***13. Атомарные операции. Толстые и тонкие мониторы.***

Атомарная операция — операция, выполняющаяся как единое целое, либо не выполняющаяся вовсе. Атомарность операций имеет особое значение в многопроцессорных компьютерах (и многозадачных операционных системах), так как доступ к разделяемым ресурсам должен быть обязательно атомарным.

Атомарная операция открыта влиянию только одного потока.

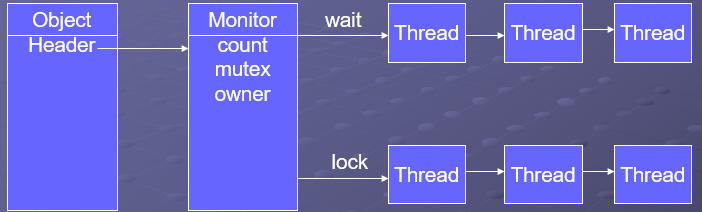
Атомарность бывает аппаратной (когда непрерывность обеспечивается аппаратурой) и программной, когда используются специальные средства межпрограммного взаимодействия: мьютекс, семафор. По своей сути программные средства обеспечения атомарности представляют собой два этапа: блокировка ресурса и выполнение самой операции. Блокировка представляет собой атомарную операцию, которая либо успешна, либо возвращает сообщение о занятости.

[CAS-Increment](https://dzone.com/articles/how-cas-compare-and-swap-java)

* CAS(addr, old, new) (Atomic Compare and Swap)
  + За одну операцию сравнивает текущее значение со старым и, если совпадает, меняет на новое
  + Если не совпадает, возвращает ошибку
* Пример: Atomic Increment
  + while (!CAS(&x, x, x + 1));
* Write Barrier – сохраняет изменения из кэша процессора в память, ограничивает переупорядочивание записей в будущее
* Read Barrier – обновляет кэш процессора из памяти, ограничивает переупорядочивание чтений в прошлое

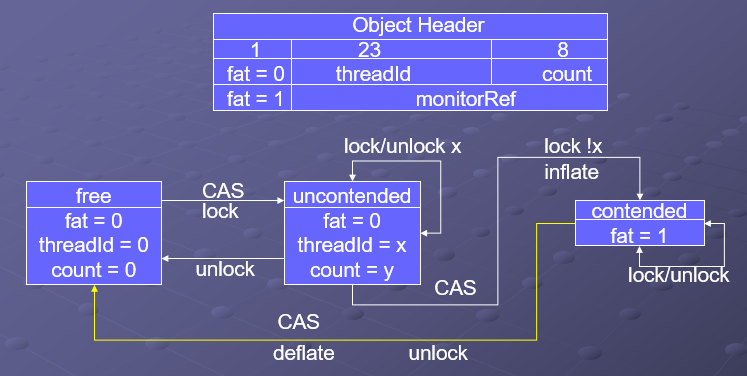
[barriers1](http://stackoverflow.com/questions/1787450/how-do-i-understand-read-memory-barriers-and-volatile)- тут норм про барьеры [barriers2](http://stackoverflow.com/questions/18485791/what-are-read-barriers-and-write-barriers-in-synchronized-block) – тут пример барьера

*Толстые Мониторы (fat locks)*

**

* Заголовок объекта содержит ссылку на структуру монитора либо объект отображается при помощи хеш-таблицы
* Монитор
  + нить владельца
  + счетчик рекурсии
  + синхронизационный ресурс ОS
  + список нитей, ждущих освобождения монитора
  + список нитей, ждущих нотификации

*Тонкие мониторы (thin locks)*



* При чтении используется Read Barrier
* Сдувание монитора редко реализуется из-за сложности

When threads in a process share and update the same data, their activities must be synchronized to avoid errors. In Java, this is done with the synchronized keyword, or with wait and notify. Synchronization is achieved by the use of locks, each of which is associated with an object by the JVM. For a thread to work on an object, it must have control over the lock associated with it, it must “hold” the lock. Only one thread can hold a lock at a time. If a thread tries to take a lock that is already held by another thread, then it must wait until the lock is released. When this happens, there is so called “contention” for the lock.

There are four different kinds of locks:

Fat locks: A fat lock is a lock with a history of contention (several threads trying to take the lock simultaneously), or a lock that has been waited on (for notification).

Thin locks: A thin lock is a lock that does not have any contention.

Recursive locks: A recursive lock is a lock that has been taken by a thread several times without having been released.

Lazy locks: A lazy lock is a lock that is not released when a critical section is exited. Once a lazy lock is acquired by a thread, other threads that try to acquire the lock have to ensure that the lock is, or can be, released. Lazy locks are used by default in Oracle JRockit JVM 27.6. In older releases, lazy locks are only used if you have started the JVM with the -XXlazyUnlocking option.

A thin lock can be inflated to a fat lock and a fat lock can be deflated to a thin lock. The JRockit JVM uses a complex set of heuristics to determine when to inflate a thin lock to a fat lock and when to deflate a fat lock to a thin lock.

[locks-javadoc](https://docs.oracle.com/cd/E13188_01/jrockit/docs142/usingJRA/applocks.html) – тут понятно написано об этом