Midterm 1, CSE 451, Winter 2001 (Prof. Steve Gribble)

Problem 1: (15 points)

Which of the following require assistance from hardware to implement correctly and/or safely? For those that do, circle them, and name the necessary hardware support:

System call	**CPU protection bit for protected kernel mode**
Process creation	**need to do a system call, so need protection bit. Also, need to manipulate address spaces (we didn't know this yet, though)**
Thread creation	[ok if you argued that kernel threads require system calls, and hence you need protection bit]
Process context switch	**need a timer interrupt to prevent starvation, also need to manipulate address spaces.**
Thread context switch	[ok if you argued that kernel threads require system calls, and hence you need protection bit]
Lock acquisition	**need support for atomic instructions (e.g., test and set) or support for disabling/enabling interrupts**
Lock release	[it is possible to build locks that need hardware support only for acquisition but not for release. I gave you points if you argued that if locks are based on interrupts, then you need to be able to reenable interrupts to release a lock.]

Problem 2: (35 points)

Three processes P1, P2, and P3 have priorities P1=1, P2=5, P3=10. ("10" is higher priority than "1".) The processes execute the following code:

```
P1:   begin                          P2:   begin
       <code sequence A>                      <code sequence A>
       lock(X);                               lock(Y);
       <critical section CS>                  <critical section CS>
       unlock(X);                             unlock(Y);
       <code sequence B>                      <code sequence B>
      end                                    end
```

```
P3:  begin
       <code sequence A>
       lock(X);
       <critical section CS>
       unlock(X);
       <code sequence B>
      end
```

The X and Y locks are initialized to "unlocked", i.e., they are free. <*code sequence A*> takes **2** time units to execute, <*code sequence B*> takes **4** time units to execute, and <*critical section CS*> takes **6** time units to execute. Assume `lock()` and `unlock()` are instantaneous, and that context switching is also instantaneous.

P1 begins executing at time **0**, P2 at time **4**, and P3 at time **8**. There is only one CPU shared by all processes.

a) (**14 points)** Assume that the scheduler uses a priority scheduling policy: at any time, the highest priority process that is ready (runnable and not waiting for a lock) will run. If a process with a higher priority than the currently running process becomes ready, preemption will occur and the higher priority process will start running.

   Diagram the execution of the three processes over time. Calculate the job throughput and the average turnaround time.

   On your diagram, use "A" to signify that the process is executing <*code sequence A*>, "B" to signify that the process is executing <*code sequence B*>, "X" to signify that the process holds lock X and is in the critical section, and "Y" to signify that the process holds lock "Y" and is in the critical section, and leave blank space if the process is not executing (for any reason).

   We've started your diagram for you on the next page.

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
P1	A	A	X	X															X	X	X	X											B	B	B	B				
P2					A	A	Y	Y			Y	Y	Y	Y	B	B	B	B																						
P3									A	A													X	X	X	X	X	X	B	B	B	B								

Job throughput: ___3 jobs in 36 units = 3/36 = 1/12 jobs per unit time__

Average turnaround time: __((36-0) + (18-4) + (32-8))/3 = 74/3 units of time__

b) **(14 points)** "Priority inversion" is a phenomenon that occurs when a low-priority process holds a lock, preventing a high-priority process from running because it needs the lock. "Priority inheritance" is a technique by which a lock holder inherits the priority of the highest priority process that is also waiting on that lock.

Repeat a), but this time assuming that the scheduler employs priority inheritance.

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
P1	A	A	X	X							X	X	X	X																			B	B	B	B				
P2					A	A	Y	Y																	Y	Y	Y	Y	B	B	B	B								
P3									A	A					X	X	X	X	X	X	B	B	B	B																

Job throughput: _____1/12 jobs per unit time (same as above) _____

Average turnaround time: ___((36-0) + (32-4) + (24-8))/3 = 80/3 units of time_____

c) **(7 points)** Did priority inheritance help? Why or why not?

Yes, it did. Although the job throughput was unaffected, and the average turnaround time increased (!), the highest priority process finished sooner than before, because P1's priority was boosted through priority inheritance.

Problem 3: (35 points)

Consider the following implementation of reader-writer locks:

```
Class ReaderWriterLock {

   Semaphore  mutex = 1,
              OkToRead = 0,
              OkToWriter = 0;

   int        AR=0,    // # of readers that have acquired a read lock
              WR=0,    // # of readers waiting to acquire a read lock
              AW=0,    // # of writers that have acquired a write lock
              WW=0;    // # of writers that are waiting for a write lock

   void AcquireReadLock() {
      P(mutex);

      if ((AW == 0) && (WW == 0)) { // THE ONLY NEEDED CHANGE!!!

         V(OkToRead);

         AR++;

      } else WR++;


      V(mutex);

      P(OkToRead);
   }

   void ReleaseReadLock() {
      P(mutex);

      AR--;

      if ((AR == 0) && (WW > 0)) {

         V(OkToWrite);

         AW++; WW--;

      }

      V(mutex);
   }
```

(continued on next page…)

```
    void AcquireWriteLock() {
       P(mutex);

       if (AW + AR  == 0) {

          V(OkToWrite);

          AW++;

       } else WW++;

       V(mutex);

       P(OkToWrite);
    }

    void ReleaseWriteLock() {
       P(mutex);

       AW--;

       if (WW > 0) {

          V(OkToWrite);

          AW++; WW--;

       } else {

          while (WR > 0) {

             V(OkToRead);

             AR++; WR--;
          }

       }

       V(mutex);
    }

}
```

(question continues on the next page)

a) **(3 points)** Assuming processes are well-behaved (i.e. they faithfully call Acquire and then Release as expected), is this implementation deadlock free? A simple yes or no will suffice.

YES.

b) **(10 points)** Describe the scheduling policy defined by this implementation (i.e., how readers and writers are scheduled relative to each other).

Readers exclude writers. Writers exclude readers and writers. Once a reader holds a readlock, writers can be starved by an influx of more readers. Once a writer holds a writelock, readers can be starved by an influx of more writers.

c) **(15 points)** A problem with this implementation is that once any reader has a readlock, then an influx of more readers can arbitrarily delay any writers from getting writelocks.

Modify the code above (in place, or next to it) so that it has the following scheduling policy:

i) writers run exclusively
ii) readers may run concurrently with other readers
iii) when any reader is granted a readlock, then all readers waiting for a readlock **at that time** are also granted readlocks
iv) no additional readers are granted readlocks if any writer has requested a writelock

d) **(7 points)** Is this new policy starvation free? If so, how do you know? If not, suggest (but don't implement) a policy that is.

No, it is not. Once a writer holds a writelock, an influx of new writers can starve readers. Here's one starvation free policy:

i) writers run exclusively
ii) readers may run concurrently with other readers, but not with any writers
iii) once there are any waiting writers, no more than N additional readlocks will be granted
iv) once there are any waiting readers, no more than M additional writelocks will be granted