# 1. JVM, Threads, and Monitors

See also:

° *http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpec-TOC.doc.html*

I have copied material from there. Thanks to Tim Lindholm Frank Yellin.

## 1.1. Overview

- Java supports multithreading on language level
- Multithreading requires synchronization
- Much of the language support is centered around synchronization
- Synchronization is done with the support of monitors
- The JVM dies, after the last not daemon thread termitates
- This means, starting a second program requires the loading of all 'helper' classes, like Object, String etc.
- This is not efficent, but increases security

## 1.2. Example 1: CPU Slot Distribution

- A example to test the distribution of cpu slot.

```
 1      public class Thread_1 extends Thread {
 2       private int priority;
 3
 4       public Thread_1 (int priority) {
 5              this.priority = priority;
 6              setPriority(priority);
 7       }
 8       public void run () {
 9          while ( true )
10              System.out.println(priority);
11       }
12       public static void main (String args []) {
13          for ( int i = Thread.MIN_PRIORITY; i <= Thread.MAX_PRIORITY; i
```

```
14                    new Thread_1(i).start();
15            }
16         }
17       }
```

Source Code: °*Src/16/Thread_1.java*

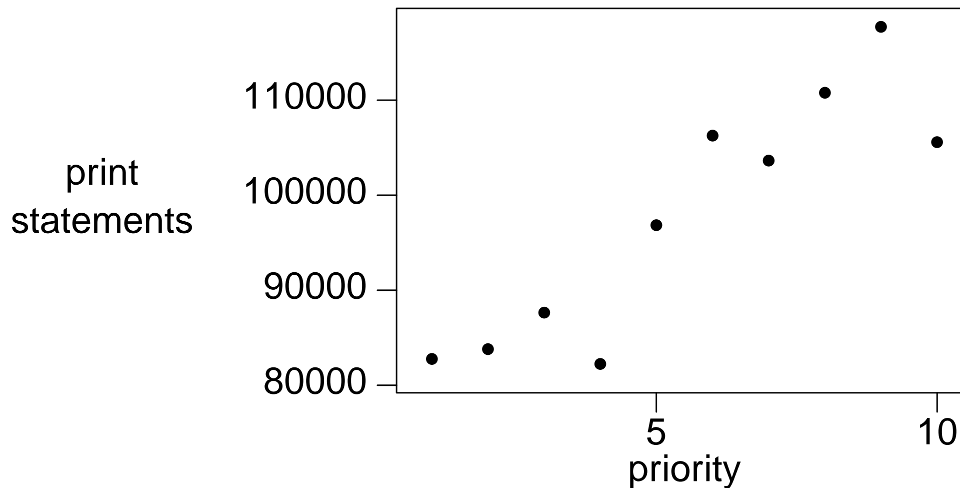
- Is this a ’valid’ approach?

## 1.3. Example 1: Result on a Single Processor Machine

- yps was used

```
$ uname -a   ## single processor
SunOS yps 5.8 Generic_111433-01 sun4u sparc SUNW,Ultra-5_10
$ java Thread_1 > `hostname`_1
$ wc -l `hostname`_1
  971883 yps_1
$ ls -l `hostname`_1
-rw-------   1 hpb      fac      2048815 Dec 14 09:33 yps_1
```

- A graph:

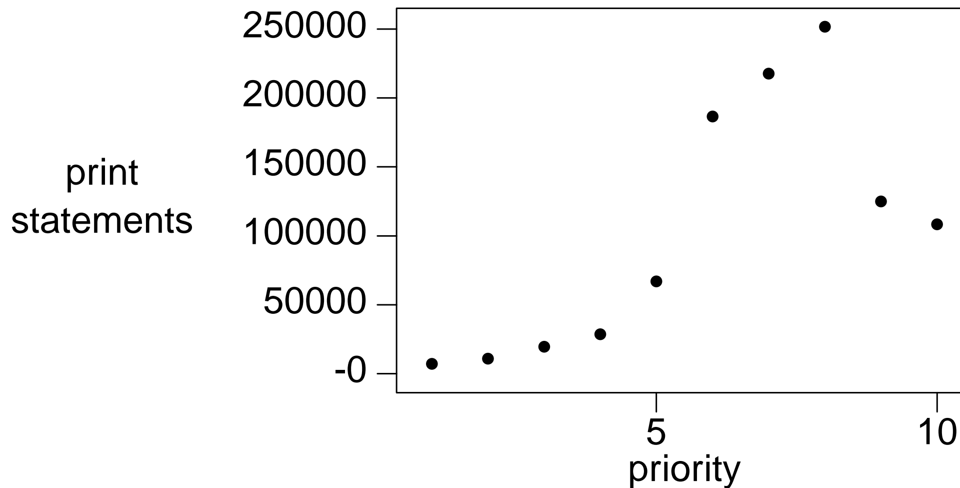


## 1.4. Example 1: Result on a Multi Processor Machine

- holly was used

```
$ uname -a    ## multi processor
SunOS holly 5.8 Generic_111433-01 sun4u sparc SUNW,Ultra-80
$ java Thread_1 > `hostname`_1
$ wc -l `hostname`_1
  984575 holly_1
$ ls -l `hostname`_1
-rw-------    1 hpb      fac      2073679 Dec 14 09:37 holly_1
```

- A graph:




## 1.5. Example 2: CPU Slot Distribution

- A second example to test the distribution of cpu slot.

```
 1
 2      public class Thread_2 extends Thread {
 3
 4       public void run () {
 5              for (int i = 1; i < 10000000; i ++ )    {
 6                      Math.atan(2.222);
 7              }
 8       }
 9
```

```
10      public static void main (String args []) {
11         int soMany = 1;
12         if ( args.length == 1 )
13             soMany = new Integer(args[0]).intValue();
14
15         for ( int i = 0; i < soMany; i ++ ) {
16                 new Thread_2().start();
17         }
18      }
19     }
```

Source Code: °*Src/16/Thread_2.java*

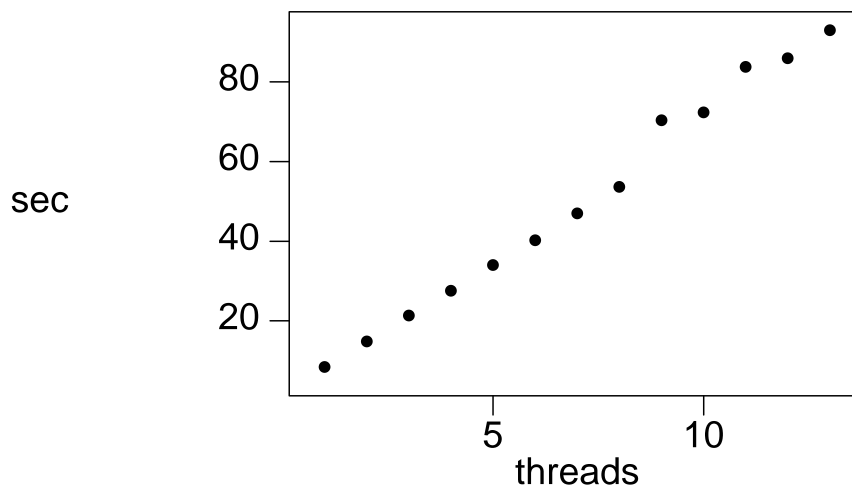
- Is this a 'valid' approach?

## 1.6. Example 2: Result on a Single Processor Machine

- yps was used

```
% uname -a
SunOS yps 5.8 Generic_111433-01 sun4u sparc SUNW,Ultra-5_10
```

- A graph:

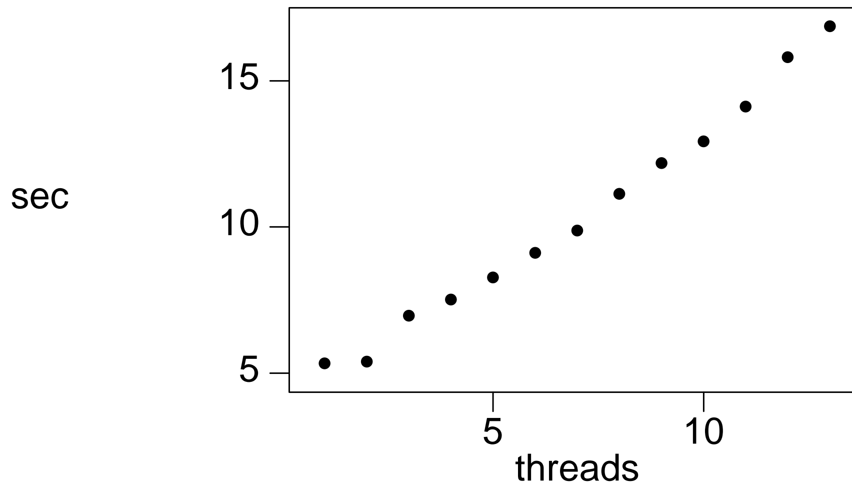


## 1.7. Example 2: Result on a Multi Processor Machine

- holly was used

```
$ uname -a    ## multi processor
SunOS holly 5.8 Generic_211433-01 sun4u sparc SUNW,Ultra-80
```

- A graph:




## 1.8. Example 3: Locking and Unlocking

- How often do we have to unlock?

```
 1
 2      public class Thread_3 extends Thread {
 3
 4       static Object o = new Object();
 5
 6       private String id;
 7
 8       public Thread_3 (String id) {
 9              this.id = id;
10       }
11
12       private int nF(int n)  {
13              try {
14                      sleep(1000);
```

```
15              } catch ( Exception e ) {}
16              synchronized ( o ) {
17                      System.out.println("---> goAndTry: " + id );
18                      int k = 0;
19                      if ( n == 3 && id == "first" )
20                              stop();
21                      if ( n > 0 )
22                              k = n * nF(n-1);
23                      else
24                              k = 1;
25                      System.out.println("<--- goAndTry: " + id );
26                      return k;
27              }
28      }
29      public void run () {
30              nF(5);
31      }
32
33      public static void main (String args []) {
34              new Thread_3("first").start();
35              new Thread_3("second").start();
36      }
37     }
```

Source Code: °*Src/16/Thread_3.java*

## 1.9. Example 4: Method Synchronization

- The use of method synchronization:

```
1       public class Thread_4 extends Thread    {
2
3           public synchronized void run () {
4               System.err.println("--> is in run()");
5               try {
6                       sleep(1000);
```

```
 7                } catch (  InterruptedException e ) { }
 8                System.err.println("<-- exit run");
 9            }
10
11            public static void main (String args []) {
12                Thread_4 immigrantSong = new Thread_4();
13                        // Led Zeppelin, Led Zeppelin III,
14                        // Released October/5/1970
15                        // http://www.led-zeppelin.com/disc-lz3.html
16                immigrantSong.start();
17                immigrantSong.run();
18            }
19        }
```

Source Code: °*Src/16/Thread_4.java*

## 1.10. Things which you know

- Priority.

  Threads with higher priority are executed in preference to threads with lower priority.

- daemon thread

  When code running in some thread creates a new Thread object, that new thread is initially marked as a daemon thread if and only if the creating thread is a daemon thread.

- deterministic behavior

  Java supports the coding of programs that, though concurrent, still exhibit deterministic behavior, by providing mechanisms for synchronizing the concurrent activity of threads.

- monitors

  To synchronize threads, Java uses monitors, which are a high-level mechanism for allowing only one thread at a time to execute a region of code protected by the monitor.

- locks

  The behavior of monitors is explained in terms of locks. There is a lock associated with each object.

- synchronized

  The synchronized statement performs two special actions relevant only to multithreaded operation:

  — After computing a reference to an object but before executing its body, it locks a lock associated with the object.

  — After execution of the body has completed, either normally or abruptly, it unlocks that same lock.

- control transfer

  The methods

  — wait,

  — notify, and

  — notifyAll

  of class Object support an efficient transfer of control from one thread to another.

- give up

  a thread can suspend itself using wait until such time as another thread awakens it using notify or notifyAll or the time was running out

- use cases

  Threads may have a

  — producer-consumer relationship (actively cooperating on a common goal) or a

  — mutual exclusion relationship (trying to avoid conflicts while sharing a common resource).

- concurrent access

  If two or more concurrent threads act on a shared variable, there is a possibility that the actions on the variable will produce timing-dependent results. This dependence on timing is inherent in concurrent programming and produces one of the

few places in Java where the result of a program is not determined solely by The Java Language Specification.

## 1.11. Green and Native Threads

- Green threads:

  Green threads are provided by the JVM, run at the user level, meaning that the JVM creates and schedules the threads itself. Therefore, the operating system kernel doesn't create or schedule them. Instead, the underlying OS sees the JVM only as one thread.

- Green threads problems:
  - — green threads cannot take advantage of a multiprocessor system.
  - — An IO-blocked thread can block the entire JVM.
- Native threads:

  Native threads are created and scheduled by the underlying operating system. The JVM creates the threads by calling OS-level APIs.

- Native threads benefits:
  - — Native threads can benefit from multiple processors.
  - — Performance can improve because an IO-blocked thread will no longer block the entire JVM. The block will only block the thread waiting for the IO resource.

— Downloads: °*http://java.sun.com/j2se/1.4/*

## 1.12. Synchronization

JMV:

The Java virtual machine provides explicit support for synchronization through its monitorenter and monitorexit instructions. For code written in the Java programming language, however, perhaps the most common form of synchronization is the synchronized method.

- synchonized method:

  A synchronized method is not normally implemented using

monitorenter and monitorexit. Rather, it is simply distinguished in the runtime constant pool by the

ACC_SYNCHRONIZED flag,

which is checked by the method invocation instructions. When invoking a method for which ACC_SYNCHRONIZED is set, the current thread acquires a monitor, invokes the method itself, and releases the monitor whether the method invocation completes normally or abruptly.

- synchonized block:
  - The program:

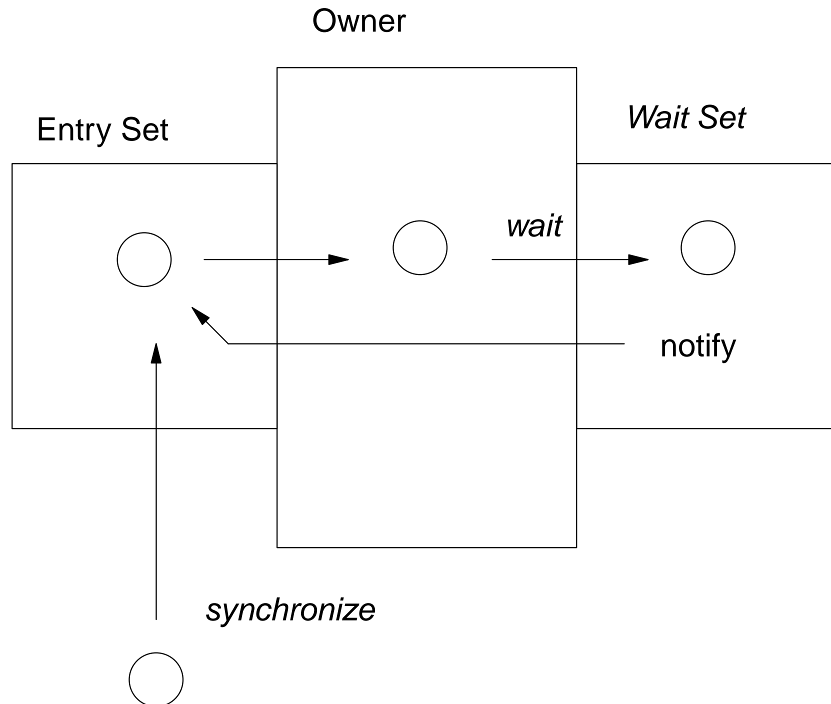
```
void onlyMe(Foo f) {
        synchronized(f) {
            doSomething();
        }
    }
```

is compiled to

```
Method void onlyMe(Foo)
   0    aload_1                 // Push f
   1    astore_2                // Store it in local variable 2
   2    aload_2                 // Push local variable 2 (f)
   3    monitorenter            // Enter the monitor associated with f
   4    aload_0                 // Holding the monitor, pass this and...
   5    invokevirtual #5        // ...call Example.doSomething()V
   8    aload_2                 // Push local variable 2 (f)
   9    monitorexit             // Exit the monitor associated with f
  10    return                  // Return normally
  11    aload_2                 // In case of any throw, end up here
  12    monitorexit             // Be sure to exit monitor...
  13    athrow                  // ...then rethrow the value to the invoker
Exception table:
        From    To      Target          Type
        4       8       11              any
```

## 1.13. Threads and Locks

- One picture *says* it all:




- object locks:

  Only one thread can lock one object. In the JVM, each object is logically associated with a monitor.

- class locks:

  Class locks are actually implemented as object locks. When tghe JVM loads a class file, it creates an off instance of java.lang.class.

- operations:

  A single thread issues a stream of

  — use,

  — assign,

  — lock, and

  — unlock

  operations as dictated by the semantics of the program it is

executing.

- JVM implementation

  The underlying Java virtual machine implementation is then required additionally to perform appropriate

— load,

— store,

— read, and

— write

operations Each of these operations is atomic.

- use action

  A use action (by a thread) transfers the contents of the thread's working copy of a variable to the thread's execution engine. This action is performed whenever a thread executes a virtual machine instruction that uses the value of a variable.

- assign action

  An assign action (by a thread) transfers a value from the thread's execution engine into the thread's working copy of a variable. This action is performed whenever a thread executes a virtual machine instruction that assigns to a variable.

- read action:

  A read action (by the main memory) transmits the contents of the master copy of a variable to a thread's working memory for use by a later load operation.

- load action

  A load action (by a thread) puts a value transmitted from main memory by a read action into the thread's working copy of a variable.

- store action:

  A store action (by a thread) transmits the contents of the thread's working copy of a variable to main memory for use by a later write operation.

- write action

  A write action (by the main memory) puts a value transmitted from the thread's working memory by a store action into the master copy of a variable in main memory.
- lock action:

  A lock action (by a thread tightly synchronized with main memory) causes a thread to acquire one claim on a particular lock.
- unlock action:

  An unlock action (by a thread tightly synchronized with main memory) causes a thread to release one claim on a particular lock.

## 1.14. Rules About Locks

- lock:

  Only one thread at a time is permitted to lay claim to a lock; moreover, a thread may acquire the same lock multiple times and does not relinquish ownership of it until a matching number of unlock operations have been performed.
- unlock:

  A thread is not permitted to unlock a lock it does not own.

## 1.15. Rules About the Interaction of Locks and Variables

- lock:

  Of a thread is to perform an unlock operation on any lock, it must first copy all assigned values in its working memory back out to main memory.
- unlock:

  A lock operation behaves as if it flushes all variables from the thread's working memory, after which the thread must either assign them itself or load copies anew from main memory.

## 1.16. Execution Order and Consistency

- constraints on the relationships among actions:
  - The actions performed by any one thread are totally ordered; that is, for any two actions performed by a thread, one action precedes the other.
  - The actions performed by the main memory for any one variable are totally ordered; that is, for any two actions performed by the main memory on the same variable, one action precedes the other.
  - The actions performed by the main memory for any one lock are totally ordered; that is, for any two actions performed by the main memory on the same lock, one action precedes the other.
  - It is not permitted for an action to follow itself.
- relationships between the actions of a thread and the actions of main memory
  - Each lock or unlock action is performed jointly by some thread and the main memory.
  - Each load action by a thread is uniquely paired with a read action by the main memory such that the load action follows the read action.
  - Each store action by a thread is uniquely paired with a write action by the main memory such that the write action follows the store action.

## 1.17. Possible Swap without Synchronization

- program:

```
class Sample {
        int a = 1, b = 2;
        void hither() {
            a = b;
        }
        void yon()
            b = a;
```

```
        }
}
```

- two threads;

  suppose that two threads are created and that one thread calls hither while the other thread calls yon. What is the required set of actions and what are the ordering constraints?
- hither goes first:

  Let us consider the thread that calls hither. According to the rules, this thread must perform a use of b followed by an assign of a. That is the bare minimum required to execute a call to the method hither.
- variable b:

  Now, the first operation on variable b by the thread cannot be use. But it may be

— assign or

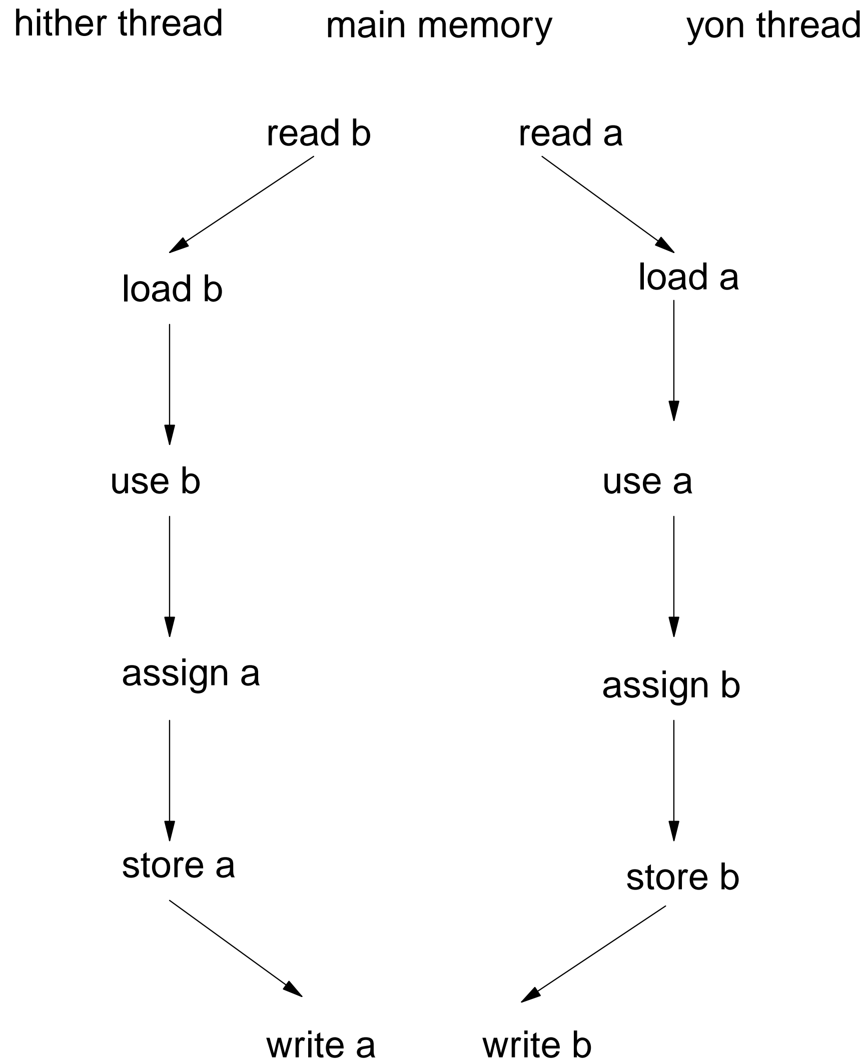
— load.

  An assign to b cannot occur because the program text does not call for such an assign operation, so a load of b is required. This load operation by the thread in turn requires a preceding read operation for b by the main memory.
- store:

  The thread may optionally store the value of a after the assign has occurred. If it does, then the store operation in turn requires a following write operation for a by the main memory.
- yon

  The situation for the thread that calls yon is similar, but with the roles of a and b exchanged.
- operations:

  The total set of operations may be pictured as follows:




•

• order of operations:

In what order may the operations by the main memory occur? The only constraint is that it is not possible both for the write of a to precede the read of a and for the write of b to precede the read of b, because the causality arrows in the diagram would form a loop so that an action would have to precede itself, which is not allowed.

• Possible order:

— Let ha and hb be the working copies of a and b for the hither thread.

— Let ya and yb be the working copies for the yon thread,

- — Let ma and mb be the master copies in main memory.
- — Initially ma=1 and mb=2.
- — Then the three possible orderings of operations and the resulting states are as follows:
- — write a → read a, read b → write b (then ha=2, hb=2, ma=2, mb=2, ya=2, yb=2)
- — read a → write a, write b → read b (then ha=1, hb=1, ma=1, mb=1, ya=1, yb=1)
- — read a → write a, read b → write b (then ha=2, hb=2, ma=2, mb=1, ya=1, yb=1)

- net result:

  Thus, the net result might be that, in main memory,
  - — b is copied into a, a is copied into b, or
  - — the values of a and b are swapped; moreover,
  - — the working copies of the variables might or might not agree.

  It would be incorrect, of course, to assume that any one of these outcomes is more likely than another. This is one place in which the behavior of a program is necessarily timing-dependent.

## 1.18. Possible Swap with Synchronization

- program:

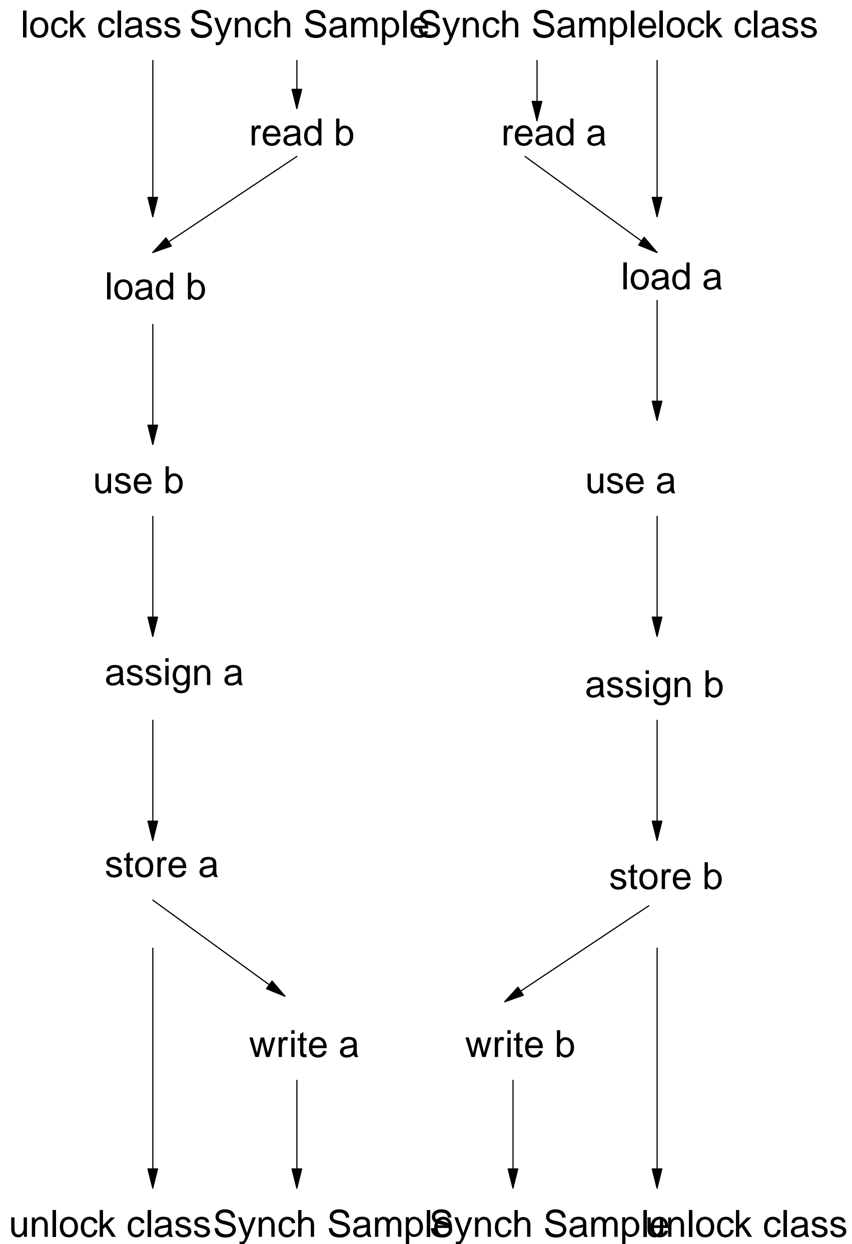
```
class SynchSample {
        int a = 1, b = 2;
        synchronized void hither() {
            a = b;
        }
        synchronized void yon() {
            b = a;
        }
}
```

- two threads;

  suppose that two threads are created and that one thread calls hither while the other thread calls yon. What is the required set of actions and what are the ordering constraints?

- operations:

  The total set of operations may be pictured as follows:




- order of operations:

  The lock and unlock operations provide constraints on the order of operations by the main memory; the lock operation

by one thread cannot occur between the lock and unlock operations of the other thread. Moreover, the unlock operations require that the store and write operations occur. It follows that only two sequences are possible:

— write a → read a, read b → write b (then ha=2, hb=2, ma=2, mb=2, ya=2, yb=2)

— read a → write a, write b → read b (then ha=1, hb=1, ma=1, mb=1, ya=1, yb=1)

- net result:
  While the resulting state is timing-dependent, it can be seen that the two threads will necessarily agree on the values of a and b.

1.19. Wait Sets Notification

- wait set:
  — Every object, in addition to having an associated lock, has an associated wait set, which is a set of threads. When an object is first created, its wait set is empty.
  — Wait sets are used by the methods wait, notify, and notifyAll of class Object. These methods also interact with the scheduling mechanism for threads.
- n lock/n unlocks
  Suppose that thread T has in fact performed N lock operations on the object that have not been matched by unlock operations on that same object.
- wait:
  The wait method then adds the current thread to the wait set for the object, disables the current thread for thread scheduling purposes, and performs N unlock operations on the object to relinquish the lock on it. Locks having been locked by thread T on objects other than the one T is to wait on are not relinquished
- release:

— Some other thread invokes the notify method for that object, and thread T happens to be the one arbitrarily chosen as the one to notify.
— Some other thread invokes the notifyAll method for that object.
— If the call by thread T to the wait method specified a time-out interval, then the specified amount of real time elapses.

The thread T is then removed from the wait set and reenabled for thread scheduling. It then locks the object again; once it has gained control of the lock, it performs N - 1 additional lock operations on that same object and then returns from the invocation of the wait method. Thus, on return from the wait method, the state of the object's lock is exactly as it was when the wait method was invoked

## 1.20. Implementation of the Thread Part in a JVM

- What are the critical things?
  — monitorenter and
  — monitorexit

## 1.21. monitorEnter

- Extract from *j2me_cldc/kvm/VmCommon/src/thread.c*:

```
enum MonitorStatusType monitorEnter(OBJECT object)
{
    MONITOR mid = OBJECT_MONITOR_OR_NULL(object);

    if (mid == NULL) {
        /* Object doesn't yet have a monitor.  Create one */

        START_TEMPORARY_ROOTS
            DECLARE_TEMPORARY_ROOT(OBJECT, tObject, object); /*  */
            mid = (MONITOR)callocObject(SIZEOF_MONITOR, GCT_MONITOR);
            object = tObject;
```

```
            mid->object = object;
            mid->hashCode = object->mhc.hashCode;
            SET_OBJECT_MONITOR(object, mid);
        END_TEMPORARY_ROOTS
    }

    if (mid->owner == NULL) {
        /* The monitor is unowned.  Make ourselves be the owner */
        mid->owner = CurrentThread;
        mid->depth = 1;
        return MonitorStatusOwn;
    } else if (mid->owner == CurrentThread) {
        /* We already own the monitor.  Just increase the entry count. *
        mid->depth++;
        return MonitorStatusOwn;
    } else {
        /* Add ourselves to the wait queue.  Indicate that when we are
         * woken, the monitor's depth should be set to 1 */
        CurrentThread->monitor_depth = 1;
        addMonitorWait(mid, CurrentThread);
        suspendThread();
        return MonitorStatusWaiting;
    }
}
```

## 1.22. monitorExit

- Extract from *j2me_cldc/kvm/VmCommon/src/thread.c*:

```
enum MonitorStatusType monitorExit(OBJECT object, char** exceptionName)
{
    MONITOR mid = OBJECT_MONITOR_OR_NULL(object);

    if (mid == NULL || mid->owner != CurrentThread) {
        *exceptionName = IllegalMonitorStateException;
        return MonitorStatusError;
    }
```

```
        *exceptionName = NULL;
        if (--mid->depth == 0) {
            /* Let someone else have the monitor */
            removeMonitorWait(mid);
            return MonitorStatusRelease;
        } else {
            return MonitorStatusOwn;
        }
    }
```

## 1.23. addThreadToQueue

Extract from *j2me_cldc/kvm/VmCommon/src/thread.c*:

```
static void addThreadToQueue(THREAD *queue,
                             THREAD thisThread,
                             queueWhere where)
{
    START_CRITICAL_SECTION
        if (*queue == NIL) {
            *queue = thisThread;
            thisThread->nextThread = thisThread;
        } else {
            /* Add thisThread after *queue, which always points
             * to the last element of the list */
            thisThread->nextThread = (*queue)->nextThread;
            (*queue)->nextThread = thisThread;
            if (where == AT_START) {
                /* We are already the first element of the queue */
                ;
            } else {
                /* make this thread be the last thread */
                *queue = thisThread;
            }
        }
    END_CRITICAL_SECTION
}
```

- What is the rational for:
  (START|END)_CRITICAL_SECTION?

The KVM/JVM is a single process.
(extract from j2me_cldc/kvm/VmCommon/h/thread.h)

```
 * An additional problem with asynchronous I/O is the possibility
 * of a race condition. To prevent a race condition corrupting the
 * RunnableThreads queue, which links asynchronous world outside
 a the VM with its internal rescheduling functions, all accesses
 * to this queue much be interlocked with a critical section.
 * Therefore, two new platform dependent functions have been
 * defined:
 *
 *     enterSystemCriticalSection()
 *
 * and
 *
 *     exitSystemCriticalSection()
 *
```