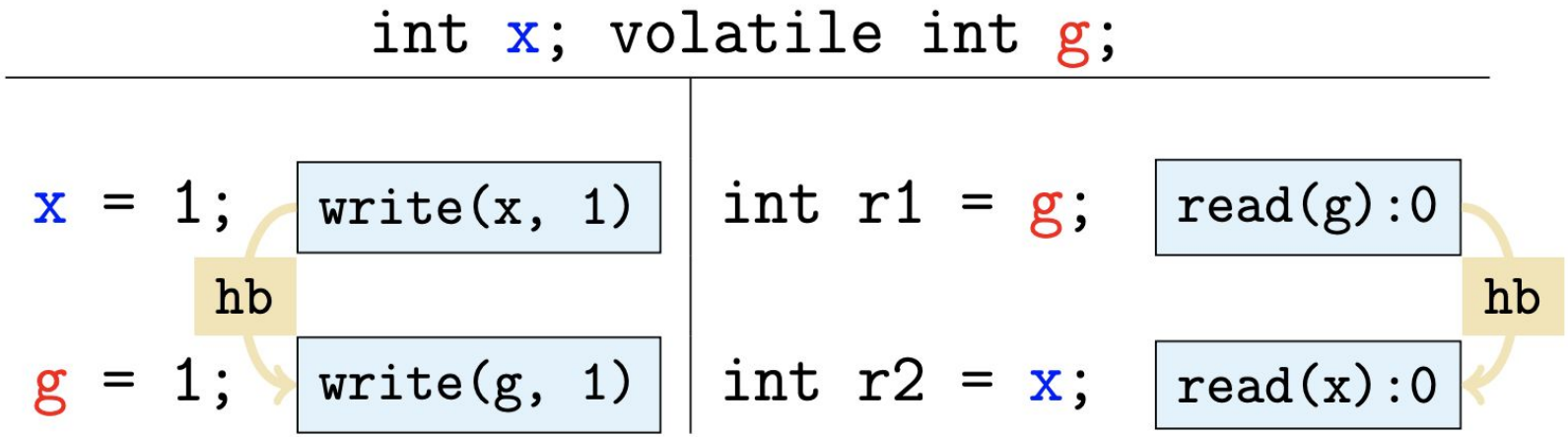


Happens-Before Consistency



Здесь результат (r1, r2) = (1, 1), исполнение HB consistent

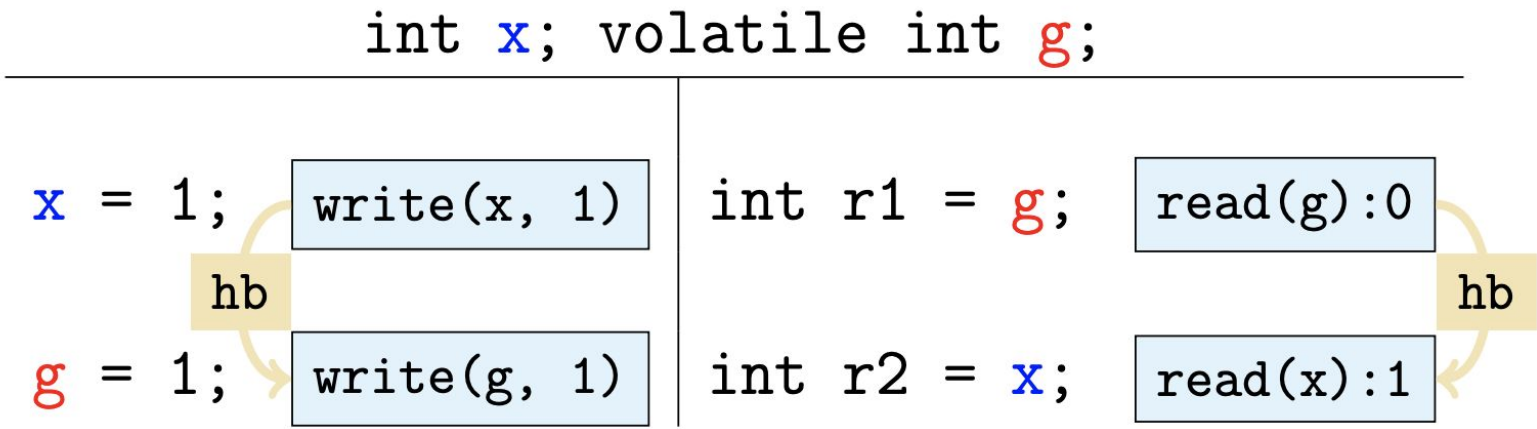
Happens-Before Consistency



Здесь результат (r1, r2) = (0, 0), исполнение HB consistent.

read(g) случился раньше (по SO) и просто не увидел write(g, 1)

Happens-Before Consistency

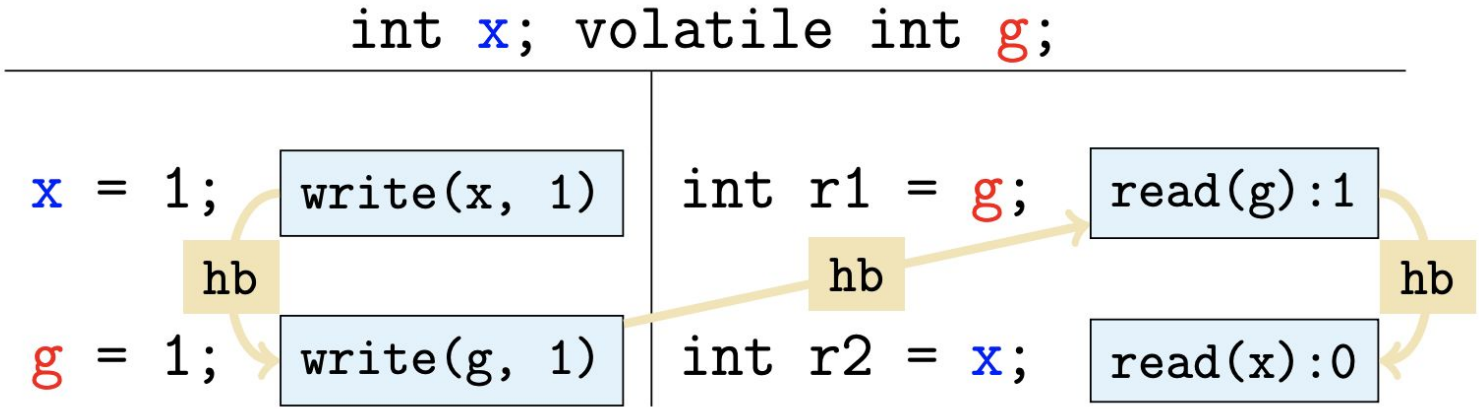


Здесь результат (r1, r2) = (0, 1), исполнение HB consistent.

Такой результат вышел через гонку между write(x, 1) и read(x)

(конфликтующие доступы в одну память, не связаны HB - **гонка**)

Happens-Before Consistency



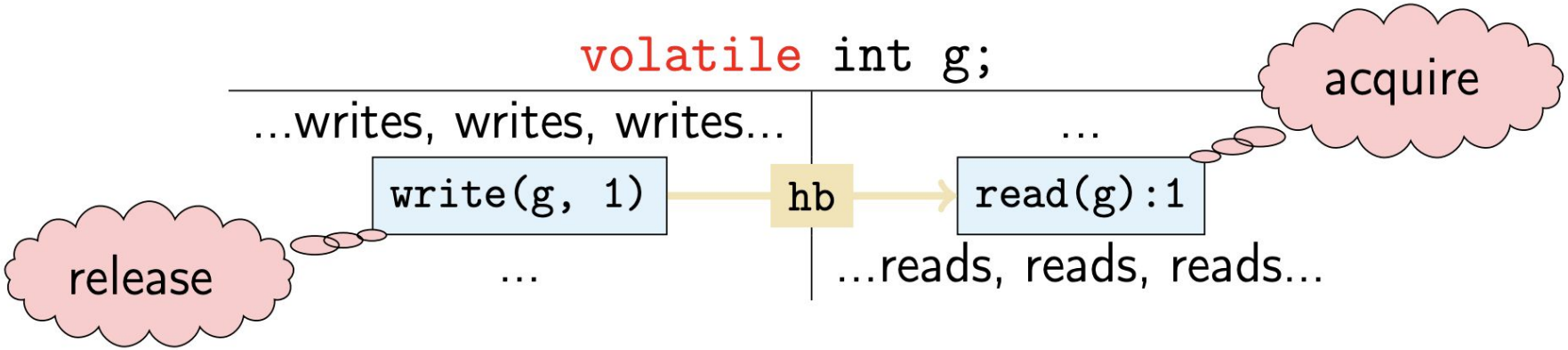
Здесь результат (r1, r2) = (1, 0), исполнение **не HB consistent**!

read(x) не мог считать 0 в данном случае, результат (1, 0) отбрасываем!

SequentialConsistency-DataRaceFree (SC-DRF)

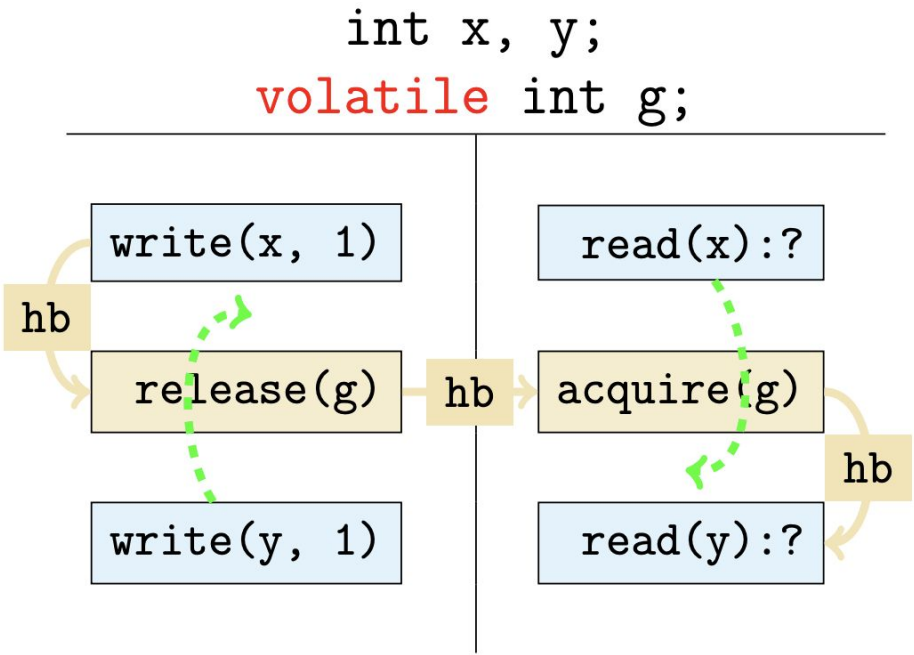
- A program is **correctly synchronized** if and only if **all sequentially consistent executions are free of data races**
- **SC-DRF: If a program is correctly synchronized, then all executions of the program will appear to be sequentially consistent**
- **Вывод:** В программе нет гонок ⇒ все чтения видят упорядоченные записи ⇒ результат исполнения программы можно объяснить каким-нибудь SC-исполнением
- **Операции над локальными данными не будут ломать SC. Операции над глобальными данными - с синхронизацией не будут ломать SC.**

Публикация



- Работает только на одной и той же переменной, одном и том же мониторе
- Работает только если мы увидели ту самую release-запись
- Всегда парные действия! Нельзя сделать release в одной стороне, и не делать acquire в другой.

Какие перестановки разрешены?



- Можно вносить инструкции в HB (за release и после acquire)
- Стрелка слева: read(y) все равно может увидеть write(y, 1) через гонку
- Стрелка справа: read(x) все равно может увидеть write(x, 1) через гонку
- Но наоборот нельзя! Потенциально выносим из HB. Мы не знаем, есть ли read(x) после acquire, рискуем потерять запись, и наоборот.

Безопасная публикация

- Потребность: сконструировать объект так и опубликовать для других потоков.
- Публикация обычно подразумевает запись в переменную ссылку на новый объект, и переменная видна другим потокам.
- Проблема: если опубликовать объект неправильно, без надлежащей синхронизации, другой поток может увидеть частично сконструированный объект (объект утек). Может привести к тому, что поток увидит объект в недопустимом состоянии.

Безопасная публикация

Публикация будет безопасной, если запись в ссылочное поле **happens-before** чтение из этого поля. Тогда второй поток увидит все поля опубликованного объекта, инициализированного этим потоком.

- инициализация ссылки из статического анализатора (гарантия JMM)
- использовать volatile или AtomicReference
- сохранение ссылки в final поле надлежаще сконструированного объекта (нет утечки ссылки this при конструировании)
- сохранение ссылки в поле, которое защищено замком (монитором)

Безопасная статическая инициализация

- Если у нас есть статические поля, которые мы инициализируем (или просто поля, которые мы инициализируем в блоке инициализации), то работа с ними накладывает дополнительные гарантии потокобезопасности от JVM.
- JVM делает lock, и каждый поток тоже как минимум с целью загрузки класса, а значит записи в память во время статической инициализации будут видны всем потокам.
- То есть, статически инициализируемые объекты не требуют явной синхронизации ни при конструировании, ни при ссылке на них.
- Только если мутируем - читателю и писателю все еще нужна синхронизация.

final и безопасность инициализации

- final поля не могут быть изменены
- Позволяет строить **немутируемые объекты** (состояние не поменять после конструирования, все поля финальны, сам объект надлежаще сконструирован т.е. ссылка this не ускользает)
- Для немутируемых объектов есть гарантия **безопасности при инициализации (initialization safety)** при совместном использовании. То есть, немутируемые объекты можно использовать без синхронизации.
- Главное - финальные поля устанавливать в конструкторе и не сливать ссылку. Тогда другие потоки будут видеть правильно сконструированную версию объекта.

Источники

- Java Language Specification (link)
- Java Memory Model Pragmatics : Aleksey Shipilëv, (link)
- Java Concurrency in Practice : Goetz, Brian