Атомарность

Атомарные операции - операции, выполняющиеся как единое целое, либо не выполняющиеся вообще. Атомарность операций имеет особое значение в многопоточной среде, так как доступ к разделяемым ресурсам должен быть обязательно атомарным.

Операция в общей области памяти называется **атомарной**, если она завершается в один шаг относительно других потоков, имеющих доступ к этой памяти.

Во время выполнения такой операции над переменной, ни один поток не может наблюдать изменение наполовину завершенным. Атомарная загрузка гарантирует, что переменная будет загружена целиком в один момент времени [6].

Каждая операция считается неатомарной до тех пор, пока другое не будет **явно** указано производителем компилятора или аппаратной платформы.

В общем случае:

* Не все ассемблерные инструкции являются атомарными (операция mov атомарна в том случае, когда операнд в памяти выровнен, и не атомарна в противном случае. То есть, атомарность гарантируется только в случае, когда 32-битное целое число находится по адресу, который делится на 4);
* Операция с памятью может быть неатомарной даже на одноядерном процессоре (выполняется за несколько инструкций);
* Одна инструкция процессора на некоторых платформах также может быть неатомарной [6];

**Реализация атомарности в Java**

Атомарные операции в Java - это операции, которые не могут быть прерваны планировщиком потоков.

В Java атомарными операциями является чтение и запись простых типов за исключением long и double.

**Инкремент и декремент - это** **не атомарные операции** потому, что по своей природе они составные. Сначала происходит считывание, увеличение/уменшение и далее запись значения.

До версии JDK 5.0 на языке Java нельзя было создавать wait-free и неблокирующие алгоритмы без использования собственного кода. Сейчас атомарные классы реализованы в пакете java.util.concurrent.atomic . Все атомарные классы переменных имеют базовый элемент compare-and-set.

В пакет java.util.concurrent.atomic входят такие классы атомарных переменных:

* AtomicBoolean;
* AtomicInteger;
* AtomicIntegerArray;
* AtomicLong;
* AtomicLongArray;
* AtomicMarkableReference;
* AtomicReference;
* AtomicReferenceArray;
* AtomicStampedReference;

Классы атомарных переменных можно рассматривать как обобщение volatile переменных, если расширить понятие изменяемых переменных до переменных с поддержкой атомарных обновлений методом compare-and-set. Чтение и запись атомарных переменных имеет такую же семантику памяти как доступ к чтению и записи изменяемых переменных.

Модели памяти

В мультипроцессорных системах процессор может иметь один или несколько уровней кэш-памяти, которые увеличивают производительность за счет ускорения доступа к данным (данные находятся ближе к процессору) и уменьшения трафика через общую шину памяти (многие операции с памятью осуществляются за счет кэша).

Кэш-память может чрезвычайно повысить производительность, но представляет целый ряд новых проблем.

**Проблема видимости значений**

Например, что произойдет если два процессора запрашивают одну и ту же область памяти одновременно? При каких условиях они получат одно и то же значение?

Даже если операция является атомарной, значение переменной может хранится в кэше ядра, и быть не видным другому потоку.

У разных процессоров (потоков) могут быть одновременно разные значения одной и той же переменной.

На аппаратном уровне (на уровне процессора) эта проблема решается с помощью таких моделей памяти:

* Сильная (strong) модель памяти. Все процессоры видят одно и то же значение в одной области памяти в один и тот же момент времени;
* Слабая (weaker) модель памяти. В этом случае кэш-память сбрасывается или делается недействительной с помощью специальных инструкций, называемых memory barriers. Использование memory barriers позволяет другим процессорам видеть изменения, сделанные текущим процессором.

Memory barriers обычно вызываются когда осуществляется блокировка или разблокировка. Эти инструкции недоступны в высокоуровневом языке программирования.

Последние тенденции в разработке процессоров идут в сторону слабой модели, поскольку она обеспечивает большую масштабируемость и производительность [7], [9].

**Проблема порядка выполнения**

Описанная выше проблема видимости усугубляется тем, что на современных компьютерах код ради скорости выполняется не в том порядке, в котором написан. Перестановка выполняется компилятором, процессором и подсистемой памяти.

Перестановка машинных команд не нарушает семантики программы на высокоуровневом языке, но влияет на взаимодействие потоков

Например, в программе поток записывает значение в поле А, потом в поле Б. При этом значение поля Б не зависит от поля А. Компилятор может переставить операции записи и значение Б попадет в память раньше А.

**Модель памяти Java**

Модель памяти Java описывает какое поведение является правильным в многопоточном коде и как потоки должны взаимодействовать через общую память. Она описывает отношения между переменными в программе и низкоуровневым чтением и записью этих переменных на реальных компьютерных системах.

Модель памяти гарантирует, что корректно синхронизированная программа будет выполняться правильно на любых процессорах.

Модель памяти Java обеспечивает:

* Атомарность чтения и записи простых типов;
* Частичную упорядоченность операций с памятью (read field, write field);
* Частичную упорядоченность для потоковых операций (lock, unlock, start, join) ;

Решение проблем **видимости** и **порядка выполнения** в Java достигается с помощью установления отношения **happens before**.

Если действия находятся в отношении happens before, то гарантируется что первое действие будет выполнено раньше и его результат будет виден второму.

Модель памяти Java обеспечивает такие правила **happens before** (список не полный):

* В рамках одного потока действия будут выполнятся в том порядке, как они описаны в программе. Действие, описанное раньше, **произойдет до** описанного позже;
* Освобождение монитора **произойдет до** любого последующего захвата того же монитора;
* Запись volatile поля **произойдет до** любого последующего чтения того же поля;
* Выполнение метода start() потока **произойдет до** любого действия в запущенном потоке;
* Все действия в первом потоке **произойдут до** того как любой другой поток успешно выйдет из join() для первого потока [7], [9];

**Модификатор volatile**

Переменные volatile решают проблемы, которые описаны выше:

* **Проблема видимости**. Модификатор volatile гарантирует, что любой поток заново прочитает volatile-поле из основной памяти вместо того, чтобы использовать кэшированное значение. Для этого содержимое кэша процессора объявляется недействительным. При записи volatile-полей гарантируется, что значение будет записано в основную память, а значит все потоки будут видеть одно и то же значение. Это достигается сбрасыванием кэша процессора в основную память;
* **Проблема порядка выполнения**. В новой модели памяти (начиная с JDK 5.0) порядок чтения/записи volatile-полей не может быть изменен. Это касается как порядка чтения/записи нескольких volatile-полей, так и чтения/записи volatile-полей вместе с обычными. Запись в volatile-поле имеет тот же эффект для памяти, что и освобождение монитора (англ. monitor release), а чтение - тот же, что и захват (англ. monitor acquire). То есть, доступ к volatile-полю устанавливает отношение **happens before**, что и обеспечивает видимость изменений для всех потоков [7], [9];

Bстроенные средства синхронизации

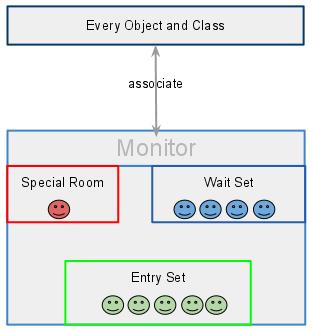
К встроенным в язык средствам синхронизации относятся:

* Мониторы (monitors);
* Методы объекта wait(), notify() , notifyAll();
* Метод Thread.join() (рассмотрен в заданиях 24, 25);

Другие средства синхронизации, такие как семафоры, мютексы, критические секции, неблокирующие алгоритмы реализованы в библиотеках типа java.util.concurrent или должны быть реализованы самостоятельно.

**Концепция монитора**

Монитор можно объяснить на следующей аналогии. Пускай есть здание, которое содержит особое помещение. Особое помещение содержит некоторый код и данные и может быть занято только одним клиентом (потоком) в данный момент времени.



Если клиент (поток) хочет занять особое помещение, он должен пройти в прихожую (Entry Set). Планировщик выбирает клиента (поток) из прихожей. Если данный клиент (поток) должен быть приостановлен, то он отправляется в комнату ожидания (Wait Set) [3].

Монитор – это средство обеспечения контроля за доступом к ресурсу.

Монитор реализует следующие свойства:

* **Атомарности** (см. выше);
* **Взаимного исключения** (mutual exclusion). Только один поток может владеть монитором. Синхронизация на мониторе означает, что как только один поток входит в synchronized-блок, защищённый монитором, никакой другой поток не может войти в блок, защищённый этим монитором пока первый поток не выйдет из synchronized-блока [5].

**Реализация монитора в Java**

В Java каждый объект и класс имеет неявный монитор.

Для того, чтобы войти в монитор объекта (захватить монитор) используется ключевое слово **synchronized**.

Для того чтобы выйти из монитора (отпустить монитор) и тем самым передать управление объектом другому потоку, владелец монитора должен всего лишь вернуться из синхронизованного метода или блока кода.

**Синхронизированный метод и блок**

В этом случае (синхронизация блока) захватывается монитор у объекта sync:

Object sync = **new** Object();

...

**synchronized**(sync){

}

Синхронизированным может быть также весь метод:

**public synchronized void** someMethod(){

// code

}

**Методы объекта wait(), notify() , notifyAll();**

В Java есть механизм общения между потоками, основанный на методах wait(), notify() и notifyAll(). Эти методы реализованы как final- методы класса Object, так что они имеются в любом Java-классе.

Все эти методы должны вызываться только из синхронизованных блоков (методов), поскольку сами они не синхронизированы.

|  |  |
| --- | --- |
| **Метод** | **Результат вызова** |
| wait() | Текущий поток отдает управление и переходит в режим ожидания до тех пор, пока другой поток не вызовет метод notify() с тем же объектом |
| notify() | Выводит из состояния ожидания первый из потоков, вызвавших wait() с данным объектом |
| notifyAll() | Выводит из состояния ожидания все потоки, вызвавшие wait() с данным объектом |

Проблемы многопоточных программ

**Состояние гонки (race conditions)**

Состояние гонки (race conditions) – это ошибочное поведение программы, которое возникает из-за того, что порядок выполнения частей кода нарушен. То есть, проблема возникает из-за того, что мы не можем знать в каком порядке будут выполнятся части кода.

Частным случаем состояния гонки, является гонка данных (data race). Гонка данных возникает, когда два потока одновременно используют одну и ту же область памяти (ресурс) и хотя бы один из потоков изменяет ее.

Таким образом, в многопоточной среде есть такие проблемы:

* Мы не можем знать в каком порядке выполняются потоки;
* Первый поток может не увидеть изменений, сделанных вторым потоком;
* Первый поток может увидеть не все изменения, сделанные вторым потоком;

Описанные проблемы приводят к непредсказуемому результату работы программы.

Пример:

if (x == 5) // The "Check"

{

y = x \* 2; // The "Act"

// If another thread changed x in between "if (x == 5)" and "y = x \* 2"  
 // above,

// y will not be equal to 10.

}

**Взаимная блокировка (deadlock)**

Ситуация в многопоточной среде, при которой несколько потоков находятся в состоянии бесконечного ожидания ресурсов, занятых самими этими потоками.

Пример взаимной блокировки:

**public class** Main {  
 **public static void** main(String[] args) {  
 *//These are the two resource objects we'll try to get locks for* **final** Object resource1 = **"resource1"**;  
 **final** Object resource2 = **"resource2"**;  
  
 *//Here's the first thread. It tries to lock resource1 then resource2* Thread t1 = **new** Thread() {  
 **public void** run() {  
 **synchronized**(resource1) {  
 System.***out***.println(**"Thread 1: locked "** + resource1);  
  
 *//Pause for a bit, simulating some file I/O or something.  
 //We're trying to force deadlock to happen here...* **try** {  
 Thread.*sleep*(50);  
 } **catch** (InterruptedException e) {}  
  
 *//Now wait 'till we can get a lock on resource 2* **synchronized**(resource2) {  
 System.***out***.println(**"Thread 1: locked "** + resource2);  
 }  
 }  
 }  
 };  
  
 *//Here's the second thread. It tries to lock resource2 then resource1* Thread t2 = **new** Thread() {  
 **public void** run(){  
 **synchronized**(resource2) {  
 System.***out***.println(**"Thread 2: locked "** + resource2);  
  
 *//Then it pauses, for the same reason as the first thread  
 //does* **try** {  
 Thread.*sleep*(50);  
 } **catch** (InterruptedException e) {}  
  
 *//Thread 1 locked resource1, and won't release it till it  
 //gets a lock on resource2.  
 //This thread holds the lock on resource2, and won't  
 //release it till it gets resource1.  
 //We're at an impasse. Neither thread can run, and the  
 //program freezes up.* **synchronized**(resource1) {  
 System.***out***.println(**"Thread 2: locked "** + resource1);  
 }  
 }  
 }  
 };  
  
 *//Start the two threads. If all goes as planned, deadlock will occur,  
 //and the program will never exit.* t1.start();  
 t2.start();  
 }  
}

Чтобы избежать взаимной блокировки нужно:

* По возможности не используйте более одной блокировки в блоке или методе;
* Всегда соблюдайте один и тот же порядок блокировки;
* Всегда отпускайте блокировки в обратном захвату порядке, т.е. руководствуйтесь логикой "первый захвачен – последний отпущен";
* По возможности используйте неблокирующие алгоритмы (wait-free, lock-free, obstruction-free) [2], [8];
* По возможности используйте современные средства синхронизации из библиотеки java.util.concurrent;

**Чем sleep() отличаетсья от wait()**

Основные отличия такие:

* wait() освобождает монитор объекта на котором он вызван, что позволяет продолжить работу другим потокам ожидающим захватить ту же самую блокировку. sleep() не освобождает никаких блокировок;
* wait() используется для синхронизации доступа к одному и тому же объекту (межпотоковая синхронизация). sleep() служит просто для приостановки потока;
* wait() – это метод объекта и может быть вызван только для объекта владеющего блокировкой, в противном случае выкинется исключение IllegalMonitorStateException. sleep() – статический метод класса Thread, останавливает поток для которого вызван;
* wait() заставляет ждать текущий поток, пока другой поток не вызовет notify(), notifyAll() для того же объекта. sleep() останавливает поток пока не истечет время или поток не будет прерван методом interrupt();

Singleton Design Pattern

**Название и классификация**

Singleton - паттерн, порождающий объекты.

**Назначение**

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа (доступ к своему экземпляру).

**Мотивация**

Для некоторых классов важно, чтобы существовал только один экземпляр. Например, должны быть только одна файловая система и единственный оконный менеджер. Как гарантировать, что у класса есть единственный экземпляр и что этот экземпляр легко доступен? Глобальная переменная дает доступ к объекту, но не запрещает инстанциировать класс в нескольких экземплярах.

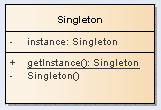
Более удачное решение - сам класс контролирует то, что у него есть только один экземпляр, может запретить создание дополнительных экземпляров, перехватывая запросы на создание новых объектов.

**Применимость**

Используйте паттерн одиночка, когда:

* Должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам;
* Единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода;

**Реализация**



Поле "instance" является приватным статическим, метод "getInstance()" публичный статический метод. Конструктор приватный или защищенный, если требуется наследование.

Реализации в Java см. в [11]

1. <http://www.skipy.ru/technics/synchronization.html>
2. <https://www.ibm.com/developerworks/ru/library/j-jtp04186/>
3. <http://www.programcreek.com/2011/12/monitors-java-synchronization-mechanism/>
4. Goetz B. - Java Concurrency in Practice
5. <https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D0%B8%D0%B7%D0%BC_%D0%B2_Java>
6. <http://habrahabr.ru/post/244881/>
7. <https://ru.wikipedia.org/w/index.php?title=%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8_Java&oldid=72817179>
8. <https://www.ibm.com/developerworks/ru/library/j-jtp11234/>
9. <https://www.cs.umd.edu/users/pugh/java/memoryModel/jsr-133-faq.html>
10. Design Patterns. Elements of Reusable Object-Oriented Software, Erich Gamma, Ralph Johnson, Richard Helm, John Vlissides
11. <http://habrahabr.ru/post/27108/>