ORACLE


# Java Concurrency:
## Битва за корректность

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

# Дисклеймеры

1. Доклад про тестирование JVM и боль.
   *(Уходите.)*
2. Доклад сложный, быстрый, беспощадный.
   *(Ещё есть шанс уйти.)*
3. Докладчик – мизантроп, сноб, и зануда.
   *(Нет, правда, никто не держит.)*
4. Доклад содержит кровь, кишки и расчленёнку JVM и JDK.
   *(Доклады про пони и бабочек... NOPE!)*

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: зачем?

- Без этого заниматься производительностью толку нет
- Concurrent-баги проявляются внезапно
- Concurrent-баги трудно диагностировать и воспроизводить
- Производительность некорректной программы = ?

# Введение: что ломается

Тысяча и одно место:

- приложение неправильно использует JMM/библиотеку
- библиотека неправильно использует JMM/JVM
- JVM неправильно использует JMM/HW
- HW неправильно следует собственной спеке
- ...или любая комбинация из этих ошибок

# Введение: весь доклад в тезисах

1. **везде есть ошибки**
   - не бывает корректных программ и железок
   - (бывают плохо протестированные)
2. **надёжная изоляция от HW – мечта**
   - часто вообще шизофреническая иллюзия
   - огромная задача, посильная огромному community
3. **пуленепробиваемый софт сложен**
   - «я пишу на Clojure, и меня это не касается»
   - диагностировать баги, фиксить, или обходить

# Подходы


# Подходы: prior art

1. Java Compability Kit (JCK)
   - разрабатывается Sun/Oracle, JCP
   - проверяет утверждения из JLS §17
   - ограничена только явной спецификацией
2. JSR166 TCK
   - разрабатывается Doug Lea и компанией
   - проверяет функциональность j.u.c.*
   - только базовые функциональные тесты
3. Litmus/DIY
   - кембриджская лаба 'Weak Consistency Models'[1]
   - проверяет семантику хардвара
   - низкоуровневый, уровня железа

[1] `http://www.cl.cam.ac.uk/~pes20/`

# Подходы: главная проблема тестов

## Отлить в граните

Concurrency-баги – это баги на *гонках*



# Подходы: главная проблема тестов

Требуется обеспечить контролируемую гонку:

- обширную, чтобы потоки встречались
- ограниченную, чтобы можно было доверять результатам
- лишняя синхронизация маскирует эффекты
- скорость теста $\sim$ надёжность теста

К сожалению, все наивные тесты не тестируют вообще ничего!

# Подходы: попытка №1

Вот это полный бред:

```
volatile int v;

void doTest() {
    Thread t1 = new Thread(() -> v++);
    Thread t2 = new Thread(() -> v++);
    t1.start(); t2.start();
    t1.join(); t2.join();

    Assert.assertTrue(2, v);
}
```

«Окошко коллизии» очень маленькое,
и на практике коллизии не будет никогда.

# Подходы: попытка №2

Уже лучше, но всё равно бредово:

```
volatile int v;
final CountDownLatch l = new CDL(2);

void doTest() {
    Thread t1 = new Thread(() -> l.countDown(); l.await(); v++);
    Thread t2 = new Thread(() -> l.countDown(); l.await(); v++);
    t1.start(); t2.start();
    t1.join(); t2.join();

    Assert.assertTrue(2, v);
}
```

Пока потоки распаркуются, поезд уже давно уйдёт.

# Подходы: наш подход

Очень похож на Litmus, но написан на Java:

- большой массив объектов-состояний
- актёры мутируют состояние под гонкой
- актёры сохраняют наблюдения
- актёры никогда не паркуются, а активно ждут
- инфра склеивает наблюдения для состояний
- инфра считает частотность результатов

Как правило, большинство тестов укладывается в эту схему.


ORACLE

# Подходы: наш подход

Даёт возможность абстрагировать тесты:

```
class MyTest implements ConcurrencyTest<State, Res> {
   void actor1(State s, Res r) { r.r1 = s.v++; }
   void actor2(State s, Res r) { r.r2 = s.v++; }

   class State {  volatile int v;  }
   State newState() { new State(); }
}
```

...и считать частотность (r1, r2):

```
          State       Occurrences         Expectation
       [1, 1] (     1,360,407)   KNOWN_ACCEPTABLE
       [1, 2] (    57,137,771)            REQUIRED
       [2, 1] (    55,286,472)            REQUIRED
```

# Подходы: засады

Очень похоже на бенчмаркинг:

- прогрев
- недетерминизм компиляции
- профили
- многопоточность

# Подходы: главная засада

Быстрая и многопоточная инфраструктура тестов подвержена тем самым багам, что ловят сами тесты.



# Подходы: главная засада

Быстрая и многопоточная инфраструктура тестов подвержена тем самым багам, что ловят сами тесты.

## Принцип Шипилёва-Мюнхгаузена

Провал на одном тесте автоматически инвалидирует результаты всех остальных тестов.

# Контрольные тесты



# Контрольные тесты: мотивация

Перед тем, как бросаться на амбразуру, нужен контроль.

Надо проверить, что:

- инфраструктура ловит легальные гонки
- инфраструктура не фейлит валидные тесты
- инфраструктура корректно синхронизована

# Контрольные тесты: синглетоны

Можно описать такой классик:

```
class MyTest implements ConcurrencyTest<State, Res> {
    int get(SingletonFactory f) {
        Singleton s = f.getInstance();
        if (s == null) {
            return 0;
        if (s.x == null)
            return 1;
        return s.x;
    }

    void actor1(SingletonFactory f, Res r) {  r.r1 = get(f); }
    void actor2(SingletonFactory f, Res r) {          get(f); }
}
```

Тогда по возвращаемому значению можно будет сказать, как там у нас дела.


ORACLE

# Контрольные тесты: синглетоны

```
public static class UnsafeSingletonFactory implements SingletonFac
    private Singleton instance; // intentionally non-volatile

    public Singleton getInstance() {
        if (instance == null) {
            synchronized (this) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}

public static class Singleton {
    public Byte x;
    public Singleton() { x = 42; }
}
```



# Контрольные тесты: синглетоны

```
public static class UnsafeSingletonFactory implements SingletonFac
    private Singleton instance; // intentionally non-volatile

    public Singleton getInstance() {
        if (instance == null) {
            synchronized (this) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}

public static class Singleton {
    public Byte x;
    public Singleton() { x = 42; }
}
```

A: (instance == null)
B: (instance.x == null)
C: (instance.x == 42)
D: NullPointerException


ORACLE

# Контрольные тесты: примитивы

Всем известная история про long/double:

```
class MyTest implements ConcurrencyTest<State, Res> {
   void actor1(State s, Res r) { s.x = -1L; }
   void actor2(State s, Res r) { r.r1 = s.x; }
   class State { long x; }
}
```

Каковы возможные значения r1?

# Контрольные тесты: примитивы

Всем известная история про long/double:

```
class MyTest implements ConcurrencyTest<State, Res> {
   void actor1(State s, Res r) { s.x = -1L; }
   void actor2(State s, Res r) { r.r1 = s.x; }
   class State { long x; }
}
```

Каковы возможные значения r1?

```
          State      Occurrences          Expectation
            [0] (    78,014,883)            REQUIRED
           [-1] (    88,013,591)            REQUIRED
 [4294967295] (         8,975)    KNOWN_ACCEPTABLE
[-4294967296] (        30,801)    KNOWN_ACCEPTABLE
```

JLS: long/double не атомарны.

# Контрольные тесты: примитивы

## A с volatile?

```
class MyTest implements ConcurrencyTest<State, Res> {
    void actor1(State s, Res r) { s.x = -1L; }
    void actor2(State s, Res r) { r.r1 = s.x; }
    class State { volatile long x; }
}
```

# Контрольные тесты: примитивы

## A с volatile?

```
class MyTest implements ConcurrencyTest<State, Res> {
    void actor1(State s, Res r) { s.x = -1L; }
    void actor2(State s, Res r) { r.r1 = s.x; }
    class State { volatile long x; }
}
```

## Всё нормально, как JLS и предписывает:

```
          State       Occurrences         Expectation
            [0] (   110,284,406)            REQUIRED
           [-1] (    32,400,899)            REQUIRED
```

# Только хардкор: что уже всплыло



# Дело о неправильных метках: #1

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

| | |
|---|---|
| `y = 1;` | `int t = y;` |
| `x = 1;` | `int r1 = x;` |
| | `int r2 = y;` |

Какие (r1, r2) возможны?

# Дело о неправильных метках: #1

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

| | |
|---|---|
| `y = 1;` | `int t = y;` |
| `x = 1;` | `int r1 = x;` |
| | `int r2 = y;` |

Какие (r1, r2) возможны?

- (1, 1) – T1 выполнился полностью раньше T2
- (0, 0) – T2 выполнился полностью раньше T1

# Дело о неправильных метках: #1

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

| | |
|---|---|
| `y = 1;` | `int t = y;` |
| `x = 1;` | `int r1 = x;` |
| | `int r2 = y;` |

Какие (`r1`, `r2`) возможны?

- (1, 1) – T1 выполнился полностью раньше T2
- (0, 0) – T2 выполнился полностью раньше T1
- (0, 1) – легально, T1 успел «`y = 1`»

# Дело о неправильных метках: #1

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

| | |
|---|---|
| `y = 1;` | `int t = y;` |
| `x = 1;` | `int r1 = x;` |
| | `int r2 = y;` |

Какие (`r1`, `r2`) возможны?

- (1, 1) – T1 выполнился полностью раньше T2
- (0, 0) – T2 выполнился полностью раньше T1
- (0, 1) – легально, T1 успел «`y = 1`»
- (1, 0) – нарушение JMM


ORACLE

# Дело о неправильных метках: #2

```
class MyTest implements ConcurrencyTest<State, Res> {
    void actor1(State s, Res r) {
        s.y = 1;
        s.x = 1;
    }
    void actor2(State s, Res r) {
        int t = s.y;
        r.r1 = s.x;
        r.r2 = s.y;
    }
    class State { volatile int x; int y; }
}
```

# Дело о неправильных метках: #2

```
class MyTest implements ConcurrencyTest<State, Res> {
    void actor1(State s, Res r) {
        s.y = 1;
        s.x = 1;
    }
    void actor2(State s, Res r) {
        int t = s.y;
        r.r1 = s.x;
        r.r2 = s.y;
    }
    class State { volatile int x; int y; }
}
```

Видим (r1, r2) = (1, 0):

- воспроизводится на на $\sim 10\%$ машин
- удаление «int t = y» устраняет проблему

# Дело о неправильных метках: #3

Ошибка в C1[2] CSE: проигнорировали `volatile` read:

$$
\begin{aligned}
t_1 &= y_1; \\
r1_2 &= x_2; \\
r2_1 &= y_1;
\end{aligned}
$$

...что дало возможность схлопнуть лоад:

$$
\begin{aligned}
t_1 &= y_1; \\
r1_2 &= x_2; \\
r2_1 &= t_1;
\end{aligned}
$$

[2] он же «клиентский компилятор»

# Дело о неправильных метках: #3

Ошибка в C1[2] CSE: проигнорировали `volatile` read:

$$
\begin{aligned}
t_1 &= y_1; \\
r1_2 &= x_2; \\
r2_1 &= y_1;
\end{aligned}
$$

...что дало возможность схлопнуть лоад:

$$
\begin{aligned}
t_1 &= y_1; \\
r1_2 &= x_2; \\
r2_1 &= t_1;
\end{aligned}
$$

Происходит только в C1,
а значит «из коробки» только на Windows.

[2] он же «клиентский компилятор»

# Дело о неправильных метках: #4

Тривиальная ошибка в компиляторе:

- `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7170145`
- Специфично только для -client

**FIXED**:

- В чётных ветках JDK7; JDK8
- Нет тестов, которые это поймали бы раньше
- С этого фейла и начались работы по перетряхиванию concurrency-тестов

# Дело о бессмертном референте: #1

Найден Кабутцем при построении тест-кейса в дискуссии на c-i@

```
final WeakReference<T> ref = new WR(obj);
```

| | |
|---|---|
| `while(ref.get() != null);` | `ref.clear();` |

# Дело о бессмертном референте: #1

Найден Кабутцем при построении тест-кейса в дискуссии на c-i@

```
final WeakReference<T> ref = new WR(obj);
```

| | |
|---|---|
| `while(ref.get() != null);` | `ref.clear();` |

Из разумных соображений, этот код должен отрабатывать нормально. На деле, первый поток застревает навсегда.

# Дело о бессмертном референте: #2

- Другие вводные:
  - воспроизводится уверенно на всех машинах
  - дизасм показывает редуцированный цикл


ORACLE

# Дело о бессмертном референте: #2

- Другие вводные:
  - воспроизводится уверенно на всех машинах
  - дизасм показывает редуцированный цикл
- Поле $Reference.referent$ не `volatile`
  - компилятор редуцирует код в зацикл, не реагирующий на очистки референса

```
T referent = ref.referent;
if (referent != null)
    while (true); // burn!
```

# Дело о бессмертном референте: #2

- Другие вводные:
  - воспроизводится уверенно на всех машинах
  - дизасм показывает редуцированный цикл
- Поле $Reference.referent$ не `volatile`
  - компилятор редуцирует код в зацикл, не реагирующий на очистки референса

```
T referent = ref.referent;
if (referent != null)
    while (true); // burn!
```

- `volatile` не поможет нативному GC

# Дело о бессмертном референте: #3

Продолбанное условие в компиляторе:

- `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7190310`
- Референс может выставить GC
- Уже и так достаточно сложный код ввиду pre/post-барьеров

**FIXED**:

- В чётных ветках JDK7; JDK8
- Теперь VM не склеивает лоады через safepoint


ORACLE

# Дело о бешеном потоке: #1

Случайно найден одним из студентов Кабутца:

| Thread t1; | Thread t2; |
|---|---|
| `while (!T.cT()`<br>`.isInterrupted());` | `t1.interrupt();` |

Из разумных соображений, первый поток должен останавливаться.

# Дело о бешеном потоке: #1

Случайно найден одним из студентов Кабутца:

| Thread t1; | Thread t2; |
|---|---|
| `while (!T.cT()`<br>`.isInterrupted());` | `t1.interrupt();` |

Из разумных соображений, первый поток должен останавливаться. ВНЕЗАПНО: он действительно корректно останавливается!

# Дело о бешеном потоке: #2

Вах, вытаскиваем предикат в отдельный метод:

```
                Thread t1; Thread t2;
def check() = T.cT().isInterrupted();
-------------------------------------
while (!check()); |  t1.interrupt();
```

[3] он же «серверный компилятор»

# Дело о бешеном потоке: #2

Вах, вытаскиваем предикат в отдельный метод:

```
                Thread t1; Thread t2;
def check() = T.cT().isInterrupted();
-------------------------------------
while (!check()); |  t1.interrupt();
```

И он «залипает»:

[3] он же «серверный компилятор»

# Дело о бешеном потоке: #2

Вах, вытаскиваем предикат в отдельный метод:

```
                Thread t1; Thread t2;
def check() = T.cT().isInterrupted();
-------------------------------------
while (!check()); |  t1.interrupt();
```

И он «залипает»:

- не фейлится в C1, уверенно фейлится в C2[3]
- break'и, модификаторы методов, etc. работают 50/50

[3] он же «серверный компилятор»

# Дело о бешеном потоке: #3

№1: `Thread.isInterrupted()` читает флаг в нативной структуре:

- `volatile` ставить некуда
- Обычно такое чтение требует вызова в VM
- C2 «вклеивает» вместо этого метода спецкод

# Дело о бешеном потоке: #3

№1: `Thread.isInterrupted()` читает флаг в нативной структуре:

- `volatile` ставить некуда
- Обычно такое чтение требует вызова в VM
- C2 «вклеивает» вместо этого метода спецкод

№2: Выносим инварианты из цикла:

- ...если не зависят от переменной индукции
- ...если не дают side effects
- ...если другим способом не нарушают JMM


ORACLE

# Дело о бешеном потоке: #4

Вклеенный код участвует в оптимизациях!

- выбрасывается из цикла, и привет.
- в нашем случае делает это в OSR-трамплине

# Дело о бешеном потоке: #4

Вклеенный код участвует в оптимизациях!

- выбрасывается из цикла, и привет.
- в нашем случае делает это в OSR-трамплине

Почему не ломается тривиальный пример?

| Thread t1; | Thread t2; |
|---|---|
| `while (!T.cT() .isInterrupted());` | `t1.interrupt();` |

# Дело о бешеном потоке: #4

Вклеенный код участвует в оптимизациях!

- выбрасывается из цикла, и привет.
- в нашем случае делает это в OSR-трамплине

Почему не ломается тривиальный пример?

| `Thread t1;` | `Thread t2;` |
|---|---|
| `while (!T.cT()`<br>`.isInterrupted());` | `t1.interrupt();` |

Ответ: Парные баги – нам просто **дико** повезло!

# Дело о бешеном потоке: #5

- Ошибка в оптимизирующем компиляторе:
  - `http://cs.oswego.edu/pipermail/concurrency-interest/2012-November/010184.html`
  - `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=8003135`
- **FIXED**:
  - В чётных ветках JDK7; JDK8
  - Выставлен эксплицитный барьер в голове интринзика
  - Удивительно, что это не было поймано раньше
  - (может быть и было, но никто не заметил)


ORACLE

# Дело о порванных шортах: #1

Всем известно про (не)атомарность long/double-ов. А что нам известно про другие типы?

```
short s = 0;
```

| | |
|---|---|
| `s = 0xFFFF;` | `short r1 = s;` |

# Дело о порванных шортах: #1

Всем известно про (не)атомарность long/double-ов. А что нам известно про другие типы?

```
short s = 0;
```

| | |
|---|---|
| `s = 0xFFFF;` | `short r1 = s;` |

JLS требует, что $r1 \in \{0x0000, 0xFFFF\}$.

# Дело о порванных шортах: #1

Всем известно про (не)атомарность long/double-ов. А что нам известно про другие типы?

```
short s = 0;
```

| | |
|---|---|
| `s = 0xFFFF;` | `short r1 = s;` |

JLS требует, что $r1 \in \{0x0000, 0xFFFF\}$.

Эксперимент показывает, что это требование выполняется.

# Дело о порванных шортах: #2

```
                    short s = 0;
```

| | |
|---|---|
| `s = 0xFFFF;` | `short t = s;`<br>`byte r1 = ((t >> 0) & 0xFF);`<br>`byte r2 = ((t >> 8) & 0xFF);` |

# Дело о порванных шортах: #2

```
                    short s = 0;
```

| | |
|---|---|
| `s = 0xFFFF;` | `short t = s;`<br>`byte r1 = ((t >> 0) & 0xFF);`<br>`byte r2 = ((t >> 8) & 0xFF);` |

Из интуитивных соображений:

$$(r1, r2) \in \{(0x00, 0x00), (0xFF, 0xFF)\}$$

# Дело о порванных шортах: #2

```
                    short s = 0;
```

| | |
|---|---|
| `s = 0xFFFF;` | `short t = s;` |
| | `byte r1 = ((t >> 0) & 0xFF);` |
| | `byte r2 = ((t >> 8) & 0xFF);` |

Из интуитивных соображений:

$$(r1, r2) \in \{(0x00, 0x00), (0xFF, 0xFF)\}$$

Эксперимент:

$$(r1, r2) \in \{..., (0x00, 0xFF), (0xFF, 0x00)\}$$


ORACLE

# Дело о порванных шортах: #3

```
                    short s = 0;
```

| | |
|---|---|
| `s = 0xFFFF;` | `short t = s;`<br>`byte r1 = ((t >> 0) & 0xFF);`<br>`byte r2 = ((t >> 8) & 0xFF);` |

# Дело о порванных шортах: #3

```
                    short s = 0;
```

| `s = 0xFFFF;` | `short t = s;`<br>`byte r1 = ((t >> 0) & 0xFF);`<br>`byte r2 = ((t >> 8) & 0xFF);` |
|---|---|

- C1 не фейлит, C2 фейлит уверенно
- так же фейлятся byte/char/short

# Дело о порванных шортах: #3

```
                    short s = 0;
```

| | |
|---|---|
| `s = 0xFFFF;` | `short t = s;`<br>`byte r1 = ((t >> 0) & 0xFF);`<br>`byte r2 = ((t >> 8) & 0xFF);` |

- C1 не фейлит, C2 фейлит уверенно
- так же фейлятся byte/char/short
- `volatile` не помогает

# Дело о порванных шортах: #4

```
short t = short_load(s.x);
r.r1 = byte_store(and(shift(t, 0), 0xFF)));
r.r2 = byte_store(and(shift(t, 8), 0xFF)));
```

# Дело о порванных шортах: #4

```
short t = short_load(s.x);
r.r1 = byte_store(and(shift(t, 0), 0xFF)));
r.r2 = byte_store(and(shift(t, 8), 0xFF)));
```

...превращается в:

```
short t = short_load(s.x);
r.r1 = byte_store(t);
r.r2 = byte_store(shift(t, 8));
```

# Дело о порванных шортах: #4

```
short t = short_load(s.x);
r.r1 = byte_store(and(shift(t, 0), 0xFF)));
r.r2 = byte_store(and(shift(t, 8), 0xFF)));
```

...превращается в:

```
short t = short_load(s.x);
r.r1 = byte_store(t);
r.r2 = byte_store(shift(t, 8));
```

...превращается в:

```
r.r1 = byte_store(unsigned_load(s.x));
r.r2 = byte_store(shift(signed_load(s.x), 8));
```

# Дело о порванных шортах: #5

```
short t = s.x;
r.r1 = (byte) ((t >> 0) & 0xFF);
r.r2 = (byte) ((t >> 8) & 0xFF);
```

...компилируется в:

```
; references: %rdx = $s; %rcx = $r
 movzwl 0xc(%rdx),%r11d       ; read s.x
 mov    %r11b,0xc(%rcx)       ; store r.r1
 movswl 0xc(%rdx),%r10d       ; read s.x again!
 shr    $0x8,%r10d            ; shift
 mov    %r10b,0xd(%rcx)       ; store r.r2
```

# Дело о порванных шортах: #5

```
short t = s.x;
r.r1 = (byte) ((t >> 0) & 0xFF);
r.r2 = (byte) ((t >> 8) & 0xFF);
```

...компилируется в:

```
; references: %rdx = $s; %rcx = $r
 movzwl 0xc(%rdx),%r11d       ; read s.x
 mov    %r11b,0xc(%rcx)       ; store r.r1
 movswl 0xc(%rdx),%r10d       ; read s.x again!
 shr    $0x8,%r10d            ; shift
 mov    %r10b,0xd(%rcx)       ; store r.r2
```

Атомарность, давай, до свиданья.


ORACLE

# Дело о порванных шортах: #6

Ошибка в компиляторе:

- `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=8000805`

**FIXED**:

- В чётных ветках JDK7; JDK8
- Конкретной оптимизации запрещено рождать лоады.

# Дело об атомных буферах: #1

Java даёт строгие гарантии на атомарность примитивных полей/массивов. Например:

```
byte[] b = new byte[100];
```

| | |
|---|---|
| `b[42] = (byte)(-1);` | `byte r1 = b[42];` |

# Дело об атомных буферах: #1

Java даёт строгие гарантии на атомарность примитивных полей/массивов. Например:

```
byte[] b = new byte[100];
```

| | |
|---|---|
| `b[42] = (byte)(-1);` | `byte r1 = b[42];` |

JLS гарантирует, что $r1 \in \{0x00, 0xFF\}$.

# Дело об атомных буферах: #2

В стандартной библиотеке есть буфера, которые «прикидываются» массивами. Что изменится в этом примере?

```
ByteBuffer b = BB.allocate(100);
```

| | |
|---|---|
| `b.put(42, (byte)(-1));` | `byte r1 = b.get(42);` |

# Дело об атомных буферах: #2

В стандартной библиотеке есть буфера, которые «прикидываются» массивами. Что изменится в этом примере?

```
ByteBuffer b = BB.allocate(100);
```

| | |
|---|---|
| `b.put(42, (byte)(-1));` | `byte r1 = b.get(42);` |

Оказывается, что $r1 \in \{0x00, 0xFF\}$.

# Дело об атомных буферах: #3

А вот это?

```
ByteBuffer b = BB.allocate(100);
```

| | |
|---|---|
| `b.putInt(42, -1);` | `int r1 = b.getInt(42);` |

# Дело об атомных буферах: #3

А вот это?

| ByteBuffer b = BB.allocate(100); | |
|---|---|
| b.putInt(42, -1); | int r1 = b.getInt(42); |

Ожидалось бы, что
$r1 \in \{0x00000000, 0xFFFFFFFF\}$.

# Дело об атомных буферах: #3

А вот это?

```
ByteBuffer b = BB.allocate(100);
```

| | |
|---|---|
| `b.putInt(42, -1);` | `int r1 = b.getInt(42);` |

Ожидалось бы, что
$r1 \in \{0x00000000, 0xFFFFFFFF\}$.

На деле: $r1 \in G$, где $G$ – множество всякого мусора, зависящее от платформы, endianness...

# Дело об атомных буферах: #4

Считается легальным поведением, ибо работа с несинхронизованным буфером запрещена JavaDoc.



# Дело об атомных буферах: #4

Считается легальным поведением, ибо работа с несинхронизованным буфером запрещена JavaDoc.

```
class Bits {
    ...
    static void putIntB(ByteBuffer bb, int bi, int x) {
        bb._put(bi    , int3(x));
        bb._put(bi + 1, int2(x));
        bb._put(bi + 2, int1(x));
        bb._put(bi + 3, int0(x));
    }
    ...
}
```

# Дело об атомных буферах: #5

Из этого неявно следует ещё одна хохма. Все они не атомарны *de jure*, а что *de facto*?

- ByteBuffer.allocate().putInt() не атомарен

# Дело об атомных буферах: #5

Из этого неявно следует ещё одна хохма. Все они не атомарны *de jure*, а что *de facto*?

- ByteBuffer.allocate().putInt() не атомарен
- ByteBuffer.allocateDirect().putInt()

# Дело об атомных буферах: #5

Из этого неявно следует ещё одна хохма. Все они не атомарны *de jure*, а что *de facto*?

- ByteBuffer.allocate().putInt() не атомарен
- ByteBuffer.allocateDirect().putInt() атомарен

# Дело об атомных буферах: #5

Из этого неявно следует ещё одна хохма. Все они не атомарны *de jure*, а что *de facto*?

- ByteBuffer.allocate().putInt() не атомарен
- ByteBuffer.allocateDirect().putInt() атомарен
- IntBuffer.allocate().putInt()

# Дело об атомных буферах: #5

Из этого неявно следует ещё одна хохма. Все они не атомарны *de jure*, а что *de facto*?

- ByteBuffer.allocate().putInt() не атомарен
- ByteBuffer.allocateDirect().putInt() атомарен
- IntBuffer.allocate().putInt() атомарен

# Дело об атомных буферах: #5

Из этого неявно следует ещё одна хохма. Все они не атомарны *de jure*, а что *de facto*?

- ByteBuffer.allocate().putInt() не атомарен
- ByteBuffer.allocateDirect().putInt() атомарен
- IntBuffer.allocate().putInt() атомарен
- IntBuffer.allocateDirect().putInt()

# Дело об атомных буферах: #5

Из этого неявно следует ещё одна хохма. Все они не атомарны *de jure*, а что *de facto*?

- ByteBuffer.allocate().putInt() не атомарен
- ByteBuffer.allocateDirect().putInt() атомарен
- IntBuffer.allocate().putInt() атомарен
- IntBuffer.allocateDirect().putInt() атомарен

# Дело об атомных буферах: #5

Из этого неявно следует ещё одна хохма. Все они не атомарны *de jure*, а что *de facto*?

- ByteBuffer.allocate().putInt() не атомарен
- ByteBuffer.allocateDirect().putInt() атомарен
- IntBuffer.allocate().putInt() атомарен
- IntBuffer.allocateDirect().putInt() атомарен
- ByteBuffer.allocate().asIntBuffer().putInt()

# Дело об атомных буферах: #5

Из этого неявно следует ещё одна хохма. Все они не атомарны *de jure*, а что *de facto*?

- ByteBuffer.allocate().putInt() не атомарен
- ByteBuffer.allocateDirect().putInt() атомарен
- IntBuffer.allocate().putInt() атомарен
- IntBuffer.allocateDirect().putInt() атомарен
- ByteBuffer.allocate().asIntBuffer().putInt() не атомарен

# Дело об атомных буферах: #6

- «Нельзя просто так взять, и сделать чтение»
- Атомарность невыровненных чтений не гарантируется на целевых платформах!



# Дело об атомных буферах: #6

- «Нельзя просто так взять, и сделать чтение»
- Атомарность невыровненных чтений не гарантируется на целевых платформах!
- Приходится выбирать из двух зол:
  1. Текущее, сразу видимое зло:
     - атомарность не гарантируется вообще никогда
     - невыровненное чтение никогда не фейлится
  2. Возможное, скрытое зло:
     - атомарность гарантируется, пока мы не пересекаем кеш-лайн
     - невыровненное чтение может внезапно зафейлиться


ORACLE

# Дело об атомных буферах: #7

Не очень ясно, что делать.

All Hail the Holy War:

- `http://cs.oswego.edu/pipermail/concurrency-interest/2012-December/010390.html`
- `http://mail.openjdk.java.net/pipermail/core-libs-dev/2012-December/013133.html`

# Дело о беге с барьерами: #1

Классический тест,
«Dekker idiom» (кусок Dekker Lock):

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

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

# Дело о беге с барьерами: #1

Классический тест,
«Dekker idiom» (кусок Dekker Lock):

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

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

При sequentially-consistent исполнении
$(r1, r2) \notin \{(0, 0)\}$.

# Дело о беге с барьерами: #1

Классический тест,
«Dekker idiom» (кусок Dekker Lock):

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

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

При sequentially-consistent исполнении
$(r1, r2) \notin \{(0, 0)\}$.
Эксперимент показывает, что это требование выполняется на всех известных нам реализациях.

# Дело о беге с барьерами: #2

- У маленькой такой компании есть маленький такой проц:
  - в проце куча регистров
  - побуждает тащить больше данных в регистрах
  - можно поправить register allocation
  - надо шедулить лоады как можно раньше

# Дело о беге с барьерами: #2

- У маленькой такой компании есть маленький такой проц:
  - в проце куча регистров
  - побуждает тащить больше данных в регистрах
  - можно поправить register allocation
  - надо шедулить лоады как можно раньше
- Маленькая компания имеет форк HotSpot'а
  - пилит HotSpot под себя
  - ранний шедулинг лоадов – тривиальная оптимизация.

# Дело о беге с барьерами: #3

- C2 обходится с кодом очень жёстко:
  - Перемалывает программу в data dependency graph, оптимизирует его, и упаковывает обратно.
  - Консистентность memory-эффектов там достигается специальными проекциями и барьерными нодами.

```
volatile int x, y;
x = 1;
r1 = y;
```

MB → store(x,1) → MB → load(r1,y) → MB



# Дело о беге с барьерами: #4

Одна из оптимизаций клеит лишние барьеры:

`MB` $\rightarrow$ `store(x,1)` $\rightarrow$ `MB` $\rightarrow$ `load(r1,y)` $\rightarrow$ `MB`

# Дело о беге с барьерами: #4

Одна из оптимизаций клеит лишние барьеры:

`MB` $\rightarrow$ `store(x,1)` $\rightarrow$ `MB` $\rightarrow$ `load(r1,y)` $\rightarrow$ `MB`

В этой оптимизации есть ошибка.

# Дело о беге с барьерами: #4

Одна из оптимизаций клеит лишние барьеры:

`MB` $\rightarrow$ `store(x,1)` $\rightarrow$ `MB` $\rightarrow$ `load(r1,y)` $\rightarrow$ `MB`

В этой оптимизации есть ошибка. Она считает, что перед `volatile read` *всё равно* есть leading-барьер, и стирает актуальный:

`MB` $\rightarrow$ `store(x,1)` $\rightarrow$ `load(r1,y)` $\rightarrow$ `MB`

# Дело о беге с барьерами: #5

Внезапно срабатывает та самая маленькая оптимизация маленькой компании:

```
MB → store(x,1) → load(r1,y) → MB
MB → load(r1,y) → store(x,1) → MB
```

# Дело о беге с барьерами: #5

Внезапно срабатывает та самая маленькая оптимизация маленькой компании:

```
MB → store(x,1) → load(r1,y) → MB
MB → load(r1,y) → store(x,1) → MB
```

Что вкупе со вторым потоком:

```
MB → load(r1,y) → store(x,1) → MB
MB → load(r2,x) → store(y,1) → MB
```

...даёт нам $(r1, r2) = (0, 0)$. Упс.

# Дело о беге с барьерами: #6

Проблема исправляется:

- `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=8007898`

# Дело о беге с барьерами: #6

Проблема исправляется:

- `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=8007898`

Кроме того:

- Нет релевантных падений на других тестах
- Написали внутрь VM instruction scheduling fuzzer, падение только на этом тесте

# Дело о б. перестановках: #1

Те же парни оптимизируют свою платформу дальше:

```
                    AtomicInteger ai;
-------------------------------------------------------
ai = new AtomicInteger(42); | r1 = ai.get();
```

# Дело о б. перестановках: #1

Те же парни оптимизируют свою платформу дальше:

| AtomicInteger ai; | |
|---|---|
| ai = new AtomicInteger(42); | r1 = ai.get(); |

Из здравого смысла, в отсутствие NPE: $r1 \in \{42\}$.

# Дело о б. перестановках: #2

```
class A {
   volatile int f;
   A(int v) { f = v; }
}

A a;
```

| | |
|---|---|
| `a = new A(42);` | `r1 = a.f;` |

# Дело о б. перестановках: #2

```
class A {
   volatile int f;
   A(int v) { f = v; }
}

A a;
```

| | |
|---|---|
| `a = new A(42);` | `r1 = a.f;` |

Опять же, из здравого смысла, в отсутствие NPE:

$$r1 \in \{42\}.$$

# Дело о б. перестановках: #3

```
class A {
   volatile int f;
   A(int v) { f = v; }
}

A a;
```

| | |
|---|---|
| `A ta = <CreateObject>;` | `A ta = a;` |
| `ta.f = 42;` | `r1 = ta.f;` |
| `a = ta;` | |

...

# Дело о б. перестановках: #3

```
class A {
   volatile int f;
   A(int v) { f = v; }
}

A a;
```

| | |
|---|---|
| `A ta = <CreateObject>;` | `A ta = a;` |
| `ta.f = 42;` | `r1 = ta.f;` |
| `a = ta;` | |

reorder!

...

# Дело о б. перестановках: #4

```
class A {
   volatile int f;
   A(int v) { f = v; }
}

A a;
```

| | |
|---|---|
| `A ta = <CreateObject>;` | `A ta = a;` |
| `a = ta;` | `r1 = ta.f;` |
| `ta.f = 42;` | |

Ать! $r1 \in \{0, 42\}$?!

# Дело о 6. перестановках: #5

Получается, что идиома «поменяй свои `final` на `volatile` и ничего не сломается», неверна?



# Дело о 6. перестановках: #5

Получается, что идиома «поменяй свои `final` на `volatile` и ничего не сломается», неверна?

Или выходит, что JSR 133 Cookbook нам врёт?

# Дело о 6. перестановках: #6

Проблема в наивном понимании JMM.
На деле, JMM определяет, что такое
«well-formed execution».



# Дело о 6. перестановках: #6

Проблема в наивном понимании JMM.
На деле, JMM определяет, что такое «well-formed execution».

Вот это не well-formed execution, который приводит к $r1 = 0$:

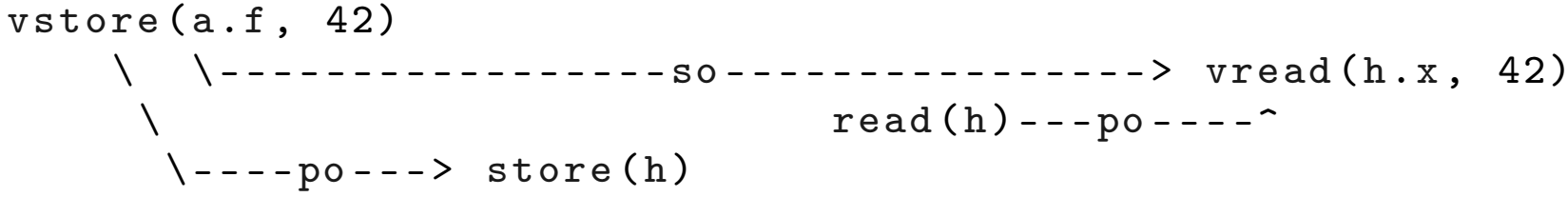
```
read(a)
    \--po--> vread(a.f, 0)
                        \----so---> vstore(a.f, 42)
                                              \---po----> store(a)
```

# Дело о 6. перестановках: #7

Проблема в наивном понимании JMM.
На деле, JMM определяет, что такое «well-formed execution».

Единственный well-formed execution приводит к чтению $r1 = 42$:



# Дело о 6. перестановках: #8

- **BUSTED**. Status quo обеспечивается логикой барьеров в реализации HotSpot:
  - если её ослабить, то можно получить ошибки
  - парни ослабили в своём форке, сверились с cookbook'ом... потом запустили наши тесты, а они сломались!


ORACLE

# Дело о 6. перестановках: #8

- **BUSTED**. Status quo обеспечивается логикой барьеров в реализации HotSpot:
  - если её ослабить, то можно получить ошибки
  - парни ослабили в своём форке, сверились с cookbook'ом... потом запустили наши тесты, а они сломались!

«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.»

# Дело о 6. перестановках: #8

- **BUSTED**. Status quo обеспечивается логикой барьеров в реализации HotSpot:
  - если её ослабить, то можно получить ошибки
  - парни ослабили в своём форке, сверились с cookbook'ом... потом запустили наши тесты, а они сломались!

«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, 2013)


ORACLE

# Дело о фантомном ППЦэ: #1

Просто массив, просто default values:

```
int[] arr;
```

| | |
|---|---|
| `arr = new int[1];` | `r1 = arr[0];` |

# Дело о фантомном ППЦэ: #1

Просто массив, просто default values:

| `int[] arr;` | |
|---|---|
| `arr = new int[1];` | `r1 = arr[0];` |

Из здравого смысла: $r1 \in \{0\}$.

# Дело о фантомном ППЦэ: #1

Просто массив, просто default values:

| `int[] arr;` | |
|---|---|
| `arr = new int[1];` | `r1 = arr[0];` |

Из здравого смысла: $r1 \in \{0\}$.

А вот на некотором серваке: $r1 \in G$,
где $G$ – множество всякого мусора.

# Дело о фантомном ППЦэ: #2

- Обычные тесты:
  - падает только на конкретном 6-core PPC
  - воспроизводится только на массивах
  - `volatile` на массиве «чинит» проблему

# Дело о фантомном ППЦэ: #2

- Обычные тесты:
  - падает только на конкретном 6-core PPC
  - воспроизводится только на массивах
  - `volatile` на массиве «чинит» проблему
- Дальше – больше:
  - барьеры выставлены правильно, в соответствии со спекой
  - роем...

# Дело о фантомном ППЦэ: #2

- Обычные тесты:
  - падает только на конкретном 6-core PPC
  - воспроизводится только на массивах
  - `volatile` на массиве «чинит» проблему
- Дальше – больше:
  - барьеры выставлены правильно, в соответствии со спекой
  - роем... роем...

# Дело о фантомном ППЦэ: #2

- Обычные тесты:
  - падает только на конкретном 6-core PPC
  - воспроизводится только на массивах
  - `volatile` на массиве «чинит» проблему
- Дальше – больше:
  - барьеры выставлены правильно, в соответствии со спекой
  - роем... роем... роем...

# Дело о фантомном ППЦэ: #2

- Обычные тесты:
  - падает только на конкретном 6-core PPC
  - воспроизводится только на массивах
  - `volatile` на массиве «чинит» проблему
- Дальше – больше:
  - барьеры выставлены правильно, в соответствии со спекой
  - роем... роем... роем... находим!
  - «Yet, hpcx exhibits non-SC behaviours for some A-cumulativity tests, [...]. We understand that this is due to an erratum in the Power 5 implementation. IBM is providing a workaround, replacing the sync barrier by a short code sequence [...]»

# Дело о фантомном ППЦэ: #3

- Хардварный баг!
  - что делать, не очень ясно
  - к результатам на конкретной машине теперь относятся подозрительно
  - (потенциально отправляется на свалку)

- `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=8007283`
- `http://www0.cs.ucl.ac.uk/staff/j.alglave/papers/cav10.pdf`

# Дело о тоталитарных порядках: #1

Есть классический тест, IRIW:

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

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

# Дело о тоталитарных порядках: #1

Есть классический тест, IRIW:

```
                volatile int x, y;
-------------------------------------------------
x = 1;  |  y = 1;  |  int r1 = y;  |  int r3 = x;
        |          |  int r2 = x;  |  int r4 = y;
```

Состояние $(r1, r2, r3, r4) = (1, 0, 1, 0)$ запрещено: прочитали x и y в разном порядке.

# Дело о тоталитарных порядках: #2

- Сравнительно новое наблюдение:
  - практически все платформы обеспечивают «write atomicity»
  - апдейт в конкретное место виден сразу всем
  - единственная платформа, где это не выполняется – PowerPC

# Дело о тоталитарных порядках: #2

- Сравнительно новое наблюдение:
  - практически все платформы обеспечивают «write atomicity»
  - апдейт в конкретное место виден сразу всем
  - единственная платформа, где это не выполняется – PowerPC
- Простого acquire недостаточно:

```
Load Acquire: ld; cmp; bc; isync
Load Seq Cst: hwsync; ld; cmp; bc; isync
```

# Дело о тоталитарных порядках: #2

- Сравнительно новое наблюдение:
  - практически все платформы обеспечивают «write atomicity»
  - апдейт в конкретное место виден сразу всем
  - единственная платформа, где это не выполняется – PowerPC
- Простого acquire недостаточно:

```
Load Acquire: ld; cmp; bc; isync
Load Seq Cst: hwsync; ld; cmp; bc; isync
```

- Требуется два барьера на volatile read'е, чтобы обеспечить SC!


ORACLE

# Дело о тоталитарных порядках: #3

- **BUSTED.**
  Та же беда с JSR133 cookbook'ом:
  - найдена в ходе стандартизации C++11 MM
  - до сих пор не отражена в JSR133 Cookbook!
  - разработчики портов на PPC негодуэ

- `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2177.html`
- `http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`
- `http://cs.oswego.edu/pipermail/concurrency-interest/2013-January/010608.html`

# Выводы


# Выводы: везде баги

- Bugs, bugs everywhere:
  - В приложениях куча ошибок.
  - В JDK куча ошибок.
  - В JVM куча ошибок.
  - В хардваре куча ошибок.
  - В документации куча ошибок.

# Выводы: везде баги

- Bugs, bugs everywhere:
  - В приложениях куча ошибок.
  - В JDK куча ошибок.
  - В JVM куча ошибок.
  - В хардваре куча ошибок.
  - В документации куча ошибок.
- Мы работаем над этим, но там поле непаханное.

# Выводы: везде баги

- Bugs, bugs everywhere:
  - В приложениях куча ошибок.
  - В JDK куча ошибок.
  - В JVM куча ошибок.
  - В хардваре куча ошибок.
  - В документации куча ошибок.
- Мы работаем над этим, но там поле непаханное.
- Баги исправляются со временем
  $\Rightarrow$ планируйте обновления софта и железа!

# Выводы: нам нужна помощь!

- Не проходи мимо:
  - Увидел подозрительное поведение, потыкай в него палочкой.
  - Потыкал и ничего не понял, спроси.
  - Потыкал и понял, что оно поломано, рапортуй!
  - Потыкал, понял, знаешь как исправить – тем более рапортуй!

# Выводы: нам нужна помощь!

- Не проходи мимо:
  - Увидел подозрительное поведение, потыкай в него палочкой.
  - Потыкал и ничего не понял, спроси.
  - Потыкал и понял, что оно поломано, рапортуй!
  - Потыкал, понял, знаешь как исправить – тем более рапортуй!
- Пишите в Спортлото!
  - `concurrency-interest@cs.oswego.edu`
  - или лично, `aleksey.shipilev@oracle.com`
  - мы с радостью потыкаем в *это* вместе с вами.

# Выводы: Personal Appeal

Дяденьки и тётеньки, вы вместо очередного %bullshit%.js, лучше попишите тестов для вашей платформы, пожалуйста. а?