CSE 410 Final Exam Sample Solution 6/08/10

Question 1. (12 points) (caches)

(a) One choice in designing cache memories is to pick a block size. Which of the following do you think would be the most reasonable size for cache blocks on a computer designed for general-purpose laptop or desktop applications? Circle your answer and justify it.

8 bytes   (64 bytes) [circled]   512 bytes   2048 bytes

64 bytes takes advantage of spatial locality by loading neighboring data that is likely to be used after a cache block is first referenced. 8 bytes is too small for this. 512 and, especially, 2048 are so large that cache misses are likely to load lots of data that is never used, increasing memory traffic for no net benefit in the cache hit rate.

(b) What problem does a 2-way associative cache solve compared to a direct-mapped one?

In a direct-mapped cache, if two active cache blocks have the same index, they will continually replace each other, generating a cache miss whenever the other one is referenced. With a 2-way associative cache, both blocks can reside in the cache at the same time.

(c) A design choice for a cache memory is whether to use a write-back or a write-through policy. What is the difference between these?

Write-through: when data is written, the change is immediately written to main memory as well as the cache.

Write-back: when data is written, only the cache copy is updated. The main memory copy is not updated until that cache block is evicted from the cache.

Question 2. (12 points) Suppose we have a memory system that has a main memory, a single-level cache, and paging virtual memory. The three levels of the memory system have the following access times:

2 ns	cache
100 ns	main memory
10 ms	paging disk

(a) The cache has a 95% hit rate. What is the effective memory access time if we consider only the cache and main memory and ignore page faults and disk access times? (To save time, you only need to clearly write out the formula, but do not have to simplify the final result.)

2 + (0.05 * 100) ns = 7 ns.

(b) Now recalculate the effective memory access time assuming the same cache hit rate (95%) plus a page fault rate of 0.001% (i.e., 99.999% of the memory accesses succeed without producing a page fault). (Again, a formula is adequate.)

2 + (0.05 * 100) + (0.001 * 10,000,000) ns = 10,007 ns.

Notes: The page fault and cache miss rates are a percentage of the total memory references, not a percentage of the references to the next level up or down in the memory hierarchy.

Also, note that the page fault rate in (b) is way too high. That's not an issue for this problem, but it's worth noting that these numbers would not be realistic in actual systems.

Question 3. (12 points) (paging) (a) Belady's MIN algorithm for replacing pages in a virtual memory system has the provably lowest page-fault rate of all theoretical or practical page replacement algorithms. What is this page replacement policy?

Evict the page that will not be referenced for the longest time in the future.

(b) No real systems use Belady's MIN algorithm. Why not?

Can't predict the future.

(c) There are several other page replacement algorithms that could be used instead of MIN in a real system including FIFO, approximate LRU, and randomly picking a page frame to reuse after a page fault. Which one of these is the best choice for realistic systems and why?

Approximate LRU. This picks pages that have been unused for a long amount of time, and it's reasonable to assume that they are less likely to be used in the near future than other pages that have been used more recently.

FIFO can easily make a wrong choice if an active page has been in main memory for a long time, plus it suffers from Belady's anomaly, where it can actually do a worse job if it has more page frames available. Random can also make a bad choice compared to LRU and its approximations.

Question 4. (10 points) (threads) Threads can be implemented as user-level threads using a library package that runs entirely in a process address space, or as kernel-level threads, where the operating system kernel is aware when a process has more than one thread associated with it.

(a) Give one advantage that user-level threads have compared to kernel-level threads.

Very low overhead for thread scheduling and context-switch operations compared to kernel threads.

(b) Give one advantage that kernel-level threads have compared to user-level threads.

Better scheduling decisions if the kernel is aware of all of the threads in the system and what the various processes are doing. One example: if a single kernel thread blocks for I/O or some other operation it does not block the entire process, which can happen if a user-level thread package is managing a single kernel thread.

Kernel threads also allow multiple threads in a single process to run simultaneously on a system with more than one processor or core.

Question 5. (8 points) Translation Lookaside Buffers (TLBs) have often been implemented with a fully associative lookup, while cache memories never use this. Give a brief technical explanation of why a fully associative lookup scheme can make sense for a TLB but not for a cache.

TLBs are typically fairly small (several dozen to a few hundred entries), so a fully associative structure can be built at a reasonable cost and without slowing down memory references. Main memory caches are normally too large for this, plus they tend to have good hit rates with a moderate amount of associativity and don't need a fully associative lookup.

Question 6. (10 points) (scheduling) Suppose we have the following jobs to execute, with the given start times and total times needed for each job.

Job	Start Time	Total time
A	0	25
B	10	60
C	25	30

Now suppose we execute those jobs using a Round-Robin scheduler with a time quantum of 20. Fill in the chart below to show the timeline of when each job executes, starting with time 0 in the first row, and filling in each successive row with a time and the letter identifying the job that starts or resumes execution at that time. Continue until all of the jobs are finished. You may not need all the rows. The first row is filled in for you

Time	**Job**
0	**A**
20	**B**
40	**A**
45	**C**
65	**B**
85	**C**
95	**B**
[105 done]	

Question 7. (10 points) (locks) In lecture we discussed the implementation of low-level lock primitives `acquire` and `free`. If we represent a lock as a word in memory with the value 1 when it is held by a thread and 0 when it is free, the `free` operation is easy – just store 0 in the lock variable. Acquire needs to set the lock variable to 1, but it must be done correctly in case some other thread holds the lock or is trying to acquire it.

A common instruction provided on many computers to help with this is compare-and-swap (`cas`). A typical version of this involves three registers – two containing values and a third containing the address of the lock. Here is a pseudo-code description of the operation of `cas`:

```
cas  x,y,addr ≡
   temp = MEM[addr];
   if (temp == x) {
      MEM[addr] = y;
   } else {
      x = temp;
   }
```

(a) Complete the following assembly language implementation of the operation `acquire` that so it can be executed by a thread to acquire a lock correctly. Register `$a0` contains the address of the lock variable. Fill in the blanks, including the correct conditional branch operand on the "b____" instruction.

```
# acquire the lock whose address is given in $a0.

acquire:

     li      $t0, ____0____

     li      $t1, ____1____

     cas     $t0, $t1, $a0

     beq     __$t0__ , __$t1__ , acquire

                 # repeat if lock not acquired yet

     jr      $ra  # return to caller once lock acquired
```

(continued next page)

Question 7. (cont.) Definition of compare-and-swap repeated here for reference:

```
cas  x,y,addr ≡
   temp = MEM[addr];     |
   if (temp == x) {      | *
      MEM[addr] = y;     |
   } else {
      x = temp;
   }
```

(b) Compre-and-swap is always implemented as a special instruction in the processor and memory system. Give a technical reason why this is necessary and why the `cas` instruction cannot be correctly replaced by a software procedure that performs the same operations as shown in the pseudo-code above.

The memory read/modify/write sequence marked (*) above must execute as a single atomic operation, with no other thread able to read or write the lock variable while it is in progress. Hardware can guarantee this; it cannot be guaranteed for the equivalent sequence of individual instructions in software.

Question 8. (10 points) In languages like C, C++, Fortran, and others, it is possible to create an array of objects (structs) without allocating the objects individually with `new` or the equivalent construct. For example, suppose a single complex number is represented as a pair of `doubles`.

```
class Complex {          // complex number
   double re;            // real part
   double im;            // imaginary part
}
```

When we create an array of `Complex` data, it would look something like this in memory:

0	1	2	3	4	5	6	7	8	9	10	…
re,im	re,im	re,im	re,im	re,im	…						

Now for the question. We would like to initialize this array to (0.0, 0.0). Here are two possibilities:

I

```
for (k = 0; k < n; k++) {
   a[k].re = 0.0;
}
for (k = 0; k < n; k++) {
   a[k].im = 0.0;
}
```

II

```
for (k = 0; k < n; k++) {
   a[k].re = 0.0;
   a[k].im = 0.0;
}
```

Is there any reason to prefer one of these solutions over the other for performance or other technical reasons? If so, why; if not, why not?

Yes, II is better. Version II goes through the entire array sequentially. Each cache block is loaded once and completely processed before going on to the next. Version I touches each cache block on each loop and there is a good chance that the second loop will need to reload the cache blocks from main memory if the cache is busy or the array is large.

If the array is too large to fit in main memory, or if memory is heavily utilized, version I could also generate significantly more page faults if the pages occupied by the array have to be brought in again from disk for the second loop.

Question 9. (8 points) (synchronization) Inspired by our success in solving the bridge traffic problems (hw8), we have been invited back to solve a synchronization problem involving cows on a Vermont farm.

This farm has a milking barn with 3 stalls. Only 3 cows can fit in the barn at a time, however if another cow sticks its head in the door, it won't leave since it wants to be milked and eat its dinner.

Use a counting semaphore to implement a pair of procedures to control entry to the barn so that at most 3 cows are in the barn at any one time. Each cow will execute procedure `enter_barn()` before entering the barn and `exit_barn()` when leaving. You need to provide an initial value for the semaphore `ok_to_enter` and implementations of these two procedures below.

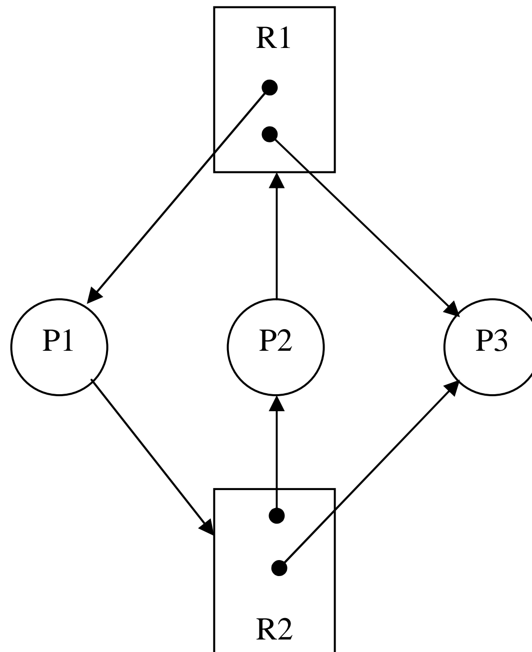
```
semaphore ok_to_enter = _____3______ ;

void enter_barn() {
      ok_to_enter.wait();
}

void exit_barn() {
      ok_to_enter.signal();
}
```

It's fine if you used other notations like `wait(ok_to_enter)` or `P` and `V`.

Question 10. (8 points) (deadlocks) Consider the following resource allocation graph. Rectangles indicate resources; circles indicate processes. There is an arrow from a process to each resource it is requesting but does not hold, and an arrow from each resource to a process that holds a lock on that resource.



Is this system deadlocked? Why or why not? (Give a precise reason why there is a deadlock or how you can prove that the system is not deadlocked.)

Not deadlocked. P3 has all of the resources it needs to execute, so it will proceed. Once P3 terminates it will release its resources (one each of R1 and R2). At that point, both P1 and P2 will be able to get the resources they are waiting for and will be able to execute.