

Java Memory Model

Проблемы многопоточного кода

Видимость (Visibility)

Один поток может в какой-то момент временно сохранить значение некоторых полей не в основную память, а в регистры или локальный кэш процессора, таким образом второй поток, выполняемый на другом процессоре, читая из основной памяти, может не увидеть последних изменений поля. И наоборот, если поток на протяжении какого-то времени работает с регистрами и локальными кэшами, читая данные оттуда, он может сразу не увидеть изменений, сделанных другим потоком в основную память.

Проблемы многопоточного кода

Изменение порядка (reordering)

Для увеличения производительности процессор/компилятор могут переставлять местами некоторые инструкции/операции. Вернее, с точки зрения потока, наблюдающего за выполнением операций в другом потоке, операции могут быть выполнены не в том порядке, в котором они идут в исходном коде.

Также эффект reordering может наблюдаться, когда один поток кладет результаты первой операции в регистр или локальный кэш, а результат второй операции кладет непосредственно в основную память. Тогда второй поток, обращаясь к основной памяти может сначала увидеть результат второй операции, и только потом первой, когда все регистры или кэши синхронизируются с основной памятью.

Еще одна причина reordering, может заключаться в том, что процессор может решить поменять порядок выполнения операций, если, например, сочтет что такая последовательность выполнится быстрее.

```
public class Container {  
    private final List<String> list = new ArrayList<>();  
  
    synchronized void addEntry(String s) {  
        list.add(s);  
    }  
  
    int size() {  
        return list.size();  
    }  
}
```

```
public static void main(String[] args) {  
    Container container = new Container();  
    Runnable runnable = () -> {  
        for(int i=0; i<100000; i++) {  
            container.add("Entry");  
        }  
    };  
    List<Thread> threads = new ArrayList<>();  
    for (long count = 10; count > 0; count--) {  
        Thread thread = new Thread(runnable);  
        thread.start();  
        threads.add(thread);  
    }  
    System.out.println("Size is "+container.size());  
    while(container.size() < 1000000) {}  
    System.out.println("Finished!");  
}
```

Оптимизация компилятора

```
public static void main(String[] args) {  
    Container container = new Container();  
    Runnable runnable = () -> {  
        for(int i=0; i<100000; i++) {  
            container.addEntry("Entry");  
        }  
    };  
    List<Thread> threads = new ArrayList<>();  
    for (long count = 10; count > 0; count--) {  
        Thread thread = new Thread(runnable);  
        thread.start();  
        threads.add(thread);  
    }  
    System.out.println("Size is " + container.size());  
    int size$0 = container.list.size;  
    if (size$0 < 1000000) {  
        while (true) {}  
    }  
    System.out.println("Finished!");  
}
```

Изменение порядка КОМПИЛЯТОРА

```
public class Visibility {  
    private boolean x = false;  
    private boolean y = false;  
  
    public void set(){  
        y = true;  
        x = true;  
    }  
  
    public void test(){  
        while(!x);  
        assert(!y);  
    }  
}
```

```
public static void main(String[] args) {  
    Visibility visibility = new Visibility();  
    new Thread(visibility::set).start();  
    new Thread(visibility::test).start();  
}
```

...


Console:

Exception in thread "Thread-1"
java.lang.AssertionError

JCStress test visibility

```
@JCStressTest
public class TestVisibility {
    int x;
    int y;
    @Actor
    public void actor1() {
        x = 1;
        y = 1;
    }
    @Actor
    public void actor2(Il_Result r) {
        r.r1 = y;
        r.r2 = x;
    }
}
```

Observed state	Occurrences
0, 0	5,108,132
0, 1	147,942
1, 0	98
1, 1	191,151,899

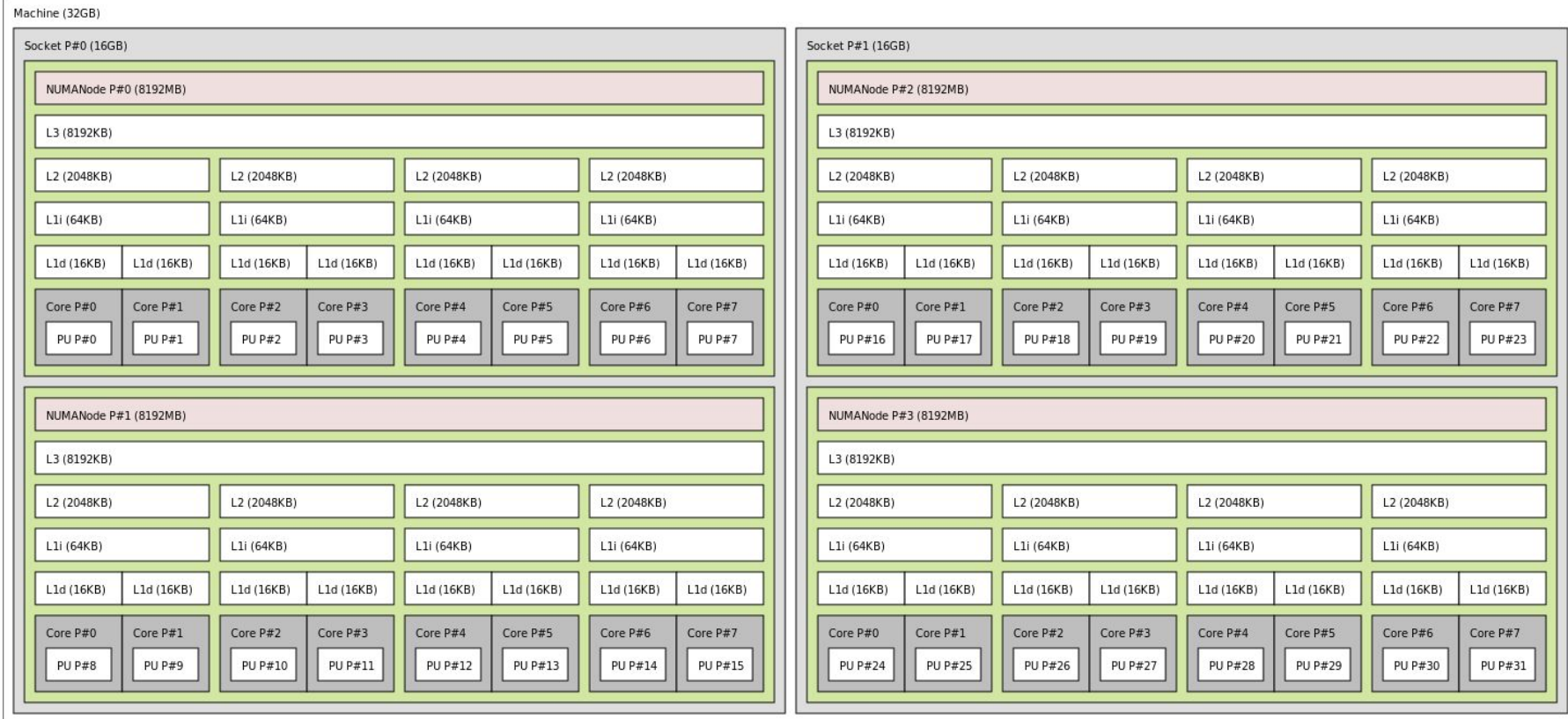


JCStress Tool → <https://wiki.openjdk.java.net/display/CodeTools/jcstress>

Изменение порядка в процессоре

$r1 := [ADDR1]$ → нет в кэше, пошли в главную память
 $r1 := r1 * 2$ → зависит от предыдущего, пока пропускаем
 $[ADDR1] := r1$ → тоже пропускаем
 $r2 := [ADDR2]$ → это можно пока сделать. ADDR2 в кэше
 $r2 := r2 * 2$ → это быстро
 $[ADDR2] := r2$ → пишем назад в кэш, тоже быстро

Иерархия кэшей AMD Bulldozer



Intel

9-е поколение настольных процессоров

9TH GENERATION INTEL® CORE™ DESKTOP PROCESSOR COMPARISONS¹



Maximum Processor Frequency (GHz)	Up to 5.0	Up to 4.9	Up to 4.6
Number of Processor Cores/Threads	8/16	8/8	6/6
Intel® Turbo Boost Technology 2.0	Yes	Yes	Yes
Intel® Hyper-Threading Technology	Yes	No	No
Intel® Smart Cache Size (MB)	16	12	9

MESI

протокол управления состоянием линий кэша

Modified (M)

строка кэша присутствует только в текущем кэше и является грязной - она была изменена из значения в основной памяти.

Exclusive (E)

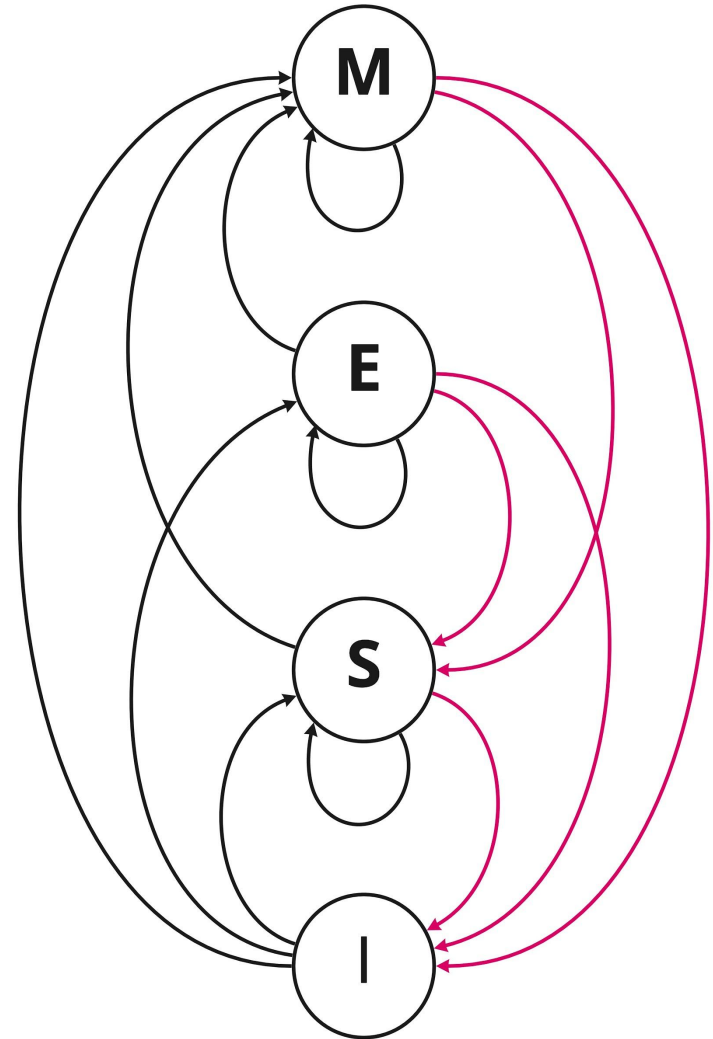
строка кэша присутствует только в текущем кэше, но она чистая - она соответствует основной памяти.

Shared (S)

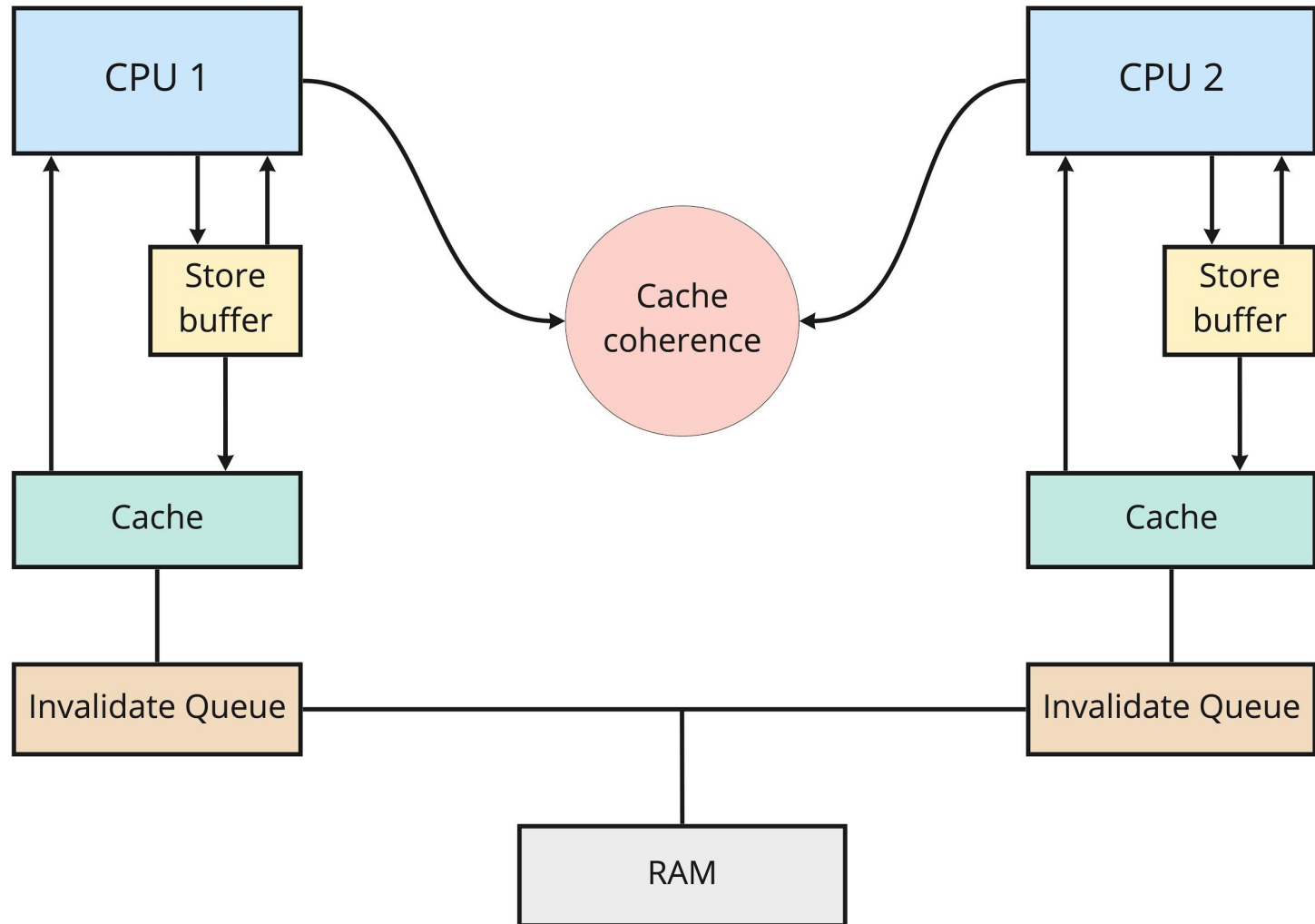
строка кэша может храниться в других кэшах машины и является чистой - она соответствует основной памяти.

Invalid (I)

строка кэша недействительна (не используется).



Схематично архитектура



Барьеры памяти (memory barriers)

Вид барьерной инструкции, которая приказывает компилятору (при генерации инструкций) и центральному процессору (при исполнении инструкций) устанавливать строгую последовательность между обращениями к памяти до и после барьера. Это означает, что все обращения к памяти перед барьером будут гарантированно выполнены до первого обращения к памяти после барьера.

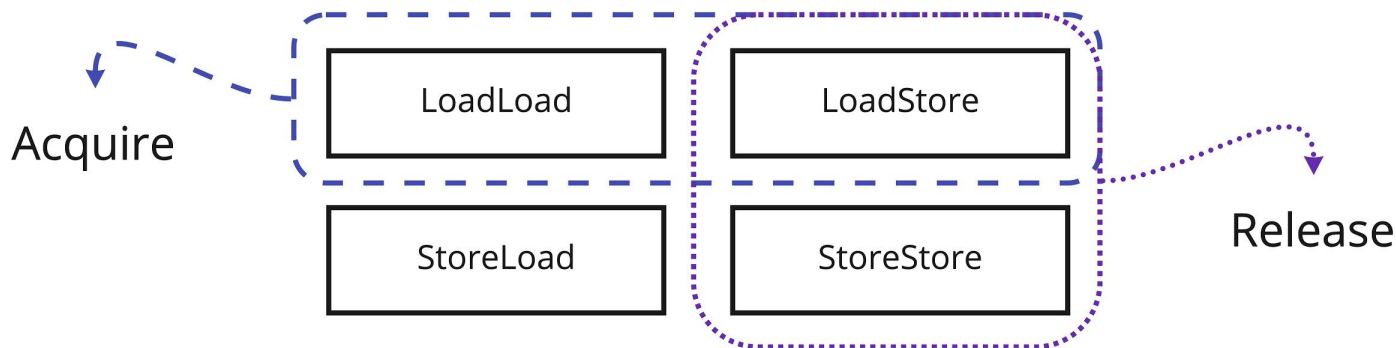
Типы барьеров

Load/Load – упорядочивает предыдущие load-инструкции с последующими.

Store/Store – упорядочивает предыдущие store-инструкции с последующими.

Load/Store – упорядочивает предыдущие load-инструкции с последующими store.

Store/Load – упорядочивает предыдущие store-инструкции с последующими load.



Все операции чтения будут выполнены до любых операций после acquire
Любые операции будут выполнены до всех операций чтения после release

Java Memory Model

- Часть спецификации языка Java (JLS 17.4)

<https://docs.oracle.com/javase/specs/jls/se12/html/jls-17.html#jls-17.4>

- Описывает взаимодействие приложения с памятью
- Даёт определённые гарантии относительно того, какие записи в память когда и как могут быть видимы
- Не зависит от реализации JVM, операционной системы и железа

Happens-before

В Java Memory Model введена такая абстракция как happens-before. Она обозначает, что если операция X связана отношением happens-before с операцией Y, то весь код следуемый за операцией Y, выполняемый в одном потоке, видит все изменения, сделанные другим потоком, до операции X.

Операции связанные отношением happens-before

- В рамках одного потока любая операция happens-before любой операцией следующей за ней в исходном коде
- Освобождение лока (unlock) happens-before захват того же лока (lock)
- Выход из synchronized блока/метода happens-before вход в synchronized блок/метод на том же мониторе
- Запись volatile поля happens-before чтение того же самого volatile поля
- Завершение метода run экземпляра класса Thread happens-before выход из метода join() или возвращение false методом isAlive() экземпляром того же треда
- Вызов метода start() экземпляра класса Thread happens-before начало метода run() экземпляра того же треда
- Завершение конструктора happens-before начало метода finalize() этого класса
- Вызов метода interrupt() на потоке happens-before когда поток обнаружил, что данный метод был вызван либо путем выбрасывания исключения InterruptedException, либо с помощью методов isInterrupted() или interrupted()

Правила happens-before

- связь happens-before транзитивна, т.е. если X happens-before Y, а Y happens-before Z, то X happens-before Z.
- освобождение/захват монитора связаны отношением happens-before, только если операции проводятся над одним и тем же экземпляром объекта.
- в отношении happens-before участвуют только два потока, о видимости и reordering остальных потоков ничего сказать нельзя, пока в каждом из них не наступит отношение happens-before с к другим потоком.

Volatile

- Запись и чтение в поле, объявленное `volatile`, называется `volatile read`, `volatile write`

Речь идёт непосредственно о записи, а не о записи членов/элементов массива

```
volatile int[] x;  
x = new int[10]; // volatile write  
x[0] = 1; // volatile read, plain write
```

- `volatile` - обладает `release` семантикой по записи и `acquire` по чтению

Атомарность (atomicity)

Операция атомарна

если невозможно наблюдать частичный результат ее выполнения. Любой наблюдатель видит либо состояние системы до атомарной операции, либо после.

- Запись в поле типа **boolean, byte, short, char, int, Object** всегда атомарна
- Запись в поле типа **long/double**: атомарна запись старших и младших 32 бит
- Запись в поле типа **long/double**, объявленное **volatile**, атомарна

JCStress test volatile

```
@JCStressTest
public class TestVisibility {
    int x;
    volatile int y;
    @Actor
    public void actor1() {
        x = 1;
        y = 1;
    }
    @Actor
    public void actor2(Il_Result r) {
        r.r1 = y;
        r.r2 = x;
    }
}
```