ФОРМАЛИЗАЦИЯ JAVA MEMORY MODEL ПОСРЕДСТВОМ ПРИМИТОВОВ СИНХРОНИЗАЦИИ

Any

21 апреля 2024 г.

Аннотация

В данной работе классифицируются некоторые понятия синхронизации из ЯВУ Java.

Введение

Некоторые программисты, во время разработки, думают будто бы программа делает ровно то, что они ей говорят. Такое мнение ошибочно, более того — опасно: разработчик недооценивает сложность того, как все устроено на самом деле и думает, что он-то все понимает, и из-за этого может возникать множество проблем. Например можно подумать, что знаешь все последсвия от состояния гонки [7].

С таким сталкиваются проффесионалы с огромным опытом [11], что говорить о тех, у кого его нет. Из-за этого, переходя к многопточности, однозначно важно отказаться от таких предубеждений.

Основной причиной несовершенства и невероятной сложности, служит тот факт, что поддережание даже ослабленной версии подобной модели — это потеря многих процентов производительности [6; 10]. И компиляторы, JIT, ОС, оптимизаторы процессора, делают очень, очень много удивительных вещей, настолько удивительных, что не осталось людей, которые могут полностью понимать то как эти сложнейшие системы взаимодействуют.

К счастью, «Любую проблему можно решить введением дополнительного уровня абстракции кроме одной — слишком большого количества уровней абстракции» [1].

Потому мы сосредоточимся исключительно на Java Memory Model (Далее JMM) не уходя в бесполезные попытки спуска вниз, как видно на рисунке 1, и рассмотрим те гарантии, что она дает, а также заглянем выше, к примитивам синхронизации и попробуем формально доказать их корректность.

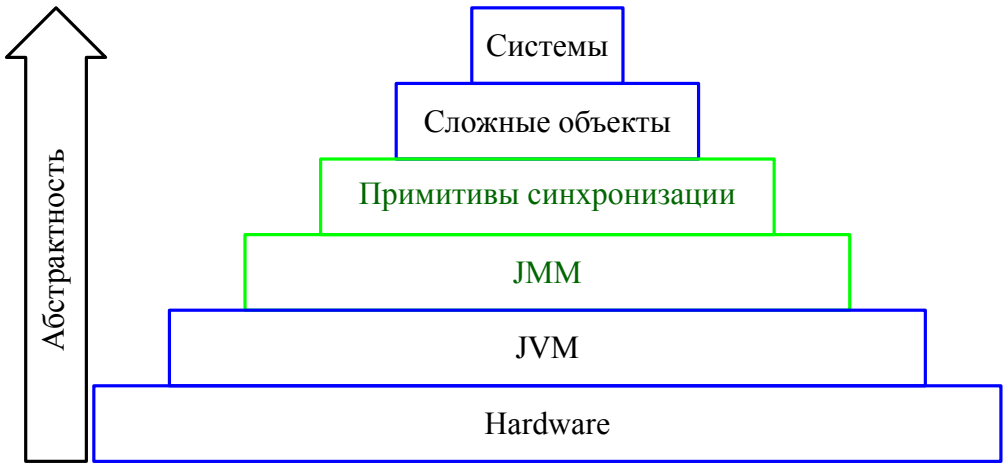


Рисунок 1: Намеченный путь

1 Исполнение абстрактной машины

Поскольку java virtual machine (Далее JVM) существует больше одной, спецификация, абстрагируясь от конкретных реализаций, вводит понятие абстрактной машины.

Её исполнение представляет собой поведение, которое разрешено проявлять многопоточным программам, работающим с разделяемыми переменными.

Разделяемыми переменными считается память, которая может быть общей для всех потоков, называемая общей памятью или памятью кучи Все поля экземпляра, статические поля и элементы массива хранятся в памяти кучи. Мы используем термин «переменная» для обозначения как полей, так и элементов массива. Переменные, локальные для метода, никогда не разделяются между потоками и не зависят от модели памяти [3, Глава 17.4.1].

Говоря иначе, спецификация отвечает на вопрос, какое значение видят разделяемые переменные при очередном чтении. Где видимостью, можно считать гарантию возможности прочтения какой-либо записи.

Поскольку в сущности описывается поведение именно памяти, традиционно это называется java memory model (Далее JMM).

1.1 Действия

Для начала, мы абстрагируемся от локальных действий, конструкций, семантик, оставляя лишь межпоточные действия, стоящие за ними, которые и образуют исполнение. Все локальное, происходящие внутри потоков и нас не интересущее, мы называем внутрипоточной семантикой. Наша же речь идет о межпоточной семантике.

Существует несколько видов межпоточных действий, которые может выполнять программа. Вот некоторые из них:

I. Чтение (обычное, или non-volatile). Чтение переменной.

II. Запись (обычная, или non-volatile). Запись переменной.

III. Synchronization Action, а именно:

   i) Volatile чтение переменной.

   ii) Volatile запись переменной.

   iii) Lock монитора.

   iv) Unlock монитора.

[3, Глава 17.4.2]

Удобно считать, что существует некое множество A исполнения E, которое содержит все действия программы. [3, Глава 17.4.6]

1.2 Порядки

ЧУМ Чуть отходя, представим себе множество элементов S. Наложим на это множество бинарную операцию: $\langle S, \phi \rangle$. Где, ϕ обладает следующими свойствами:

1. Рефлексивность: $\forall a \ (a \phi a)$

2. Антисимметричность: $\forall a, b\ (a \phi b) \wedge (b \phi a) \rightarrow a = b$

3. Транзитивность: $\forall a, b, c\ (a \phi b) \wedge (b \phi c) \rightarrow a \phi c$

В итоге получаем частично упорядоченное множество (Далее ЧУМ). Наглядным примером такого множества будет ряд натуральных чисел и операция $\leq$ на них. При этом важно понимать, что такие свойства, вполне допускают случай, когда для каких-то двух элементов операция ϕ ни имеет смысла. Именно поэтому множество частично упорядочено.

Это можно представить как множество апельсинов и яблок. Допустим у нас существует некоторая операция η, которая может упорядочить апельсины и яблоки отдельно, но при этом между собой они несравнимы.

ЛУМ Если же у ЧУМ вида $\langle S, \theta \rangle$, операция θ обладает свойством тотальности:

$$\forall a, b\ (a \phi b) \vee (b \phi a)$$

То мы можем говорить о том, что это линейно упорядоченное множество (Далее ЛУМ). [3, Глава 17.4.7]

Более близким нам примером ЛУМа будет, наложение упорядочивающей бинарной операции на все элементы A. Мы получим наивные представление о программе, как о машине Фон Неймана, а именно последовательно согласованную модель. «Последовательно согласованная модель (Далее ПСМ) — результат любого выполнения будет таким же, как если бы операции всех процессоров выполнялись в некотором последовательном порядке, а операции каждого отдельного процессора появлялись бы в этой последовательности в порядке, заданном его программой. [5]»

В этом определении интересно то, что оптимизатору дается откуп на любые оптимизации, лишь бы результат был тот же. Такое поведение называется семантикой as-if-serial, а её отсутсвие есть строго согласованная модель. [2, с. 336—337] Очевидно, что в Java любое ЛУМ, также и ПСМ.

Программный порядок Взяв из множества A подмножество действий некоторого потока t, мы получим A_t. Программным же порядком (Далее PO, от program order) можно считать ЛУМ вида $\left\langle A_t, \xrightarrow{\text{PO}} \right\rangle$. [3, Глава 17.4.3]

В основном именно с его помощью мы переносим информацию о программе в исполнение.

Синхронизационный порядок Вычленив из множества A все синхронизационные действия, у нас выйдет подмножество A_s. Синхронизационным порядком (Далее SO, от synchronization order), мы считаем ЛУМ вида $\left\langle A_s, \xrightarrow{\text{SO}} \right\rangle$. [3, Глава 17.4.4]

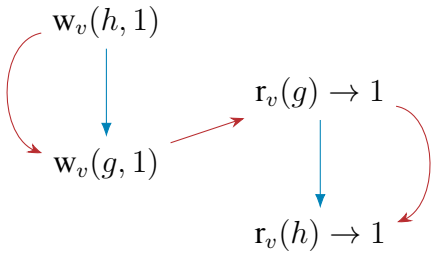
В отличии от PO SO выходит за рамки отдельного потока и позволяет упорядочивать действия между ними. Можно даже сказать, что именно это основа какой бы то нибыло синхронизации программ.

Для лучшего понимания, рассмотрим пример представленный в таблице 1.

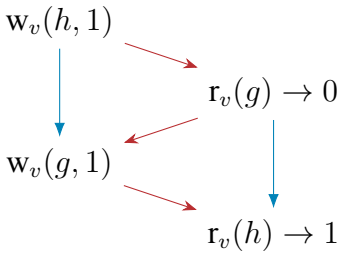
Таблица 1: Volatile чтение

`volatile int g, h`	
`h = 1`	`int r1 = g`
`g = 1`	`int r2 = h`

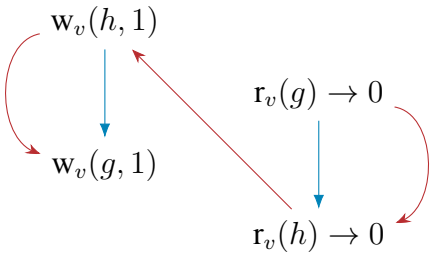
Сформируем из этой программы действия и упорядочим их уже известными нам операциями. Что проилюстрировано на рисунке 2



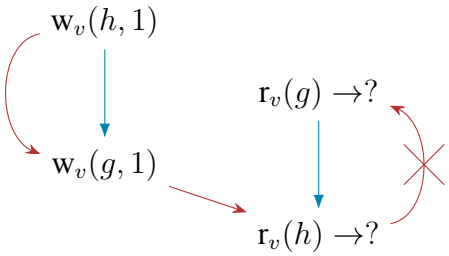
(a) Полная видимость



(b) Частичная видимость



(c) Полная невидимость



(d) Невозможное исполнение

Рисунок 2: Связь PO и SO

Заметим, что во первых возможных исполнений несколько. А во вторых, связываются все A_s в одну общую цепочку. И как видно по рисунку 2d, существуют некоторые невозможные исполнения. Это следует из того свойства, что SO согласуется с PO. Так, можно даже представлять, что их операция общая на всем множестве A. И в том примере мы нарушаем свойство антисимметричности.

В рисунке 2a $\mathrm{r}_v(h)$ увидит 1, поскольку $\mathrm{w}_v(h,1) \xrightarrow{\text{SO}} \mathrm{r}_v(h)$, по свойству транизитивности. Анологично и для 2b, но при этом заметим, что не существует связи $\mathrm{w}_v(g,1) \xrightarrow{\text{SO}} \mathrm{r}_v(g)$, поэтому и будет виден 0.

В итоге, невозможно исполнение, когда в $(r1, r2)$ получится $(1,0)$. При этом допустимы все следующие значения: $(1,1)$, $(0,1)$ и $(0,0)$. Как следствие, JMM выбирает любое из них.

Межпоточный мост Для последующих рассуждений необходимо дать чуть больше свободы порядкам, сформировав ЧУМ на основе SO, где synchronized-with (Далее SW) подпорядок SO, вида $\left\langle A_s, \xrightarrow{\text{SW}} \right\rangle$. Формирующийся по определенным правилам. Вот, некоторые из них:

- Действие по unlock монитора `m` SW со всеми последующими действиями по lock `m` (где «последующие» определяется в соответствии с SO).
- Запись в `volatile` переменную `v` SW со всеми последующими чтениями `v` любым потоком (где «последующие» определяется в соответствии с SO).
- Запись значения по умолчанию (`0`, `false`, `null`) в каждую переменную SW с первым действием в каждом потоке.

[3, Глава 17.4.4]

Объединение SW и PO Помните мы говорили о представлении как об общем операторе SO и PO? Так вот, оператор happen-before (Далее HB) имеено такой. И в отличии от слабо связанных SO и PO, HB есть объединение SW и PO, являясь ЧУМ вида $\left\langle A, \xrightarrow{\text{HB}} \right\rangle$. [3, Глава 17.4.5]

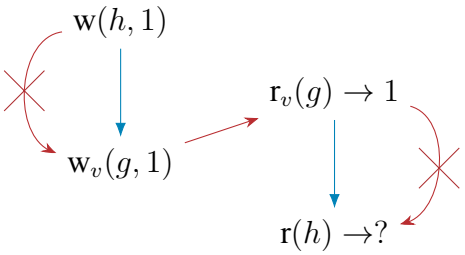
Для демонстрации того, что нам это дает, рассмотрим программу в таблице 2, которая очень похожа на другую программу в таблице 1.

Таблица 2: Volatile и plain чтение

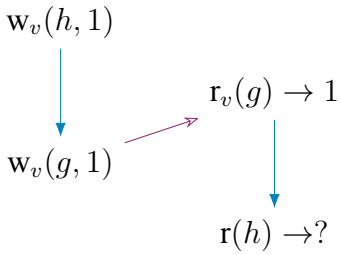
```
volatile int g
int h
```

`h = 1`	`int r1 = g`
`g = 1`	`int r2 = h`

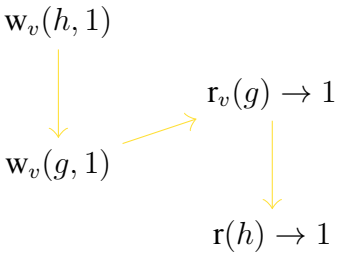
На рисунке 3 изображено одно конкретное исполнение, которое преобразуется в HB.



(a) Отсутвие необходимого SO



(b) Построение SW



(c) Построение HB

Рисунок 3: Разница гарантий

Как видно по части 3a одного SO недостаточно для того, чтобы сказать, что будет в $r2$. Поэтому строим его подпорядок в части 3b, а на его основе и HB 3c. Как становится видно, HB позволяет видеть значения не volatile переменных. Это очень полезное свойство, которое мы используем в следующей части.

1.3 Источник гарантий

Ранее мы говорили, мол что будет записано такое-то значение в переменную. Но откуда у нас гаранития этого? Хороший вопрос, поскольку до этого, в сущности семантика не имела веса и никакого влияния на то, что происходит в реальности.

Примем функцию $W(r)$ за функцию записи, которая для $\forall r \in A$ дает действие записи увиденное в E. И вот формальное определение видимости для SO:

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

Аналогично для HB:

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

[4, Глава 7.3]

Первую часть, упрощенно можно понимать, как чтение видит любую запись, которая не упорядочена после него. Это также допускает гонки.

Вторая же часть, уплощает длинные цепи, и можно понимать, это как видим самую последнюю запись.

Всё это, обычно это упрощают до утверждения, что volatile гарантирует видимость последний записи.

1.4 Требование причинности

На самом деле, HB недостаточно для корректной JMM. Поэтому вводится еще и требования причинности. Не будем раскрывать все формальности, а возьмем лишь следствие. Корректным исполнение не будет, если в нем нарушена причинность или существует каузальный цикл. Иначе говоря, значения не могут возникнуть из воздуха.

Рассмотрим пример представленный на таблице 3.

Таблица 3: Цикл причинности

```
int x = 0, y = 0
```

`r1 = x`	`r2 = y`
`if(r1 == 1)`	`if(r2 == 1)`
`  y = 1`	`  x = 1`

В ней $(r1, r2)$ всегда равны $(0, 0)$.

Разберем причину этого, на примере $(1, 1)$.

Разделяемые переменные здесь x, y, изначально равняясь нулю они могут принять единицу только в том, случае если каким-то образом будет проигнорировано условие. И как видно на рисунке 4, если такое случится, образуется цикл.

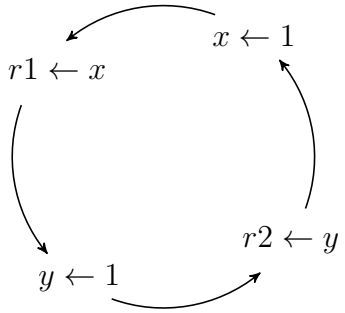


Рисунок 4: Цикличная зависимость

1.5 Иммутабельность

Немного особняком стоят final переменные, также как и с требованием причинности, рассмотрим обобщенные гарантии. Если поле класса является final, и во время конструирования объекта ссылка не утекла, то запись в это поле HB по отношению к его чтению.

Такая гарантия позволяет достаточно сильно упростить многие структуры и вообще использование многопточности. Именно поэтому, многие классы в Java иммутабельны.

2 Формализация примитивов

2.1 Mutex

Начнем с построения ключевого примитива синхронизации — мьютекса, как мы помним его название образовано от слов mutual exclusion(взаимное исключение). Ключевой он потому, что его построение позволяет абстрагироваться до уровня блокирующих алгоритмов и уже сделать например семафор, а там блокирующую очередь и прочие алгоритмы.

```
class Mutex {
    private final AtomicBoolean bool = new
       AtomicBoolean();

    public void lock() {
        while (!bool.compareAndSet(false, true)) ;
    }
```

```
    public void unlock() {
        bool.set(false);
    }
}
```

Легко заметить, что этот код обеспечивает взаимное исключение, поскольку, как только один поток сможет записать в переменную `bool` `true`, никакой другой не может выйти из внутреннего цикла while, пока не будет записано `false` в `unlock()` или же JVM принудительно убьет поток.

Алгоритм также обеспечивает прогресс: если `bool` равна `false`, то какой-то поток выйдет из него, благодаря гарантиям CAS, а если же она равна `true`, какой-то поток находится в области между вызовом `lock()` и вызовом `unlock()`, конечно при условии, что пользователь нашего объекта поместил вызов `unlock()` в секцию `finally`, а также критическая секция рано или поздно завершится, тогда в конечном итоге поток доходит до `unlock()` и впускает следующий поток.

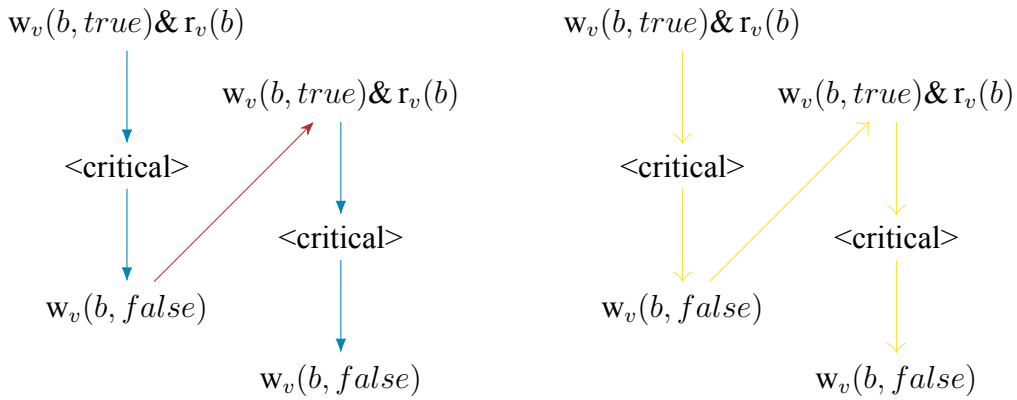
Рассмотрим таблицу 4.

Таблица 4: Пример использования объекта

`Mutex m = new Mutex();`	

`m.lock();`
`try{`
`  <critical>`
`}finally{`
`  m.unlock();`
`}`
	`m.lock();`
	`try{`
	`  <critical>`
	`}finally{`
	`  m.unlock();`
	`}`

А также её исполнения на рисунке 5.



(a) SO (b) HBO

Рисунок 5: Исполнения таблицы 4

Из гарантии взаимного исключения следует, что SO будет существовать вне зависимости от порядка владения исключительным доступом к критической секции, а также количества потоков. И поскольку иных исполнений JMM не позволяет, благодаря свойству транзитивности, можно заключить, что $unlock() \xrightarrow{hb} lock()$.

Мы получили индентичный по гарантиям с intrinsic lock объект.

2.2 Коробка и acquire/release

Чуть усложним условия и скажем, что у нас есть некоторая коробка, которая хранит в себе что-то, а также имеет возможность закрываться. Ну например мы не хотим, что бы в процессе передачи кто-то изменил или просмотрел содержимое. А также очевидно, нам бы хотелось гарантию, что получатель сможет корректно получить значение.

```
class Box<T> {
    private T t;
    private volatile boolean closed;

    public void set(T t) {//acquire
        if (!closed) this.t = t;
    }
```

```
    public T get() {       //release
        return closed ? null : t;
    }

    public void close() { closed = true; }
    public void open() { closed = false; }
    public boolean isClose() { return closed; }
}
```

Рассмотрим типичное её использование в таблице 5.

Таблица 5: Использование коробки

`Box<Integer> box = new Box<>();`	

`b.set(42);`
`b.close();`
	`if(b.isClose()) {`
	`  b.open();`
	`  int r = b.get();`
	`}`

Для того, чтобы понять, что будет в r, нам необходимо составить исполнение, которое видно на рисунке 6.

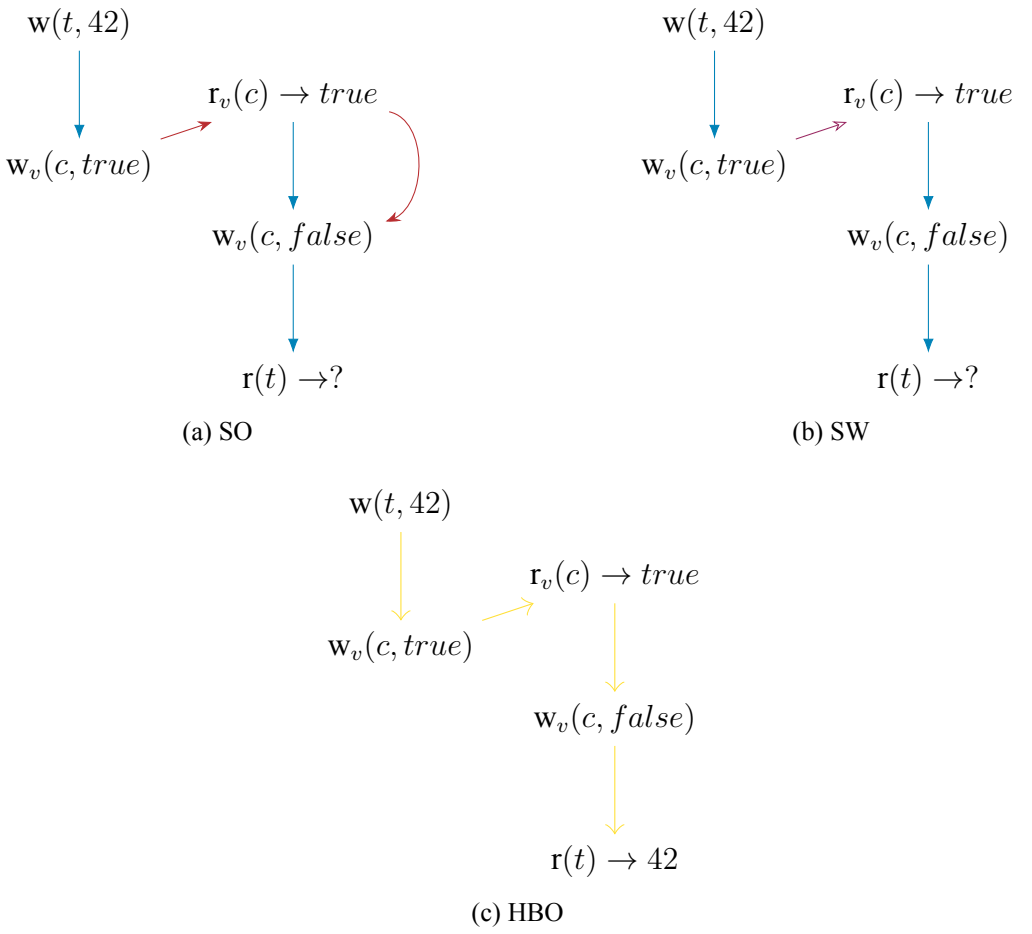


Рисунок 6: Исполнения таблицы 5

К сожалению, если и существует одно исполнение, не факт, что не существует другого с иными результатами. Потому, поставим некоторые гипотезы, попробов найти такие исполнения.

1. Возможно ли $\mathrm{r}_v(c) \to false$?

   Да, однако в таком случае ветка будет иной и запись в r вовсе не случится. Это не наш случай.

2. Могло ли $\mathrm{r}(t)$ быть раньше $\mathrm{w}(t, 42)$? И соовтветсвенно прочитать null.

   Нет, поскольку в таком случае мы нарушим PO, следовательно HB могут быть только в таком порядке, а значит и прочитать только 42.

Делаем вывод о том, что $set(t) \xrightarrow{\text{hb}} get()$. Можно сформулировать это свойство чуть иначе, через приобретение(Acquire) и освобождение(Release). Это второе выведенное нам высокуровневое свойство, вместе с мьютексом, позволяет строить корректные системы любой сложности. На самом деле в java таковые уже реализованы и по сути остается их только использовать.

Заключение

В результате была изучена возможность доказательств корректности любых многопоточных программ в рамках JMM. Также была формально разобрана основная ее часть. Вместе с тем, стали понятны приницпы и гарантии, на более высоком уровне. В качестве вдохновения были использованы некоторые статьи.[8; 9]

Список литературы

1. *Callon R.* The Twelve Networking Truths. — 1996. — Апр. — DOI: 10.17487/RFC1925.
2. *Goetz B.* Java Concurrency in Practice. — Addison-Wesley Professional, 2006. — ISBN 0321349601.
3. *Gosling J.* The Java Language Specification. — 2023. — URL: https://docs.oracle.com/javase/specs/jls/se21/html/index.html.
4. JSR-133: Java™ Memory Model and Thread Specification / Oracle. — 2004. — URL: https://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf.
5. *Lamport L.* How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs // IEEE Transactions on computers. — 1979. — Сент. — Т. C—28, № 9. — С. 690—691. — DOI: 10.1109/TC.1979.1675439.
6. *Liu L.* A volatile-by-default JVM for server applications // Proceedings of the ACM on Programming Languages. — 2017. — Окт. — Т. 1, № 49. — С. 1—25. — DOI: 10.1145/3133873.
7. *Paul*. A Beautiful Race Condition. — 06.2009. — URL: https://mailinator.blogspot.com/2009/06/beautiful-race-condition.html.

8. *Shipilëv A.* Java Memory Model Pragmatics. — 2014. — URL: https://shipilev.net/blog/2014/jmm-pragmatics.
9. *Shipilëv A.* Close Encounters of The Java Memory Model Kind. — 2016. — URL: https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/.
10. *Wong H.* Measuring Reorder Buffer Capacity. — 05.2013. — URL: https://blog.stuffedcow.net/2013/05/measuring-rob-capacity/.
11. *Zhou H.* Hunting Garbage Collection Related Concurrency Bugs through Critical Condition Restoration // FEAST'20: Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation. — 2020. — Нояб. — С. 17—22. — DOI: 10.1145/3411502.3418426.

A Использованные нотации

В работе использована нотация представленная в таблице 6.

Таблица 6: Пример нотации

`int i, j;`	
Thread 1	Thread 2
`i = 1;`	`j = 2;`

В ней, первая строка означает то, что произойдет строго до всех действий в тредах. Вторая, обозначает названия отрезков кода. Сами отрезки начинают исполнение одновременно и продолжают конкурентно.

Также ее дополняет другая нотация, легенда которой представленна на рисунке 7.

$\mathrm{r}_v(x)$ — $\mathrm{read}_{volatile}(x)$

$\mathrm{w}_v(x, n)$ — $\mathrm{write}_{volatile}(x, n)$

$\mathrm{l}(v)$ — $\mathrm{lock}(v)$

$\mathrm{u}(v)$ — $\mathrm{unlock}(v)$

Рисунок 7: Легенда