ORACLE

Java Memory Model

прагматика модели

Алексей Шипилёв
aleksey.shipilev@oracle.com, @shipilev

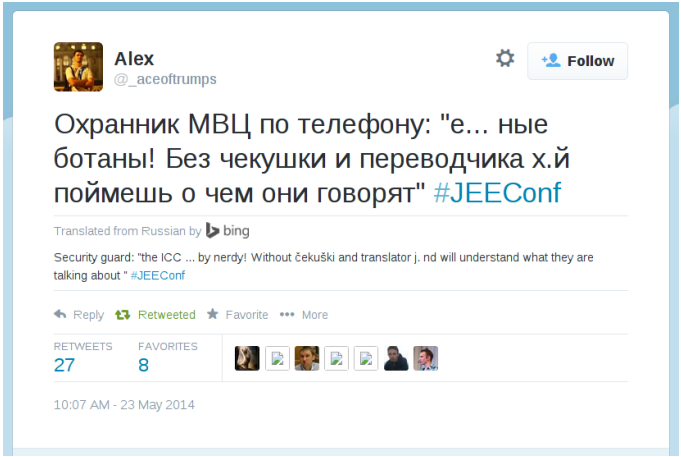
The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Введение

Введение: Дисклеймеры

1. Доклад рассказывает про Java Memory Model
   (Уходите.)
2. ...с помощью боли, математики и такой-то матери
   (Ещё есть шанс уйти.)
3. Доклад долгий, сложный, беспощадный. Серьёзно.
   (Нет, правда, никто не держит.)
4. Доклад не про устройство джавовой кучи! Не про GC!
   (Доклады про плюшевый энтерпрайз в других залах)

Введение: Картинка-детектор



Java ORACLE

Введение: Абстрактные машины

- Любой язык программирования определяет свою семантику через поведение абстрактной машины, выполняющей программу на этом языке.
- Спецификация языка = спецификация абстрактной машины[1]

Показательный пример:
`Brainfuck`[2] = типичный ассемблер для машины Тьюринга

[1] Java ≠ Java bytecode ⇒ спецификация Java ≠ спецификация JVM

[2] `http://en.wikipedia.org/wiki/Brainfuck`

Введение: Модель памяти

- Часть спецификации абстрактной машины: модель того, как работает хранилище данных = *модель памяти*
- Оказывается, что модели памяти достаточно ответить на один простой вопрос...

Введение: Модель памяти

- Часть спецификации абстрактной машины: модель того, как работает хранилище данных = *модель памяти*
- Оказывается, что модели памяти достаточно ответить на один простой вопрос...

Какие значения может прочитать конкретный `read` в программе?

Введение: В последовательных программах...

- Исполняем инструкции языка одну за одной? Тогда модель памяти очевидна:

  «Чтения, следующие за записями в программе, должны видеть ранее записанные значения»[3]

- Часто под «моделью памяти» имеют в виду «модель памяти, покрывающая семантику многопоточных программ»

[3] e.g. для C99: ISO/IEC 9899:1999, «5.1.2.3 Program execution»

Введение: ...тоже не всё просто

- Хрестоматийный пример для C 89/99:

```
int i = 5; (.)
i = (++i + ++i); (.)
assert (13 == i); (.) // FAILS
```

- Отсутствие *точек следования*[4] в выражении приводит к неопределённому поведению (между (.) (.) может быть что угодно)
- Модели памяти нужны в том числе для того, чтобы судить о поведении однопоточных программ

[4] ISO/IEC 9899:1999, «Annex C: Sequence Points»

Введение: Coming back to reality

Реализации ЯП делают одну из двух вещей:

1. Напрямую эмулируют абстрактную машину на входной программе и существующей машине (интерпретация)
2. Специализируют абстрактную машину входной программой, и выполняют результат на существующей машине (компиляция)

Реализациям нужно вложиться в специфицированное поведение абстрактной машины ЯП.

Введение: Be careful what you wish for

Модель памяти = trade-off между
долбанутостью программирования *на языке*,
долбанутостью *быстрой и корректной реализации языка*,
и долбанутостью *хардвара*

- Мечтать не вредно: можно потребовать много удобных штук
- ...вопрос в том, не уйдёт ли на создание подходящей реализации ЯП и железа к нему стотыщмильонов лет?

Введение: логика повествования

Мы попробуем:

1. Показать, что JMM нам нужна
2. Обозначить наши желания
3. Посмотреть, что нам реально доступно
4. Понять, как спецификация балансирует между (2) и (3)
5. Заглянуть, как работают (консервативные) реализации JMM

Формальные требования JMM будут вот в такой рамке

Access atomicity

Access atomicity: Сказка

Хочется:

Атомарность доступа к базовым типам

Т.е. для любого базового типа T:

`T t = V1;`	
`t = V2;`	`T r1 = t;` `assert (r1 ∈ {V1, V2})`

Access atomicity: Реальность

Нужна поддержка со стороны железа, чтобы оно действительно делало атомарные чтения/записи

Засады:

- Отсутствие хардварных операций для крупных чтений: как прочитать 8-байтный `long` на 32-битном x86? А на 32-битном ARM-е?
- Требования подсистемы памяти: в примеру, при пересечении кеш-лайна на x86 атомарность теряется

Access atomicity: Компромисс (часть 1/2)

Чтения/записи атомарны для всего, кроме `long` и `double`

`volatile long` и `volatile double` атомарны

- Большая часть железа в 2004 умела читать до 32 бит за раз, с 64-битными чтениями пришлось мириться
- Ссылки имеют битность, соответствующую машинной
- Можем вернуть атомарность (подчёркивая возможный performance penalty)

Access atomicity: Компромисс (часть 2/2)

Почти везде невыровненное чтение теряет атомарность
(и уж точно теряет производительность)

- Реализация вынуждена выравнивать типы по их длине:

```
o.o.j.samples.JOLSample_02_Alignment.A[5]
 OFFSET  SIZE  TYPE DESCRIPTION
      0    12       (object header)
     12     4       (alignment/padding gap)
     16     8  long A.f
```

[5] http://openjdk.java.net/projects/code-tools/jol/

Access atomicity: Quiz

Что напечатает?

```
AtomicLong al = new AtomicLong();
```

`al.set(-1L);`	`println(al.get());`

Access atomicity: Quiz

Что напечатает?

```
AtomicLong al = new AtomicLong();
```

`al.set(-1L);`	`println(al.get());`

Почему не `0 x FFFF FFFF 0000 0000`?

Access atomicity: Quiz

Что напечатает?

```
AtomicLong al = new AtomicLong();
```

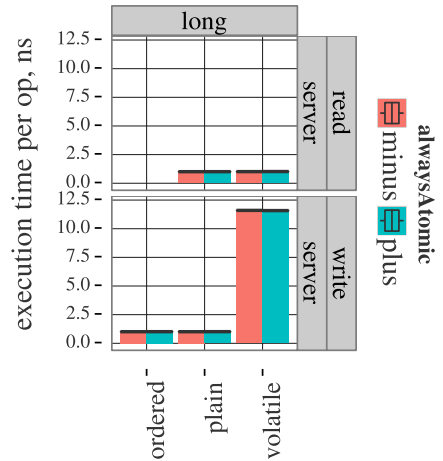
`al.set(-1L);`	`println(al.get());`

Почему не `0 x FFFF FFFF 0000 0000`?
Никакой магии: «`volatile long`» внутри гарантирует.

Access atomicity: JMM 9

- Исключения для `long`/`double` вызваны прагматикой 2004 г.
  - Повсеместно распространены 32-битные x86
  - Древние, дремучие ARM'ы и PowerPC'ы
- В 2014 году уже всё гораздо лучше!
  - В серверном мире остались вообще 32-битные машины?
  - Даже на 32-битных давно есть 64-битные (векторные) инструкции
  - На большинстве платформ `long`/`double` де-факто атомарны
  - ...но мы всё равно заставляем писать `volatile`, ибо WORA
- Вопрос: Может, пора выпилить эти исключения из спецификации?

Access atomicity: JMM 9



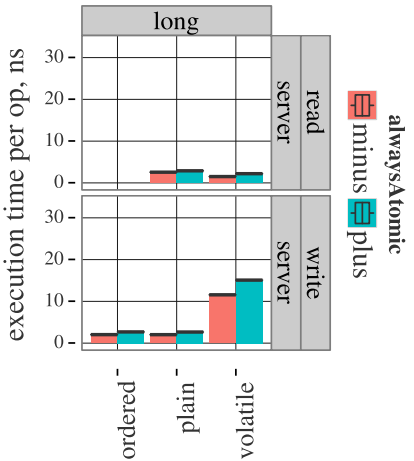
x86, Ivy Bridge, 64-bit:

Никакой разницы:

- `double` уже давно атомарен
- `long` работает на нативной битности

[6] `http://shipilev.net/blog/2014/all-accesses-are-atomic/`

Access atomicity: JMM 9



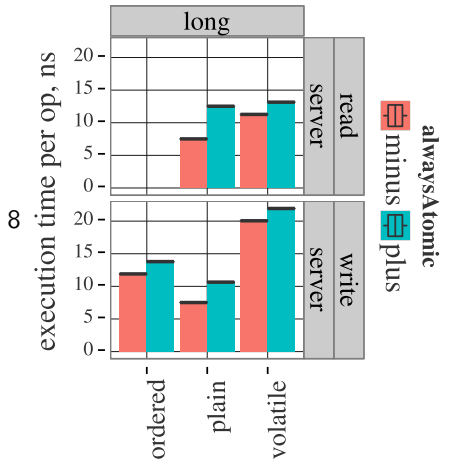
x86, Ivy Bridge, 32-bit:

Разницы чуть-чуть:

- `double` уже давно атомарен
- `long` на живых векторных инструкциях

[7] `http://shipilev.net/blog/2014/all-accesses-are-atomic/`

Access atomicity: JMM 9



ARMv7, Cortex-A9, 32-bit:

Разницы чуть-чуть:

- `double` уже давно атомарен
- `long` на живых векторных инструкциях

[8] `http://shipilev.net/blog/2014/all-accesses-are-atomic/`

Access atomicity: Value types

- Многие хотят value types в Java. Кроме всяких бонусов, они вносят лёгкий аромат неадеквата в модель памяти.
- К примеру, C/C++11 требует атомарность для любого POD:

```
typedef struct TT {
    int a, b, c, ..., z; // 104 bytes
} T;
std::atomic<T> atomic();
```

`atomic.set(new T());`	`T t = atomic.get();`

Access atomicity: Value types

- Многие хотят value types в Java. Кроме всяких бонусов, они вносят лёгкий аромат неадеквата в модель памяти.
- К примеру, C/C++11 требует атомарность для любого POD:

```
typedef struct TT {
    int a, b, c, ..., z; // 104 bytes
} T;
std::atomic<T> atomic();
```

`atomic.set(new T());`	`T t = atomic.get();`

- Реализация **вынуждена** «приседать» и на `set()`, и на `get()`

Word tearing

Word tearing: Сказка

Хочется:

Незалежность операций над независимыми элементами (полями, элементами массивов и т.п.). Например:

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

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

Word tearing: Реальность

Нужна поддержка со стороны железа, чтобы оно действительно делало независимые чтения/записи

Засады:

- Отсутствие хардварных операций для мелких чтений/записей: как атомарно записать 1-битный `boolean`, если атомарно можно записать N ($N \geq 8$) бит?

Word tearing: Решение

Word tearing запрещён

- Большая часть железа умеет адресовать от 8 бит за раз
- Если железо умеет адресовать минимум N бит, значит, минимальный размер базового типа *в реализации* тоже разумно сделать N бит
- На большинстве платформ все типы не теряют память (кроме 8-битного `boolean`)

Word tearing: Experimental Proof

Объекты выровнены на 8 байт.
Всё, кроме `boolean`, точного размера под диапазон значений:

```
$ java -jar jol-internals.jar ...
 Running 64-bit HotSpot VM.
 Using compressed references with 3-bit shift.
 Objects are 8 bytes aligned.
 Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
 Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
```


Word tearing: Quiz

Что напечатает?

```
BitSet bs = new BitSet();
```

`bs.set(1);`	`bs.set(2);`	
`<term>`	`<term>`	`<join both>`
		`println(bs.get(1));`
		`println(bs.get(2));`

[9] Есть ли хоть одна реализация, которая напечатает (F, F)?

Word tearing: Quiz

Что напечатает?

```
BitSet bs = new BitSet();
```

`bs.set(1);`	`bs.set(2);`	
`<term>`	`<term>`	`<join both>`
		`println(bs.get(1));`
		`println(bs.get(2));`

Напечатает любую[9] из комбинаций (T, T), (F, T), (T, F).

[9] Есть ли хоть одна реализация, которая напечатает (F, F)?

Word tearing: Bit fields

- Многие хотят `struct`'ы в Java, или вообще способ контролировать layout объектов. Ага, удачи, сэмулируй это:

```
typedef struct TT {
    unsigned a:7;
    unsigned b:3;
} T;
```

T t;	
`t.a = 42;`	`r1 = t.b;`

Word tearing: Bit fields

- Многие хотят `struct`'ы в Java, или вообще способ контролировать layout объектов. Ага, удачи, сэмулируй это:

```
typedef struct TT {
    unsigned a:7;
    unsigned b:3;
} T;
```

T t;	
`t.a = 42;`	`r1 = t.b;`

- Реализация **вынуждена** «приседать» и на записи `a`, и на чтении `b`.

Word tearing: Bit fields

- Многие хотят `struct`'ы в Java, или вообще способ контролировать layout объектов. Ага, удачи, сэмулируй это:

```
typedef struct TT {
    unsigned a:7;
    unsigned b:3;
} T;
```

T t;	
`t.a = 42;`	`r1 = t.b;`

- Реализация **вынуждена** «приседать» и на записи a, и на чтении b. (C/C++11 на это феерически забил)

Word tearing: JMM 9

MOVE
ALONG
NOTHING
TO SEE
HERE

SC-DRF

SC-DRF: Сказка

Хочется:

Простой способ анализировать приложения.

К примеру:

`opA();`	`opD();`
`opB();`	`opE();`
`opC();`	`opF();`

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

SC-DRF: Сказка (формальнее)

Sequential Consistency (SC):

(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»

SC-DRF: Сказка (формальнее)

Sequential Consistency (SC):

(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»

SC-DRF: Сказка (формальнее)

SC – иезуитское определение:

- Программу можно сильно переколбасить, лишь бы нашёлся нужный порядок в оригинальной программе, который приводит к тому же SC-результату

`int a = 0, b = 0;`	
`a = 1;`	`b = 2;`
`print(b);`	`print(a);`

...	
`print(2);`	`print(1);`

SC-DRF: Реальность

- Отношения оптимизаций и модели памяти можно выразить через перестановки чтений/записей
- Можно ли осуществить это преобразование?

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

→

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

SC-DRF: Реальность

`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 исходной программе при SC обязательно либо «`r2 = b`», либо «`a = 1`» должно быть последним, а значит, `(r1, r2)` либо `(*, 2)`, либо `(0, *)`.
- Новая программа приводит к `(r1, r2) = (1, 0)`

SC-DRF: Реальность

Sequential Consistency - очень привлекательная модель.
Даёшь её в массы в XVII пятилетке!

- Очень сложно определить, какие оптимизации можно делать, не нарушая при этом SC
- *В теории*, какой-нибудь Глобальный МегаОптимизатор (ГМО) может сделать такой анализ
- *На практике* и рантаймы, и железо получаются исключительно без ГМО $\Rightarrow$ большая часть оптимизаций запрещена

SC-DRF: Реальность в хардваре

[10] http://en.wikipedia.org/wiki/Memory_ordering

SC-DRF: Реальность в хардваре

Процессоры жёстко спекулируют и переупорядочивают операции (ради производительности!)

Memory ordering in some architectures[2][3]

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	zSeries
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	Y

10

[10] http://en.wikipedia.org/wiki/Memory_ordering

SC-DRF: немного определений

- 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 (из нескольких потоков, не связаны синхронизацией)

Программа с гонками рождает непредсказуемые результаты!
Язык вынужден давать механизмы упорядочивания доступов.

SC-DRF: компромисс

Нам нужна более слабая модель!
(Вспоминаем про trade-off-ы)

Если подойти к делу основательно, то:

- Разрешим оптимизации в рантаймах и хардваре
- Позволим упорядочивать операции без взрыва мозга девелоперов
- Спецификация будет невменяема, но чуть менее чем наполовину

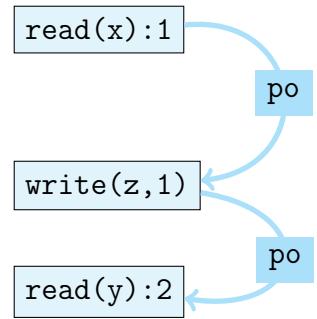
SC-DRF: JMM Formalism TL;DR;

- JMM специфицирует, какие *результаты* разрешены языком
- JMM определяет *действия*. Действия тащат на себе значения: e.g. «read(x):1» означает, что мы действительно прочитали «1» из «x». Результат конкретной программы разрешён, если есть действие, которое читает нужное нам значение
- Действия связаны в *исполнения*, которые дают ещё и порядки над действиями ($\xrightarrow{po}$, $\xrightarrow{so}$, $\xrightarrow{sw}$, $\xrightarrow{hb}$). Если валидное исполнение даёт нам какой-то результат, то этот результат разрешён

SC-DRF: Program Order

Program Order (PO) связывает действия внутри одного потока

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

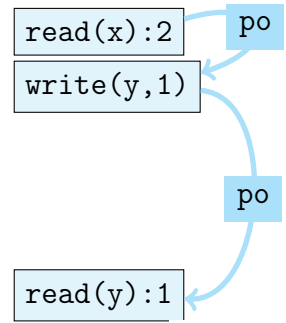


SC-DRF: Program Order

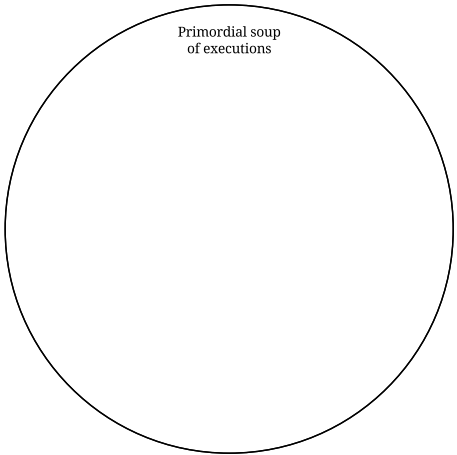
PO – линейный порядок

(Операторы программы не образуют линейный порядок вообще!)

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



SC-DRF: По дороге к валидным исполнениям



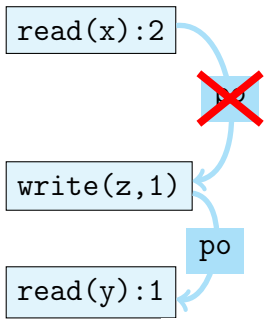
Где-то во множестве всех возможных исполнений может быть исполнение, которое оправдывает конкретный результат конкретной программы.

Вся суть JMM в том, чтобы найти такое исполнение.

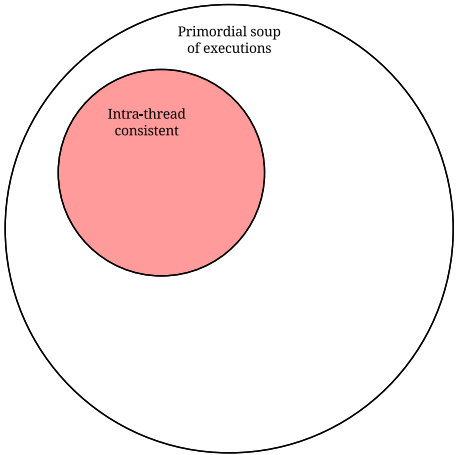
SC-DRF: Program Order

Intra-thread consistency: для каждого потока действия в PO те же самые, что получились бы в изолированном исполнении

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



SC-DRF: PO constraints



Intra-thread consistency выделяет исполнения, которыми можно пользоваться для конкретной программы.

Это единственное место, где в JMM протекает информация о последовательной программе!

SC-DRF: Synchronization Actions

Слабые модели памяти упорядочивают не все операции,
а только некоторые избранные.

Synchronization Actions (SA):

- volatile read, volatile write
- lock monitor, unlock monitor
- (синтетические) первое и последнее действие в потоке
- действия, запускающие поток
- действия, обнаруживающие останов потока (Thread.join(), Thread.isInterrupted() и т.п.)

SC-DRF: Synchronization Order

Synchronization Actions образуют Synchronization Order (SO)

- SO – линейный порядок
  - каждый поток видит SA в одном и том же порядке
  - это единственный порядок действий, который нужно иметь линейным
- Порядок действий в SO совместен с PO
  - SA в одном потоке видны в порядке программы
  - инварианты парности lock/unlock выдержаны
- **Synchronization order consistency**:
  Все чтения в SO видят последние записи в SO

SC-DRF: SO constraints, Dekker

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

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

Подумаем: какие (r1, r2) допустимы?

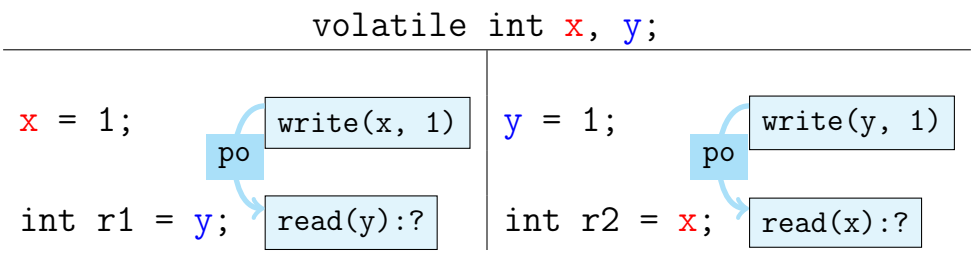
SC-DRF: SO constraints, Dekker

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

`x = 1;`	`write(x, 1)`	`y = 1;`	`write(y, 1)`
`int r1 = y;`	`read(y):?`	`int r2 = x;`	`read(x):?`

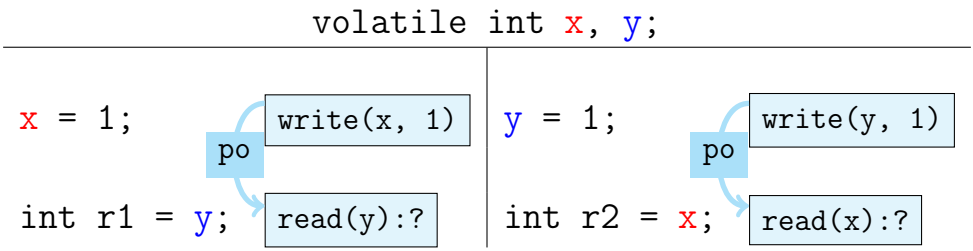
Ввиду intra-thread consistency, нам стоит рассматривать исполнения, состоящие из четырёх действий

SC-DRF: SO constraints, Dekker



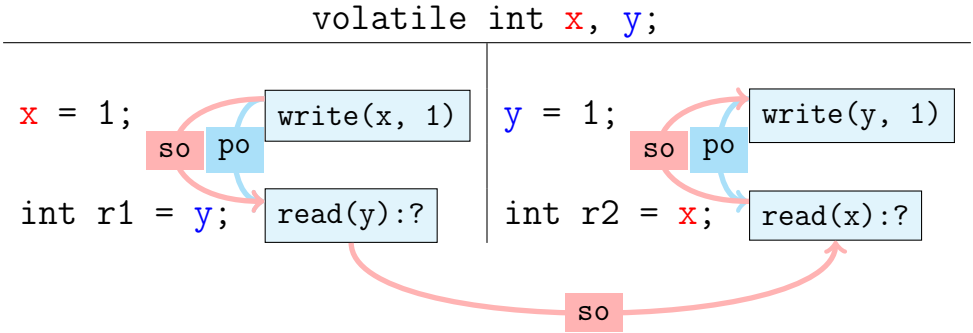
Действия в потоках связаны $\xrightarrow{po}$

SC-DRF: SO constraints, Dekker



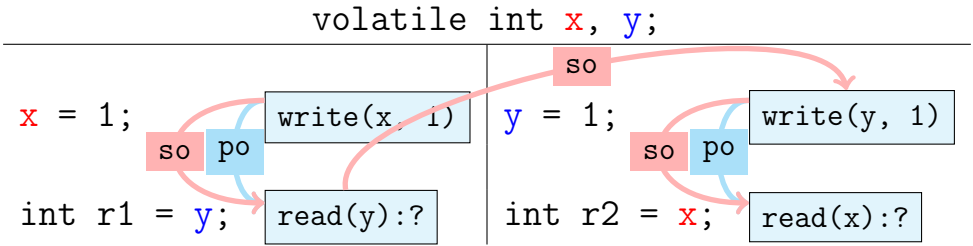
Все эти действия работают с `volatile`, следовательно, все они в SO. Однако, этот порядок можно наложить разными вариантами...

SC-DRF: SO constraints, Dekker



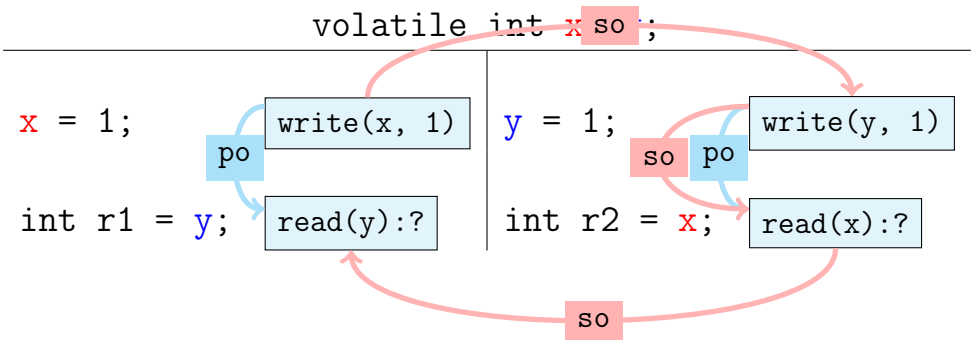
Вариант 1: $\xrightarrow{so}$ не совместен с $\xrightarrow{po}$, исполнение можно отбросить

SC-DRF: SO constraints, Dekker



Вариант 2: $\xrightarrow{so}$ совместен с $\xrightarrow{po}$, а ввиду SO consistency чтения обязаны увидеть `read(y):0` и `read(x):1`

SC-DRF: SO constraints, Dekker



Вариант 3: $\xrightarrow{so}$ совместен с $\xrightarrow{po}$, а ввиду SO consistency чтения обязаны увидеть `read(y):1` и `read(x):1`

SC-DRF: Наблюдение: SA are SC!

Synchronization actions – sequentially consistent!

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

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

- Последнее действие в PO должно быть последним в SO
- Следовательно, `read(y):?` или `read(x):?` будет последним, и точно увидит запись
- Следовательно, `(r1, r2) = (0, 0)` запрещен

SC-DRF: SO constraints, IRIW

Ещё классика, «Independent Reads of Independent Writes» (IRIW):

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

`x = 1;`	`y = 1;`	`int r1 = y;` `int r2 = x;`	`int r3 = x;` `int r4 = y;`

- Все исполнения, дающие $(r1, r2, r3, r4) = (1, 0, 1, 0)$ ломают или SO consistency, или совместность SO и PO

SC-DRF: SO constraints, IRIW

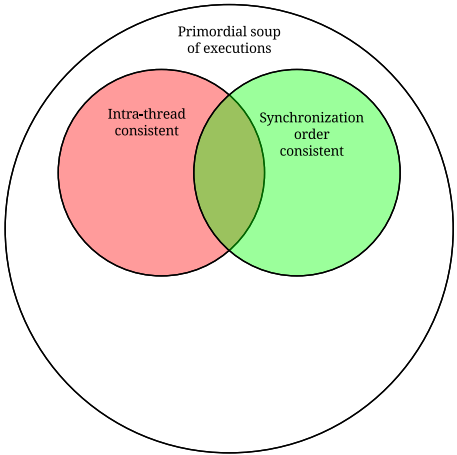
Ещё классика, «Independent Reads of Independent Writes» (IRIW):

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

`x = 1;`	`y = 1;`	`int r1 = y;` `int r2 = x;`	`int r3 = x;` `int r4 = y;`

- Все исполнения, дающие $(r1, r2, r3, r4) = (1, 0, 1, 0)$ ломают или SO consistency, или совместность SO и PO
- Посыпьте Java-программу `volatile`-ами, и sequential consistency к вам вернётся!

SC-DRF: SO constraints



Synchronization order consistency даёт SC-скелет для программы.

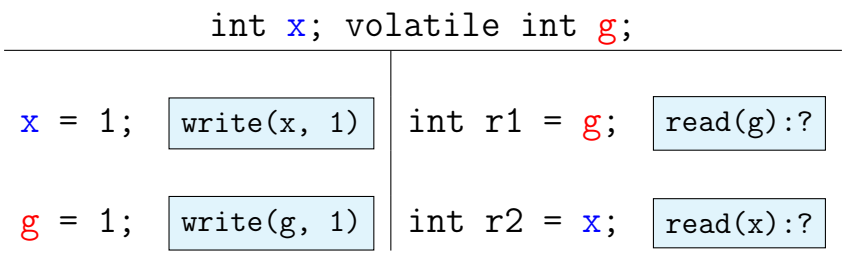
Это единственный линейный порядок, который требует спецификация.

SC-DRF: Проблемы с SO

Сам по себе SO ещё не приводит к слабой модели:

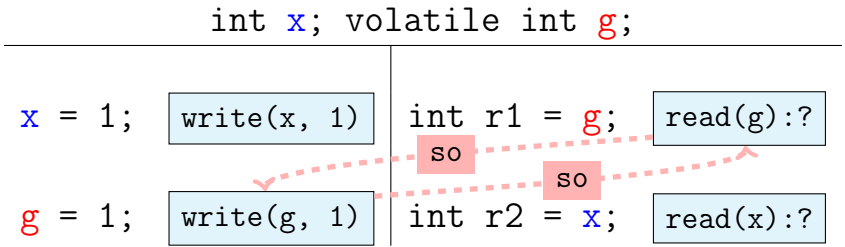
- «Всё или ничего»: или мы превращаем все действия в SA, или миримся с тем, что не-SA действия «плавают» по программе, сея хаос и разрушения
- Все действия в SA превратить нельзя, потому что мы превратим всё в SC, и разрушим оптимизации
- Нужен более слабый порядок для не-SA операций: happens-before

SC-DRF: Прекурсоры HB, публикация



Подумаем: запрещено ли здесь (r1, r2) = (1, 0)?

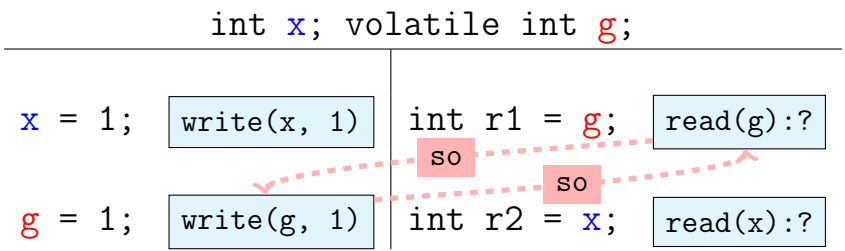
SC-DRF: Прекурсоры HB, публикация



so
$\longrightarrow$ упорядочивает только действия над g!

Получаем либо read(g):0, либо read(g):1

SC-DRF: Прекурсоры HB, публикация



Есть валидные исполнения
и с `read(x):0`, и с `read(x):1`,
вне зависимости от `read(g):?`

SC-DRF: Synchronizes-With Order (SW)

- PO не связывает действия разных потоков
- Чтобы рассуждать о межпотоковом поведении, нам нужен порядок, который связывает действия разных потоков
- Пока что у нас есть SO, но он линеен, и приносит слишком строгие ограничения. Поэтому нам нужен отдельный частичный порядок:

Synchronizes-With Order (SW):
Подпорядок SO, ограниченный
конкретными reads/writes, locks/unlocks, etc.

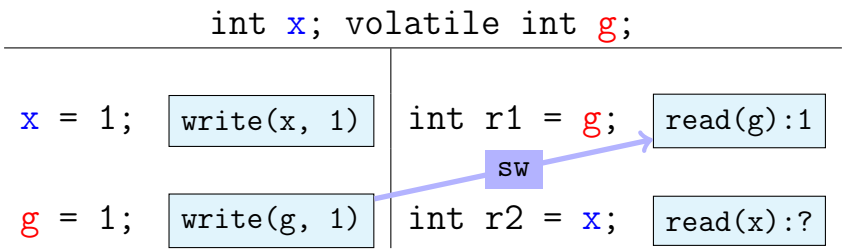
SC-DRF: Synchronizes-With (SW)

`int x; volatile int g;`

`x = 1;`	`write(x, 1)`	`int r1 = g;`	`read(g):0`
`g = 1;`	`write(g, 1)`	`int r2 = x;`	`read(x):?`

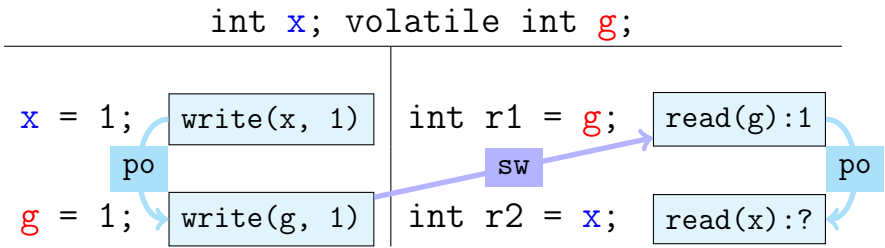
Большинство SA не связано друг с другом $\xrightarrow{\text{sw}}$

SC-DRF: Synchronizes-With (SW)



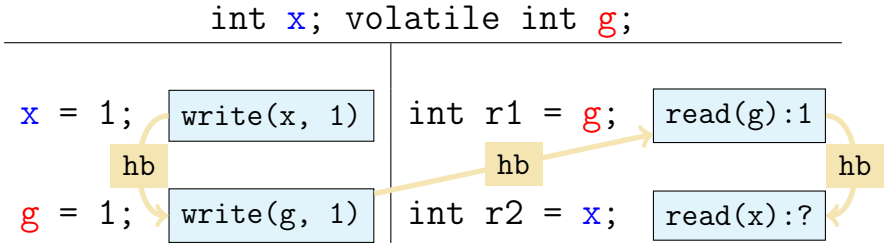
Если одно SA видит другое, то они связаны $\xrightarrow{sw}$.
Это даёт связь «между потоками».

SC-DRF: Synchronizes-With (SW)



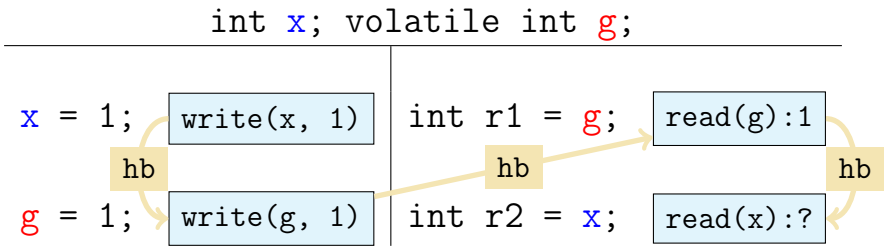
Достроим для этого случая ещё и $\xrightarrow{\text{po}}$.
Это даёт связь «внутри потока».

SC-DRF: Happens-before (HB)



$\xrightarrow{hb}$ = транзитивное замыкание объединения $\xrightarrow{po}$ и $\xrightarrow{sw}$

SC-DRF: Happens-before (HB)



HB consistency: чтения видят либо последнюю запись в $\xrightarrow{hb}$, либо что-нибудь ещё через гонку

SC-DRF: Happens-before (HB)

Пусть $W(r)$ – запись, которую видит r, и A – множество всех действий программы. Тогда happens-before consistency:

$$\forall r \in Reads(A) : \neg(r \xrightarrow{\text{hb}} W(r)) \wedge$$
$$\neg(\exists w \in Writes(A) : (W(r) \xrightarrow{\text{hb}} w) \wedge (w \xrightarrow{\text{hb}} r))$$

SC-DRF: Happens-before (HB)

Пусть $W(r)$ – запись, которую видит r, и A – множество всех действий программы. Тогда happens-before consistency:

$$\forall r \in Reads(A) : \neg(r \xrightarrow{hb} W(r)) \wedge$$

$$\neg(\exists w \in Writes(A) : (W(r) \xrightarrow{hb} w) \wedge (w \xrightarrow{hb} r))$$

Либо $W(r)$ не связано с r (гонка), либо $W(r) \xrightarrow{hb} r$

SC-DRF: Happens-before (HB)

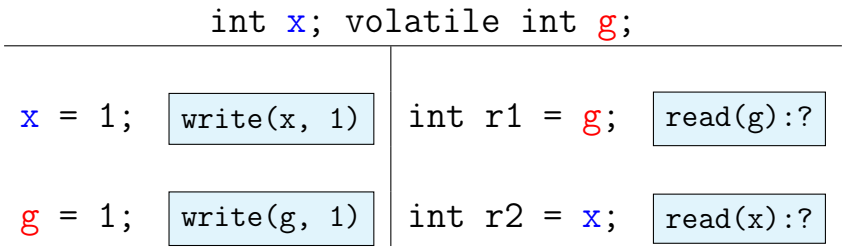
Пусть $W(r)$ – запись, которую видит r, и A – множество всех действий программы. Тогда happens-before consistency:

$$\forall r \in Reads(A) : \neg(r \xrightarrow{\text{hb}} W(r)) \wedge$$

$$\neg(\exists w \in Writes(A) : (W(r) \xrightarrow{\text{hb}} w) \wedge (w \xrightarrow{\text{hb}} r))$$

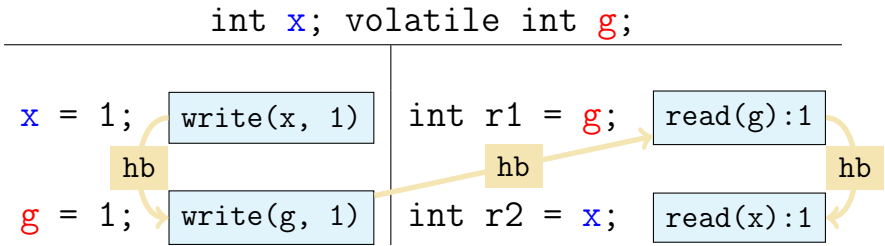
Отсутствуют intervening writes (работает при $W(r) \xrightarrow{\text{hb}} r$)

SC-DRF: HB, Publish



Проанализируем с точки зрения HB consistency...

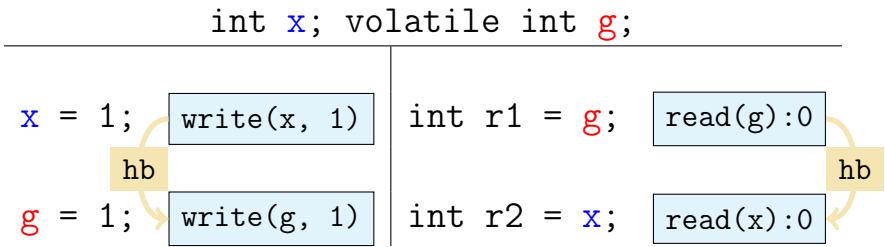
SC-DRF: HB, Publish



Вариант 1: HB consistent, видим последнюю запись в $\xrightarrow{hb}$

$$(r1, r2) = (1, 1)$$

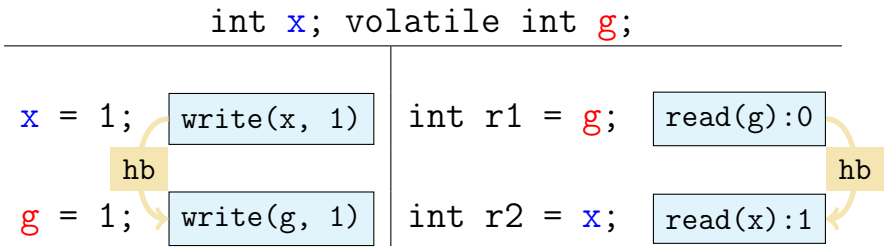
SC-DRF: HB, Publish



Вариант 2: HB consistent, видим значение по-умолчанию

$$(r1, r2) = (0, 0)$$

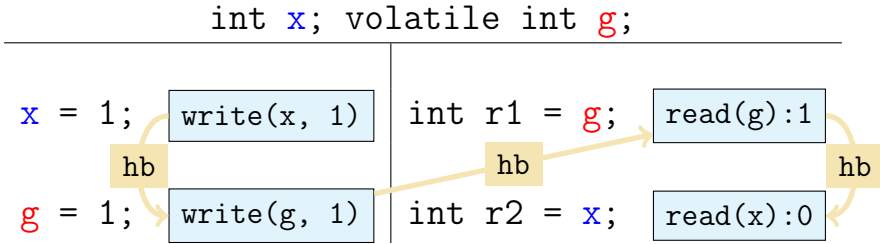
SC-DRF: HB, Publish



Вариант 3: HB consistent (!), чтение через гонку!

$$(r1, r2) = (0, 1)$$

SC-DRF: HB, Publish



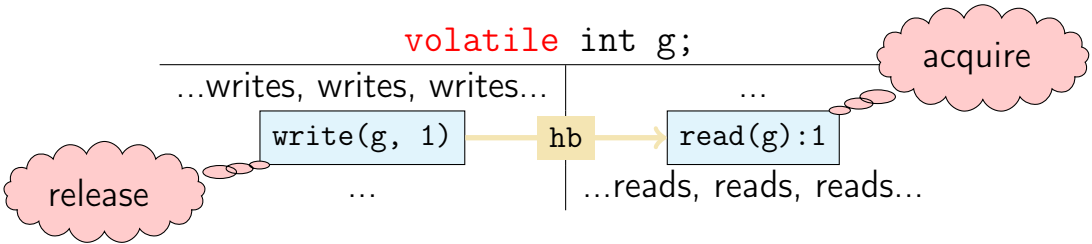
Вариант 4: HB **in**consistent, результат отбросить

SC-DRF: Определение

SequentialConsistency-DataRaceFree: «Correctly synchronized programs have sequentially consistent semantics»

- Перевод: В программе нет гонок $\Rightarrow$ все чтения видят упорядоченные записи $\Rightarrow$ результат исполнения программы можно объяснить каким-нибудь SC-исполнением
- Интуиция №1: Операции над локальными данными (как правило) не ломают SC
- Интуиция №2: Операции над глобальными данными синхронизованы SC-примитивами (доступными прямо в железе)

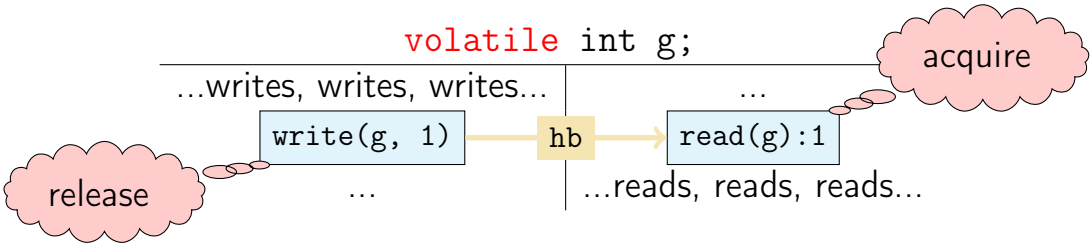
SC-DRF: Публикация



Предыдущий пример можно обобщить в «safe publication»:

- Работает только на одной и той же переменной, одном и том же мониторе

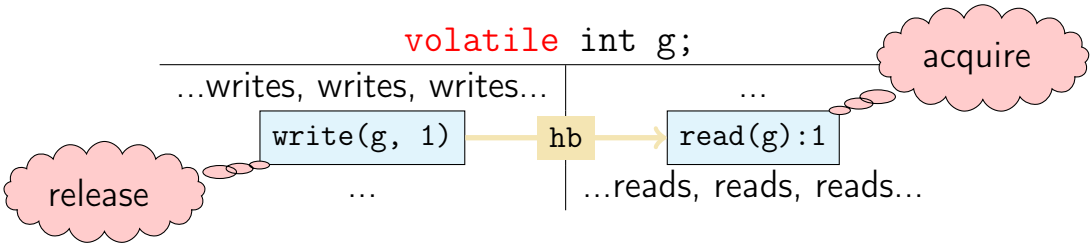
SC-DRF: Публикация



Предыдущий пример можно обобщить в «safe publication»:

- Работает только на одной и той же переменной, одном и том же мониторе
- Работает только если мы *увидели* ту самую `release`-запись

SC-DRF: Публикация



Предыдущий пример можно обобщить в «safe publication»:

- 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.

SC-DRF: Quiz

CR 9234251: Optimize getter in C.get()

```
class C<T> {
  T val;
  public synchronized void set(T v) {
    if (val == null) { val = v; }
  }
  public synchronized T get() {
    // TODO FIXME PLEASE PLEASE PLEASE:
    // THIS ONE IS TOO HOT IN PROFILER!!!111ONEONEONE
    return val;
  }
}
```


SC-DRF: Quiz

RFR (XS) CR 9234251: Optimize getter in C.get():

```
class C<T> {
  T val;
  public synchronized void set(T v) {
    if (val == null) { val = v; }
  }
  public T get() {
    // This one is safe without the synchronization.
    //   (Yours truly, CERTIFIED SENIOR JAVA DEVELOPER)
    return val;
  }
}
```


SC-DRF: Quiz

RFR (XS) CR 9234251: Optimize getter in C.get():

```
class C<T> {
  static volatile int BARRIER; int sink;
  T val;
  public synchronized void set(T v) {
    if (val == null) { val = v; }
  }
  public T get() {
    sink = BARRIER; // acquire membar
    // Obviously, we need a memory barrier here!
    //   (Yours truly, SUPER COMPILER GURU)
    return val;
  }
}
```


SC-DRF: Quiz

Так уже лучше: вернули SW-ребро, восстановили HB.

```
class C<T> {
  volatile T val;
  public synchronized void set(T v) {
    if (val == null) { val = v; }
  }
  public T get() {
    // This one is safe without the synchronization.
    // <Sigh>. Now it's safe.
    //  ($PROJECT techlead, overseeing certified idiots)
    return val;
  }
}
```

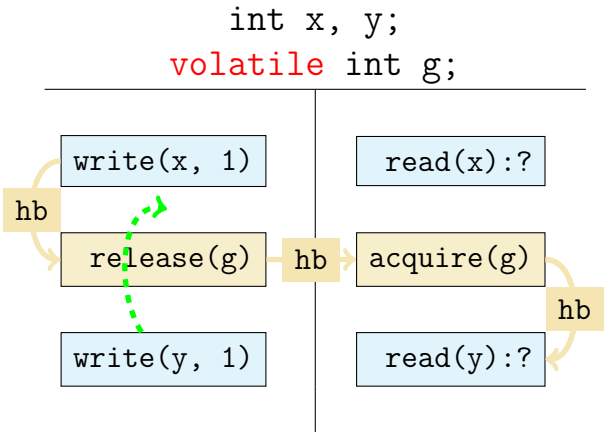

ПЕРЕРЫВ

10 минут на глотнуть воздуха свободы
(или перебежать на другой доклад)

SC-DRF: Roach Motel

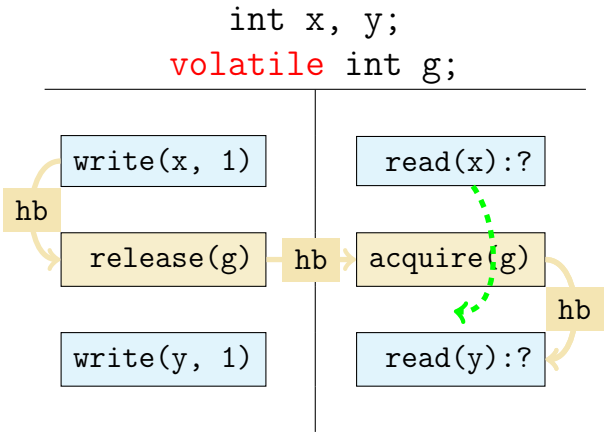
Требования HB consistency разрешают простой класс локальных оптимизаций, **«roach motel»**

SC-DRF: Roach Motel



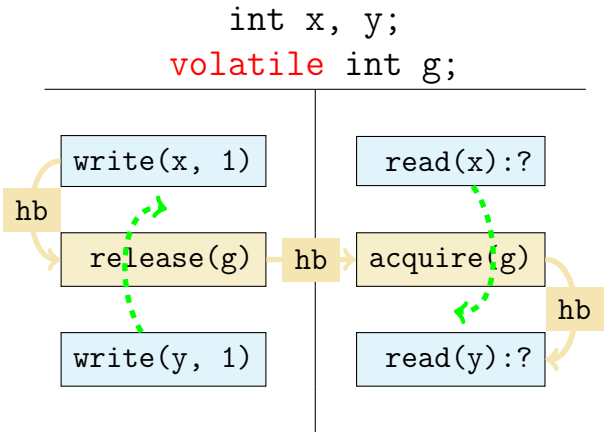
`write(y, 1)` можно переставить перед `release`, потому что он не мешает никаким зависимостям по x, а `read(y):?` справа всё равно может видеть эту запись через гонку

SC-DRF: Roach Motel



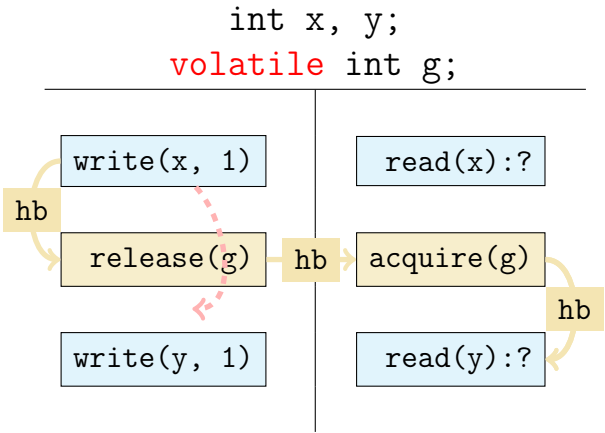
`read(x):?` можно переставить за `acquire`, потому что он всё равно может видеть `write(x, 1)` через гонку

SC-DRF: Roach Motel



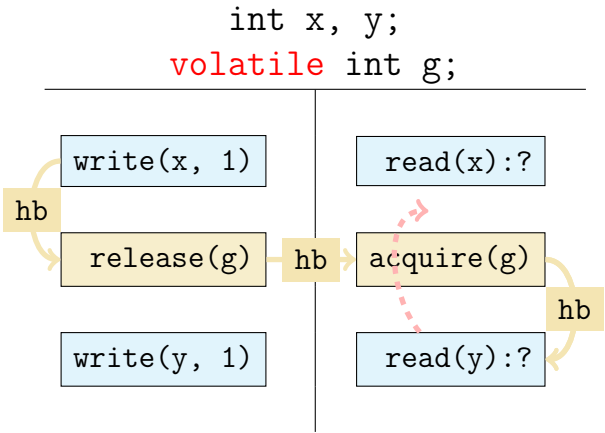
В целом, «Вносибельно после `acquire`» + «Вносибельно перед `release`» = «Вносибельно в `acquire`+`release` блоки» $\Rightarrow$ работает, к примеру, lock coarsening.

SC-DRF: Roach Motel



`write(x, 1)` нельзя просто переставить за `release`, потому что мы потенциально выносим его из $\xrightarrow{hb}$. Консервативная реализация не знает, есть ли дальше чтение x, которое должно его увидеть. ГМО мог бы глобальным анализом это узнать и таки переставить.

SC-DRF: Roach Motel



`read(y):?` нельзя просто переставить перед `acquire`, потому что мы потенциально выносим его из $\xrightarrow{hb}$. Консервативная реализация не знает, есть ли вверх по течению запись, которую мы должны увидеть. ГМО мог бы это узнать и таки переставить.

SC-DRF: Quiz

Что напечатает? Варианты: 0, 41, 42, 43, <ничего>

```
int a = 0;
volatile boolean ready = false;
```

`a = 41;`	`while(!ready) {};`
`a = 42;`	`println(a);`
`ready = true;`	
`a = 43;`	

SC-DRF: Quiz

Что напечатает? Варианты: 0, 41, 42, 43, <ничего>

```
                int a = 0;
volatile boolean ready = false;
```

`a = 41;`	`while(!ready) {};`
`a = 42;`	`println(a);`
`ready = true;`	
`a = 43;`	

Напечатает или 42 (последняя в HB), или 43 (гонка).

SC-DRF: Quiz #2

Что напечатает? Варианты: 0, 41, 42, 43, <ничего>

```
int a = 0;
boolean ready = false;
```

`a = 41;`	`while(!ready) {};`
`a = 42;`	`println(a);`
`ready = true;`	
`a = 43;`	

SC-DRF: Quiz #2

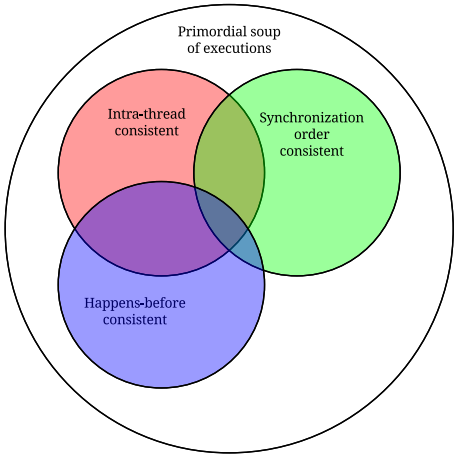
Что напечатает? Варианты: 0, 41, 42, 43, <ничего>

```
int a = 0;
boolean ready = false;
```

`a = 41;`	`while(!ready) {};`
`a = 42;`	`println(a);`
`ready = true;`	
`a = 43;`	

Все варианты возможны (гонка, гонка, гонка!)

SC-DRF: HB constraints



Happens-before consistency частично упорядочивает обычные чтения/записи.

HB определён только для конкретных пар чтений-записей.

SC-DRF: Немножко бенчмарков

`https://github.com/shipilev/jmm-benchmarks/`

- 2x12x2 Xeon E5-2697, 2.70GHz
- OEL 6, JDK 7u40, x86_64
- Измеряем не производительность спеки, а производительность некоторой её реализации

SC-DRF: Немножко бенчмарков

`https://github.com/shipilev/jmm-benchmarks/`

- 2x12x2 Xeon E5-2697, 2.70GHz
- OEL 6, JDK 7u40, x86_64
- Измеряем не производительность спеки, а производительность некоторой её реализации

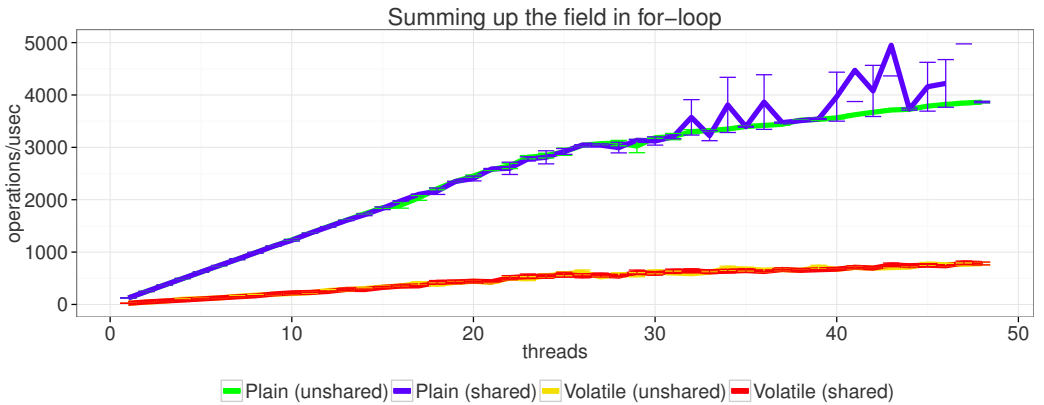
SC-DRF: Hoisting

```
@State(Scope.(Benchmark|Thread))
public static class Storage {
    private (volatile) int v = 42;
}

@GenerateMicroBenchmark
public int test(Storage s) {
    int sum = 0;
    for (int c = 0; c < s.v; c++) {
        sum += s.v;
    }
    return sum;
}
```


SC-DRF: Hoisting

Не так страшен `volatile`, сколько поломанные оптимизации:



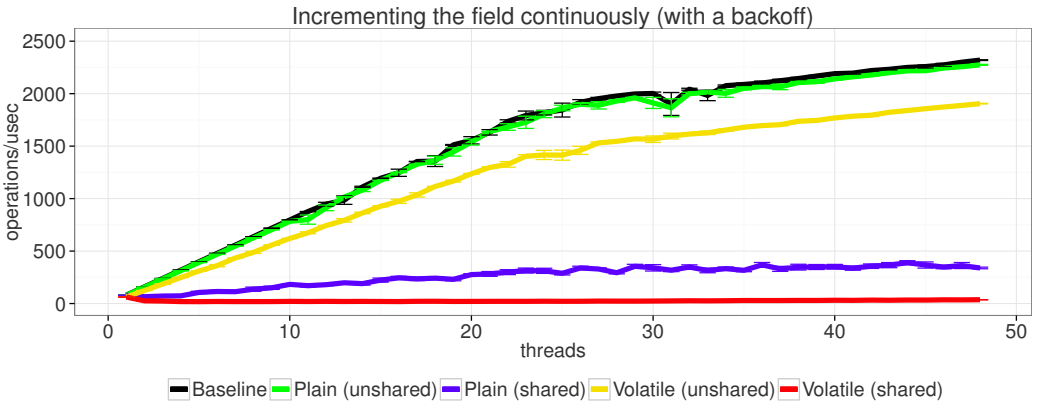
SC-DRF: Writes

```
@State(Scope.(Benchmark|Thread))
public static class Storage {
    private (volatile) int v = 42;
}

@GenerateMicroBenchmark
public int test(Storage s) {
    BlackHole.consumeCPU(8); // ~15ns
    return s.v++;
}
```


SC-DRF: Writes

Не так страшен `volatile`, сколько data sharing:



SC-DRF: JMM 9

- SC-DRF нынче признаётся наиболее удачной моделью
  - Формально идея существует ещё с 90-х годов
  - Адаптировано в Java в 2004
  - ...и теперь ещё в C/C++ в 2011
- В некоторых местах за SC приходится очень больно платить
  - Пример: PowerPC + IRIW = кровь, кишки, расчленёнка
  - Пример: Linux Kernel RCU = релаксации SC для ARM/PowerPC местами дают конские приросты производительности ...и даже без кажущегося взрыва мозга
- Вопрос: можно ли как-нибудь ослабить это требование, не разрушив всю модель?

OoTA

OoTA: Сказка

«SC-DRF. Построй свою любовь»

- Локальные трансформации разрешены, пока не встретим синхронизацию
- Отношения локальных трансформаций и синхронизаций тоже определены (в самом простом случае, «roach motel»)
- Если локальные трансформации переколбашивают конфликтные доступы, значит, там и так была гонка, и девелопер ССЗБ

OoTA: Реальность

Но есть случаи, когда локальные трансформации ломают SC:

```
int a = 0, b = 0;
```

`r1 = a;`	`r2 = b;`
`if (r1 != 0)`	`if (r2 != 0)`
`  b = 42;`	`  a = 42;`

Корректно синхронизована:
все SC исполнения не содержат гонок.
Возможно только `(r1, r2) = (0, 0)`.

OoTA: Оптимизации

Немножко спекулятивных оптимизаций:
почему бы не записать в b, и откатить, «если что»?
(Покажем на компиляторах, хотя может и хардвар спекулировать)

```
int a = 0, b = 0;
```

```
int r1 = a;
if (r1 != 0)
   b = 42;
```

OoTA: Оптимизации

Немножко спекулятивных оптимизаций:
почему бы не записать в b, и откатить, «если что»?
(Покажем на компиляторах, хотя может и хардвар спекулировать)

```
int a = 0, b = 0;
```

```
int r1 = a;
if (r1 != 0)
   b = 42;
```

→

```
int r1 = a;
b = 42;
if (r1 == 0)
  b = 0;
```


OoTA: Оптимизации

Немножко спекулятивных оптимизаций:
почему бы не записать в b, и откатить, «если что»?
(Покажем на компиляторах, хотя может и хардвар спекулировать)

```
int a = 0, b = 0;
```

```
int r1 = a;
if (r1 != 0)
   b = 42;
```

→

```
int r1 = a;
b = 42;
if (r1 == 0)
  b = 0;
```

→

```
b = 42;
int r1 = a;
if (r1 == 0)
  b = 0;
```


OoTA: Как ныне сбирается вещий Эдип...

```
int a = 0, b = 0;
```

`b = 42;`	
	`r2 = b;`
	`if (r2 != 0)`
	`    a = 42;`
`r1 = a;`	
`if (r1 == 0)`	
`    b = 0;`	

- Приводит к `(r1, r2) = (42, 42)`
- В присутствии гонок спекуляция превращается в самоподтверждающееся пророчество!

OoTA: Causality loops

JLS вкратце: OoTA значения запрещены.

- Если мы прочитали значение, то это значит, что кто-то его **до нас** записал
- Самая сложная часть спецификации: весь формализм в JLS 17.4.8 построен для того, чтобы дать определение этому «до нас» = «causality requirements»
- JMM определяет специальный процесс валидации исполнений через «commit»-ы действий

OoTA: Commit semantics

17.4.8 Executions and Causality Requirements

We use $f|_d$ to denote the function given by restricting the domain of f to d. For all x in d, $f|_d(x) = f(x)$, and for all x not in d, $f|_d(x)$ is undefined.

We use $p|_d$ to represent the restriction of the partial order p to the elements in d. For all x,y in d, $p(x,y)$ if and only if $p|_d(x,y)$. If either x or y are not in d, then it is not the case that $p|_d(x,y)$.

A well-formed execution $E = < P, A, po, so, W, V, sw, hb >$ is validated by *committing* actions from A. If all of the actions in A can be committed, then the execution satisfies the causality requirements of the Java programming language memory model.

Starting with the empty set as C_0, we perform a sequence of steps where we take actions from the set of actions A and add them to a set of committed actions C_i to get a new set of committed actions C_{i+1}. To demonstrate that this is reasonable, for each C_i we need to demonstrate an execution E containing C_i that meets certain conditions.

Formally, an execution *E satisfies the causality requirements of the Java programming language memory model* if and only if there exist:

- Sets of actions C_0, C_1, ... such that:
  - C_0 is the empty set
  - C_i is a proper subset of C_{i+1}
  - $A = \cup (C_0, C_1, ...)$

  If A is finite, then the sequence C_0, C_1, ... will be finite, ending in a set $C_n = A$.

  If A is infinite, then the sequence C_0, C_1, ... may be infinite, and it must be the case that the union of all elements of this infinite sequence is equal to A.
- Well-formed executions E_1, ..., where $E_i = < P, A_i, po_i, so_i, W_i, V_i, sw_i, hb_i >$.

Given these sets of actions C_0, ... and executions E_1, ... , every action in C_i must be one of the actions in E_i. All actions in C_i must share the same relative happens-before order and synchronization order in both E_i and E. Formally:

1. C_i is a subset of A_i
2. $hb_i|_{C_i} = hb|_{C_i}$
3. $so_i|_{C_i} = so|_{C_i}$

The values written by the writes in C_i must be the same in both E_i and E. Only the reads in C_{i-1} need to see the same writes in E_i as in E. Formally:

4. $V_i|_{C_i} = V|_{C_i}$
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$

All reads in E_i that are not in C_{i-1} must see writes that happen-before them. Each read r in C_i - C_{i-1} must see writes in C_{i-1} in both E_i and E, but may see a different write in E_i from the one it sees in E. Formally:

6. For any read r in A_i - C_{i-1}, we have $hb_i(W_i(r), r)$
7. For any read r in (C_i - C_{i-1}), we have $W_i(r)$ in C_{i-1} and $W(r)$ in C_{i-1}

Given a set of sufficient synchronizes-with edges for E_i, if there is a release-acquire pair that happens-before (§17.4.5) an action you are committing, then that pair must be present in all E_j, where $j \geq i$. Formally:

8. Let ssw_i be the sw_i edges that are also in the transitive reduction of hb_i but not in po. We call ssw_i the *sufficient synchronizes-with edges for* E_i. If $ssw_i(x, y)$ and $hb_i(y, z)$ and z in C_i, then $sw_j(x, y)$ for all $j \geq i$.

   If an action y is committed, all external actions that happen-before y are also committed.
9. If y is in C_i, x is an external action and $hb_i(x, y)$, then x in C_i.

OoTA: Commit semantics

17.4.8 Executions and Causality Requirements

We use $f|_d$ to denote the function given by restricting the domain of f to d. For all x in d, $f|_d(x) = f(x)$, and for all x not in d, $f|_d(x)$ is undefined.

We use $p|_d$ to represent the restriction of the partial order p to the elements in d. For all x,y in d, $p(x,y)$ if and only if $p|_d(x,y)$. If either x or y are not in d, then it is not the case that $p|_d(x,y)$.

A well-formed execution $E = < P, A, po, so, W, V, sw, hb >$ is validated by *committing* actions from A. If all of the actions in A can be committed, then the execution satisfies the causality requirements of the Java programming language memory model.

Starting with the empty set as C_0, we perform a sequence of steps where we take actions from the set of actions A and add them to a set of committed actions C_i to get a new set of committed actions C_{i+1}. To demonstrate that this is reasonable, for each C_i we need to demonstrate an execution E containing C_i that meets certain conditions.

Formally, an execution E *satisfies* [illegible] *ents of the Java programming language memory mode*

- Sets of actions C_0, C_1, ... such tha
  - C_0 is the empty set
  - C_i is a proper subset of C_{i+1}
  - $A = \cup (C_0, C_1, ...)$

  If A is finite, then the seq [illegible] et $C_n = A$.

  If A is infi [illegible] ust be the [illegible] ual to A.

  [illegible] V_i, sw_i, hb_i >.

Given these sets of actions C_0, ... and executions E_1, ... , every action in C_i must be one of the actions in E_i. All actions in C_i must share the same relative happens-before order and synchronization order in both E_i and E. Formally:

1. C_i is a subset of A_i
2. $hb_i|_{C_i} = hb|_{C_i}$
3. $so_i|_{C_i} = so|_{C_i}$

The values written by the writes in C_i must be the same in both E_i and E. Only the reads in C_{i-1} need to see the same writes in E_i as in E. Formally:

4. $V_i|_{C_i} = V|_{C_i}$
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$

All reads in E_i that are not in C_{i-1} must see writes that happen-before them. Each read r in C_i - C_{i-1} must see writes in C_{i-1} in both E_i and E, but may see a different write in E_i from the one it sees in E. Formally:

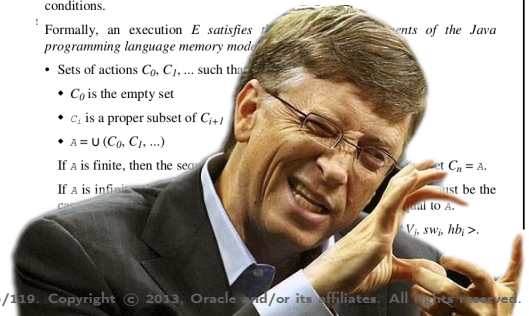
6. For any read r in A_i - C_{i-1}, we have $hb_i(W_i(r), r)$
7. For any read r in (C_i - C_{i-1}), we have $W_i(r)$ in C_{i-1} and $W(r)$ in C_{i-1}

Given a set of sufficient synchronizes-with edges for E_i, if there is a release-acquire pair that happens-before (§17.4.5) an action you are committing, then that pair must be present in all E_j, where $j \geq i$. Formally:

8. Let ssw_i be the sw_i edges that are also in the transitive reduction of hb_i but not in po. We call ssw_i the *sufficient synchronizes-with edges for* E_i. If $ssw_i(x, y)$ and $hb_i(y, z)$ and z in C_i, then $sw_j(x, y)$ for all $j \geq i$.

   If an action y is committed, all external actions that happen-before y are also committed.

9. If y is in C_i, x is an external action and $hb_i(x, y)$, then x in C_i



Java ORACLE

OoTA: C/C++11

Спецификация OoTA настолько сложна, что C/C++11 сдался:
C/C++11 не специфицировала эту часть своей модели

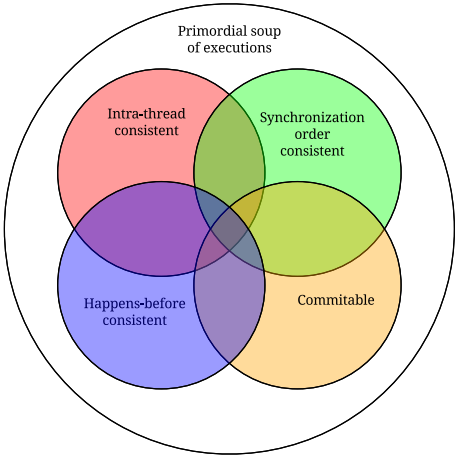
- Избежала всего геморроя со спецификацией (Пиррова победа)
- Особенно в присутствии relaxed atomics, которые нам нужны, потому что мы же любим низкоуровневые оптимизации, да?
- Эксперты в C/C++1x WG чешут темечко, как (в теории) запретить компиляторам спекулятивно рождать значения

OoTA: JMM 9

Пути решения проблемы OoTA:

1. Продолжать в том же духе: предпринять попытку до/перепилить формализм, чтобы он был понимабелен/верифицируем и исправить в нём ошибки
2. Консервативно запретить спекулятивные записи: это будет означать `LoadStore` перед каждой записью
3. Сдаться, и надеяться на вменяемость реализаций

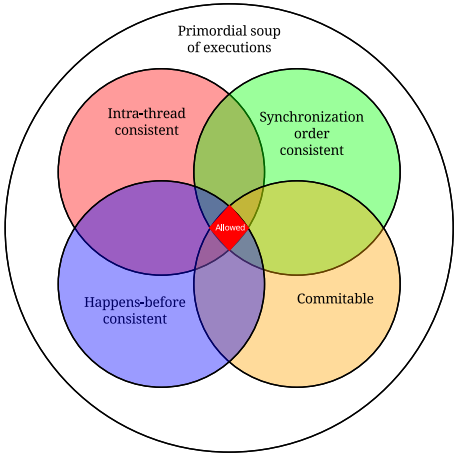
OoTA: Commit semantics



Commit semantics делает финальные проверки, чтобы исключить причинностные циклы.

Это нужно, чтобы запретить Out of Thin Air значения.

OoTA: ...Land of Mordor where the Shadows lie



Исполнения, которые прошли все проверки модели, теперь могут использоваться для получения возможных результатов программы.

Если исполнение не проходит хотя бы через один фильтр, то мы можем его сразу отбросить.

Finals

Finals: Quiz

Что напечатает?

```
class A {
    int f;
    A() { f = 42; }
}

A a;
```

`a = new A();`	`if (a != null)` `  println(a.f);`

Finals: Quiz

Что напечатает?

```
class A {
    int f;
    A() { f = 42; }
}

A a;
```

`a = new A();`	`if (a != null)` `  println(a.f);`

<ничего>, 0, 42, или бросит NPE.

Finals: Quiz

«Правильно» не бросит NPE:

```
class A {
   int f;
   A() { f = 42; }
}

A a;
```

`a = new A();`	`A ta = a;`
	`if (ta != null)`
	`  println(ta.f);`

Finals: Сказка

Хотелось бы получить только 42:

```
class A {
   ????? int f;
   A() { f = 42; }
}

A a;
```

`a = new A();`	`A ta = a;`
	`if (ta != null)`
	`  println(ta.f);`

Finals: Сказка

Хочется иметь объекты,
которые можно безопасно публиковать через гонки

- ...чтобы не платить за «лишние» synchronization actions
- ...чтобы не нарушить security, если какой-нибудь (злонамеренный) дурак наш защищённый объект опубликовал через гонку

Finals: Быль

Спекуляция приводит к дефолтам и кризисам.
Но совсем сажать за спекуляцию нельзя,
ибо она приносит профит.

- Способ точечно сломать спекуляцию: никогда не показывать неполные объекты: тогда неоткуда ни рантайму, ни железу, ни чёрту в ступе взять левое значение
- ...естественный способ поддержать в языке – новые объекты, ибо про них ничего не известно
- ...включать магическим словом `final`: если пользователи хотят иного, не ставят `final`

Finals: Решение

В конце конструктора происходит freeze action

Freeze action «замораживает» поля

- Если поток прочитал ссылку на объект, то он увидит замороженные значения
- Если поток прочитал из `final`-поля ссылку на другой объект, то состояние того как минимум настолько же свежее, как и на время freeze'а

Finals: Формально

$$w \xrightarrow{\text{hb}} F \xrightarrow{\text{hb}} a \xrightarrow{\text{mc}} r1 \xrightarrow{\text{dr}} r2,$$

w – запись целевого поля, F – freeze action, a – некое действие (но не чтение final-поля), $r1$ – чтение final-поля, $r2$ – чтение целевого поля

Вводятся два новых частичных порядка:

- *dereference order (dr)* (цепочки доступа внутри потока)
- *memory order (mc)* (цепочки публикации между потоками)

Finals: Формально

$$w \xrightarrow{hb} F \xrightarrow{hb} a \xrightarrow{mc} r1 \xrightarrow{dr} r2,$$

w – запись целевого поля, F – freeze action, a – некое действие (но не чтение final-поля), $r1$ – чтение final-поля, $r2$ – чтение целевого поля

Если единственный путь через $\xrightarrow{dr}$ и $\xrightarrow{mc}$ до записи поля лежит через F, то можем увидеть только замороженное значение, ура!

А если есть другие пути...

Finals: Пример[11]

Thread 1

```
T l = new T() {{
  fx = 42; w
}}; f
GLOBAL = l; a
```

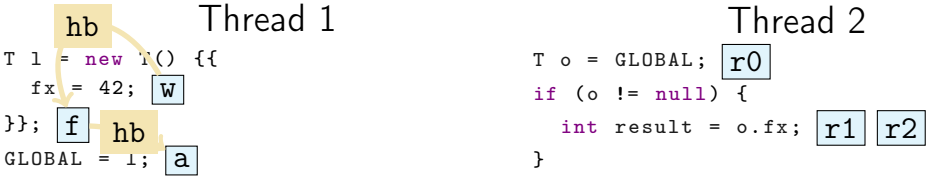
Thread 2

```
T o = GLOBAL; r0
if (o != null) {
  int result = o.fx; r1 r2
}
```

Возможно ли в `result` получить 0?

[11] Цельнотянут у Владимира Ситникова и Валентина Коваленко

Finals: Пример[11]

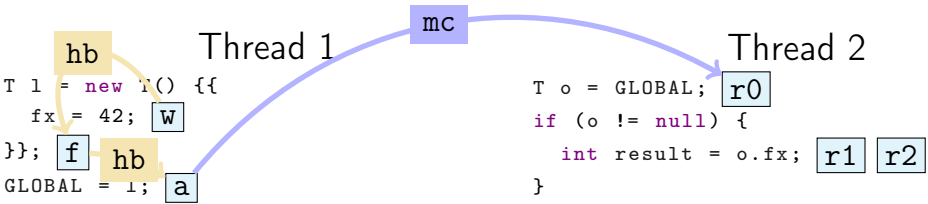


Действия в одном потоке образуют happens-before:

$$w \xrightarrow{\text{hb}} f,\ f \xrightarrow{\text{hb}} a$$

[11] Цельнотянут у Владимира Ситникова и Валентина Коваленко

Finals: Пример[11]

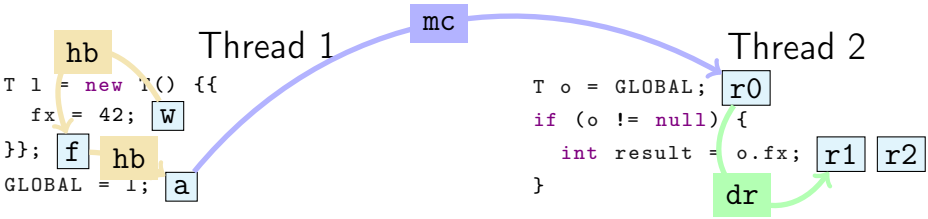


$r0$ видит запись a:

$$a \xrightarrow{\text{mc}} r0$$

[11] Цельнотянут у Владимира Ситникова и Валентина Коваленко

Finals: Пример[11]

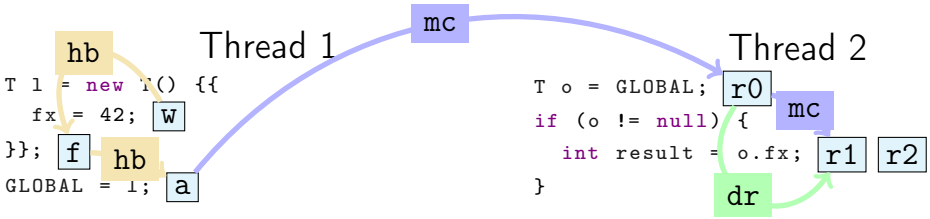


Поток 2 не создавал объект, r1 читает его поле, а r это единственное чтение адреса объекта, поэтому dereference chain:

$$r0 \xrightarrow{dr} r1$$

[11] Цельнотянут у Владимира Ситникова и Валентина Коваленко

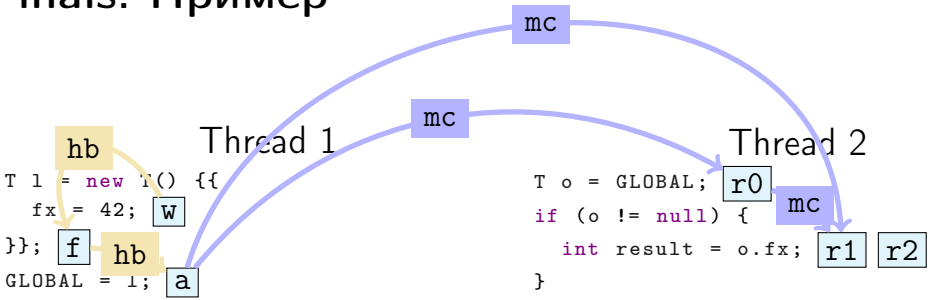
Finals: Пример[11]



$$r0 \xrightarrow{dr} r1 \Rightarrow r0 \xrightarrow{mc} r1$$

[11] Цельнотянут у Владимира Ситникова и Валентина Коваленко

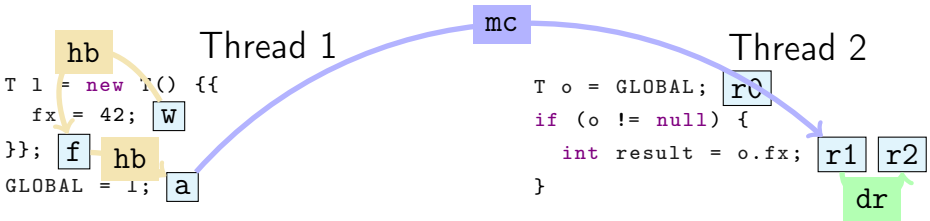
Finals: Пример[11]



$$a \xrightarrow{\text{mc}} r1 \ (\xrightarrow{\text{mc}} \text{транзитивно})$$

[11] Цельнотянут у Владимира Ситникова и Валентина Коваленко

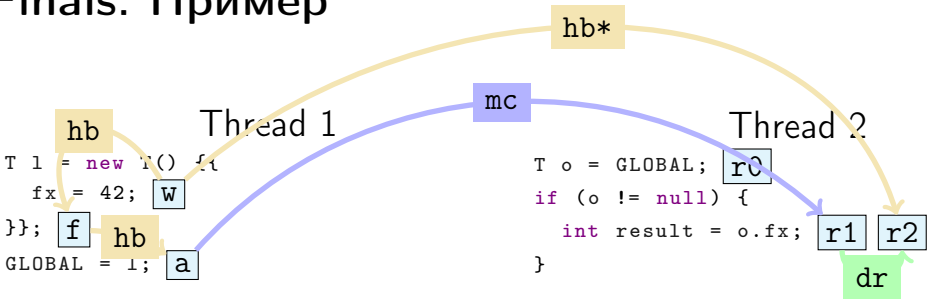
Finals: Пример[11]



Возьмём $r2 = r1$, тогда $r1 \xrightarrow{dr} r2$ ($\xrightarrow{dr}$ рефлексивно)

[11] Цельнотянут у Владимира Ситникова и Валентина Коваленко

Finals: Пример[11]

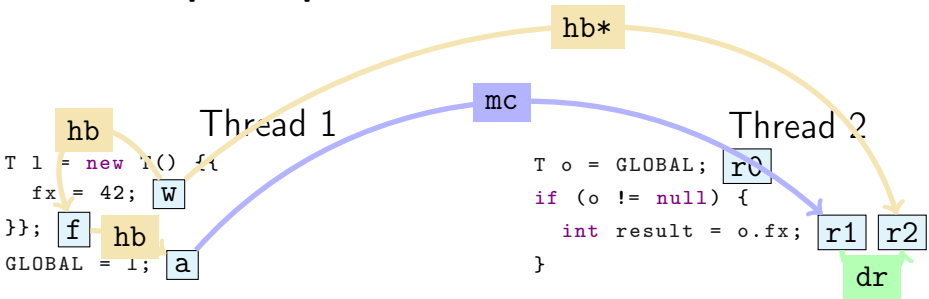


Нашли всё необходимое для HB^*:

$$w \xrightarrow{hb} f \xrightarrow{hb} a \xrightarrow{mc} r1 \xrightarrow{dr} r2 \Rightarrow w \xrightarrow{hb} r2$$

[11] Цельнотянут у Владимира Ситникова и Валентина Коваленко

Finals: Пример[11]



$$(w \xrightarrow{\text{hb}} r2) \Rightarrow \mathsf{r} \in \{42\}$$

[11] Цельнотянут у Владимира Ситникова и Валентина Коваленко

Finals: Засады

Гарантии на замороженность
пропадают при преждевременной публикации:

T p, q;

T t1 = <new>	T t2 = p	T t4 = q
t1.f = 42	r2 = t2.f	r4 = t4.f
p = t1	T t3 = q	
<freeze t1.f>	r3 = t3.f	
q = t1		

$r4 \in \{42\}$;
но r2, r3 $\in \{0, 42\}$, ибо p «утёк»

Finals: Прагматика

Дать оптимизациям мега-свободу по кешированию `final`-ов

- «All references are created equal»: локальный оптимизатор не перегружен анализом ссылок
- Как только оптимизатор увидел опубликованную ссылку, он может скэшировать все его `final` поля, и баста!
- Т.е. если оптимизатор увидел недоконструированный объект, то хана

Finals: Реализация

Довольно просто реализовать на большинстве платформ

- Достаточно запретить переупорядочивание инициализации `final`-полей и публикации объекта на большинстве архитектур
- Все известные промышленные архитектуры[12] не переупорядочивают загрузку объекта, и загрузку поля из него (dependent loads)

[12] кроме Alpha, но она отправилась к праотцам, куда ей и дорога

Finals: Quiz

Что напечатает?

```
class A {
   final int f;
   { f = 42; }
}
A a;
```

`a = new A();`	`A ta = a;`
	`if (ta != null)`
	`  println(ta.f);`

Finals: Quiz

Что напечатает?

```
class A {
   final int f;
   { f = 42; }
}
A a;
```

`a = new A();`	`A ta = a;`
	`if (ta != null)`
	`  println(ta.f);`

Конечно, 42.

Finals: JMM 9

- Текущая спецификация плохо относится к не-`final`-полям
- Если поле записано в конструкторе, и никогда не модифицируется? (e.g. юзер прошляпил `final` на поле)
- Если поле уже `volatile`? (e.g. `AtomicInteger`)
- Если объект строится билдерами? (lazy lists?)

- Вопрос: не стоит ли дать гарантии на инициализацию для **всех** полей и **всех** конструкторов?

Finals: JMM 9

https://github.com/shipilev/jmm-benchmarks/

- 2x12x2 Xeon E5-2697, 2.70GHz; OEL 6, JDK 8b121, x86_64
- 1x4x1 Cortex-A9, 1.7 GHz; Linaro 12.11, JDK 8b121, SE Embedded
- Измеряем не производительность спеки, а производительность некоторой её реализации

Finals: JMM 9

https://github.com/shipilev/jmm-benchmarks/

- 2x12x2 Xeon E5-2697, 2.70GHz; OEL 6, JDK 8b121, x86_64
- 1x4x1 Cortex-A9, 1.7 GHz; Linaro 12.11, JDK 8b121, SE Embedded
- Измеряем не производительность спеки, а производительность некоторой её реализации

Java ORACLE

Finals: JMM 9: Initialization (chained)

```
@GenerateMicroBenchmark
public Object test() {
   return new Test_[N](v);
}

// chained case
class Test_[N] extends Test_[N-1] {
   private [plain|final] int i_[N];
   public <init>(int v) {
      super(v);
      i_[N] = v;
   }
}
```

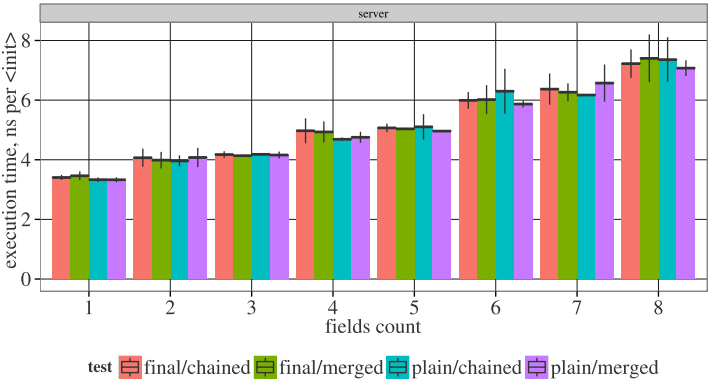

Finals: JMM 9: Initialization (merged)

```
@GenerateMicroBenchmark
public Object test() {
   return new Test_[N](v);
}

// merged case
class Test_[N] {
   private [plain|final] int i_1, ..., i_[N];
   public <init>(int v) {
      i_1 = i_2 = ... = i_[N] = v;
   }
}
```


Finals: JMM 9: Результаты (x86)

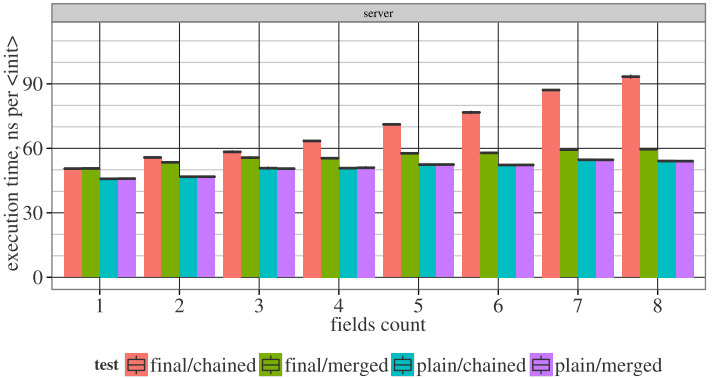
На Total Store Order это вообще бесплатно:[13]



[13] http://shipilev.net/blog/2014/all-fields-are-final/

Finals: JMM 9: Результаты (ARMv7)

На weakly-ordered архитектурах нужно клеить барьеры:[14]



[14] http://shipilev.net/blog/2014/all-fields-are-final/

Заключение

Заключение: Lingua Latina...

«The best way is to build up a small repertoire of constructions that you know the answers for and then never think about the JMM rules again unless you are forced to do so! Literally nobody likes figuring things out from the JMM rules as stated, or can even routinely do so correctly. This is one of the many reasons we need to overhaul JMM someday.»

Заключение: Lingua Latina...

«The best way is to build up a small repertoire of constructions that you know the answers for and then never think about the JMM rules again unless you are forced to do so! Literally nobody likes figuring things out from the JMM rules as stated, or can even routinely do so correctly. This is one of the many reasons we need to overhaul JMM someday.»

(Doug Lea, private communication, 2013)

Заключение: Известные проблемы

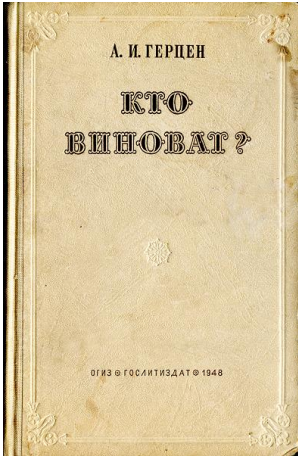
- JSR 133 Cookbook не содержит некоторых машинно-специфичных особенностей, обнаруженных много позднее создания текущей JMM
- Некоторые новые примитивы в библиотеке вообще не специфицируемы в текущей модели (e.g. `lazySet`, `weakCompareAndSet`)
- JMM специфицирована для Java, что делать с JVM-based языками?
- В формальной спецификации JMM есть формальные неточности, которые ставят раком автоматические верификаторы

Заключение: JMM Overhaul

«Java Memory Model update»
`http://openjdk.java.net/jeps/188`

- Improved formalization
- JVM languages coverage
- Extended scope for existing unspec-ed primitives
- C11/C++11 compatibility
- Testing support
- Tool support

Заключение: Чтение



Заключение: Чтение

- Goetz et al, «Java Concurrency in Practice»
- Herilhy, Shavit, «The Art of Multiprocessor Programming»
- Adve, «Shared Memory Models: A Tutorial»
- McKenney,«Is Parallel Programming Hard, And, If So, What Can You Do About It?»
- Manson, «Java Memory Model» (Special PoPL issue)
- Huisman, Petri, «JMM: The Formal Explanation»

Q/A

Backup

Backup: Actions

Action: $A =< t, k, v, u >$

- t – the thread performing the action
- k – the kind of action
- v – the variable or monitor involved in the action
- u – an arbitrary unique identifier for the action

Backup: Executions

Execution: $E =< P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} >$

- P – program; A – set of program actions
- $\xrightarrow{po}$ – program order;
- $\xrightarrow{so}$ – synchronization order
- $\xrightarrow{sw}$ – synchronizes-with order
- $\xrightarrow{hb}$ – happens-before order
- $W(r)$ – «write seen function», answers what write the read observes; $V(r)$ – answers what value the read observes

