CS 332 (Operating Systems)	**Spring 2012**
Exam #1: Take-Home	*By: Holly French*

1. (a) When this is called, it is applied to the current thread. Thus, ThreadedKernel is where we usually manipulate threads, so I would call increasePriority() there, or else in a selfTest in the Priority Scheduler class and then I would call selfTest() in the Threaded Kernel. I would do this if I wanted to increase the priority of a current thread.

   (b) Same for decrease priority. It's a boolean method, so it is applied to the current thread.

   (c) Each thread has an associated threadState which includes both its base prioirty and current effective priority.

   Here's how it works:

   Say we have some shared resource. We want to prevent race conditions so we put a lock on that one resource. That means that if a thread wants to access a resource it must acquire the lock first. If it is unable to acquire the lock (in other words, another thread is using the resource), it must wait until the thread currently accessing the resource releases the lock.

   So say we have a thread (Thread 1) with priority 1, a very low priority. Thread 3 has priority 3, much higher than Thread 1, with priority 1. Let's say Threads 1, 2, and 3 have priority 1, 2, and 3, respectively, and Thread 1 is holding the resource and is thus locking it and preventing the highest priority thread (Thread 3) from using it. Thread 3 must wait for Thread 1 to release the lock. Because Thread 3 is waiting for the lock, it can't run, while only Threads 1 and 2 can. Thread 2 has the higher priority, so it will get selected. Thread 1 cannot run, so it will have to wait until Thread 2 finishes and Thread 3 releases the lock.

   One solution to this problem of extra waiting around is to donate priority. If Thread 1 had the same priority as Thread 3, it would not be blocked. So Thread 3 can share its priority to thread 1, and Thread 1's priority effectively becomes 3. After Thread 1 runs, then priorities can be restored back to the original.

   Nachos has a priority queue, which has a boolean value for transferPriority, and we have a data structure (threadState) with which can store all the priorities which also account for the priority donations. We can get the effective priority of each thread, and run threads based on their effective priority (which accounts for priority donations). Anytime we set priority, we need to re-update effective priority:

```
void setPriority(int priority) {
       this.priority = priority;
       updateEffectivePriority();
}
```

   We also need to check the priorities when waitForAccess() is happening. Either the blocking thread's priority may be changed or the blocked thread's priority may be changed, so we should update effective priority in waitForAccess().

   Lastly, anytime we call acquire(waitQueue), we should update effective priority, since it is called when the associated thread has acquired access to whatever it is guarded by, so we should check and update effective priorities.

   Thread State is useful because it stores all this information about priority (both regular and effective), so anytime we get priority, we get thread state.

2. Achievement unlocked.

3. (a) $\{r1r2\}$ - both can read at the same time (start and finish at the same time). Everything else arrives in short order, so they are all enqueued in the order in which they arrive.

$w1$ - as soon as there are no readers currently reading and $w1$ is first in queue, it locks the document and writes

$r3$

$w2$ - locks the document after r3 finishes reading

$w3$ - locks the document after w2 finishes (and unlocks)

$\{r4r5r6\}$

$w4$

$w5$

$w6$

$\{r8r9\}$

$w7$

$r10$

Basically, this all runs in order, but if there are sequential readers in the queue, they can read at the same time. Unfortunately, this isn't terribly efficient since there are several times where we only have one reader thread running at once.

(b) A writer can only edit the file if there are no readers reading or waiting to read. This means that readers should have the highest possible priority (higher priority than the writers, that is, so that they will run first). However, we can do this without directly assigning priorities! We can make use of the variables activeReaders, activeWriters, waitingReaders, and waitingReaders. So instead of dealing with priorities, we can run a check to make sure there are no waiting readers before the writers go. We should actually never have waiting readers unless there is a very late reader though, since they should just immediately go, unless one comes in very late while writing is happening. So we just need to modify one line in the Writer thread to make sure a writer thread doesn't run while there are waiting readers! We also need to change the if-statements in Writers to wake the readers if there are any waiting, and they get priority over the writers!

```
package nachos.threads;

import nachos.machine.*;
import java.util.Random;

public class ReadersWriters {
static Lock lock = new Lock();
static Condition okToRead = new Condition(lock);
static Condition okToWrite = new Condition(lock);
static int activeReaders = 0;
static int activeWriters = 0;
static int waitingReaders = 0;
static int waitingWriters = 0;
static void Reader() {
     lock.acquire();
        while (activeWriters > 0) {
         waitingReaders++;
         okToRead.sleep();
        waitingReaders--;
        }
       activeReaders++;
       // reading stuff here
       System.out.println("reading stuff here");
       lock.release();
```

```
        lock.acquire();
        activeReaders--;
        System.out.println("done reading");
        if (activeReaders==0 && waitingWriters>0)
               okToWrite.wake();
         lock.release();
        }
static void Writer() {
lock.acquire();
        while (activeWriters + activeReaders + waitingReaders> 0) {
               waitingWriters++;
               okToWrite.sleep();
               waitingWriters--;
        }
        activeWriters++;
        // writing stuff here
        System.out.println("writing stuff here");
        lock.release();

        lock.acquire();
        activeWriters--;
        System.out.println("done writing");

        if (waitingReaders > 0)
               okToRead.wakeAll();
        else if (waitingWriters > 0)
               okToWrite.wake();
         lock.release();
}

}
```

4. Achievement unlocked.