Midterm solution set
CSE 451, Winter 2005 (Prof. Steve Gribble)

Problem 1: (33 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 **3** time units to execute, and *<critical section CS>* takes **7** 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 **5**, 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	X											B	B	B				
P2						A	A	Y			Y	Y	Y	Y	Y	Y	B	B	B																					
P3									A	A														X	X	X	X	X	X	X	B	B	B							

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

Average turnaround time: __((36-0) + (19-5) + (33-8))/3 = 75/3 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	X																				B	B	B				
P2						A	A	Y																	Y	Y	Y	Y	Y	Y	B	B	B							
P3									A	A					X	X	X	X	X	X	X	B	B	B																

Job throughput: ______1/12 jobs per unit time (same as above) ________

Average turnaround time: ___((36-0) + (33-5) + (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 2: (34 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);    // P == decrement

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

         V(OkToRead);    // V == increment

         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

SEE NEW CODE IN SOURCE CODE ON PREVIOUS PAGES

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

Problem 3: (33 points)

Imagine GribbleCorp produced a 64 bit architecture – in other words, pointers to memory addresses on a GribbleCorp machine are 64 bits long. Your job is to design the virtual to physical translation logic for a GribbleCorp machine.

a. **(16 points)** Assume that pages are 4096 bytes big, page table entries (PTEs) are 4 bytes long, and you are told to build a 3 level page table structure, where page tables at each level fit inside a single 4096 byte memory page.

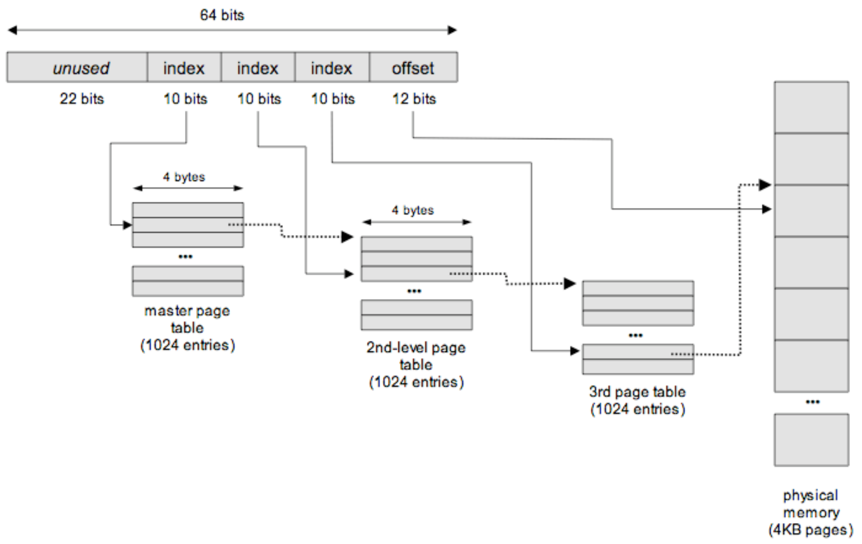
   Draw the virtual address translation logic. Be sure to show which parts of a virtual address are used as an offset into a page, and which parts of a virtual address are used as indexes into each of the page tables.

   How big is the addressable virtual address space?

   Pages are 4096 bytes long, and page table entries are 4 bytes long.

   - → **therefore, there are 1024 page table entries per page table**
     (this is true for both the master page table, secondary page tables, and tertiary page tables)
   - → **therefore, the indexes into the page tables need to be 10 bits long**
     (2^10 = 1024)
   - → **since pages are 4096 bytes, the offset into a page needs to be 12 bites long**
     (2^12 = 4096)

   Thus, the logic looks like:



64 bits
unused
index
index
index
offset
22 bits
10 bits
10 bits
10 bits
12 bits
4 bytes
4 bytes
master page table (1024 entries)
2nd-level page table (1024 entries)
3rd page table (1024 entries)
physical memory (4KB pages)

   Since there are 22 unused bits in a virtual address, and 42 used bits, the addressable virtual address space is 2^42 bytes large.

b. **(17 points)** Now assume page table entries (PTEs) are 8 bytes long, page tables must fit in exactly one page, and you are told to build a 4-level (!!) paging structure. What is the largest page size that you can support?

Sketch out the resulting virtual to physical translation logic.

Let's assume that a page is X bytes long, and we need Y bits to index a page. (In other words, 2^Y = X.)

Then, we need a Y bit offset.

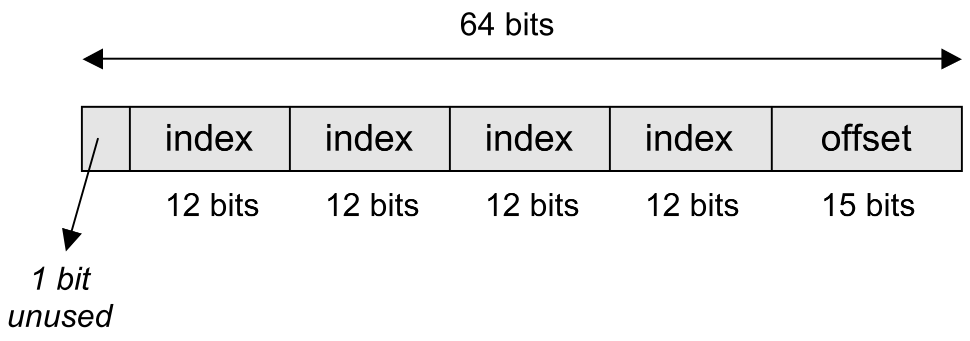
Also, each page table will contain X / 8 entries, since each PTE is 8 bytes long. Thus, we need Y-3 bits to index a page table. (2^(Y-3) = (2^Y) * (2^-3) = X / 8)

We need to find the largest Y so that we can fit 4 indexes + 1 offset in 64 bits:

4(Y-3) + Y <= 64
5Y <= 76
Y <= 15.2, or Y = 15 is the largest (integer) Y we can have.

Thus, the offset is 15 bits long, and a page is 2^15 = 32768 bytes big.
➔ 32,768 bytes is the largest page size we can support

Given this, we chop up a virtual address as follows:



The virtual to physical translation logic looks just the same as in 3a, except with 4 levels of page tables, and:

- **page tables are 32768 bytes long**
- **each page table holds 32768/8 = 4096 entries**
- **the offset = 15 bits long**

(1 bonus point) In what year did Apple Computer release the Macintosh personal computer?

1984!!

The URL for the commercial I showed in class is:

http://www.uriah.com/apple-qt/1984.html