UNIVERSITY OF TORONTO

Spring 2010 Midterm Test
Course: CSC 369 H1S
Instructor: Angela Demke Brown
Duration: 50 minutes
Aids allowed: none

Last Name: ____________________

Given Name: ____________________

Student Number: ____________________

This midterm test consists of 5 questions on 10 pages (including this one and two scrap pages). When you receive the signal to start, please make sure that your copy of the test is complete.

If you need more space for one of your solutions, use the last pages of the test and indicate clearly the part of your work that should be marked. We have been careful to leave enough space for your answers.

In written answers, be as specific as possible and explain your reasoning. Clear, concise answers will be given higher marks than vague, wordy answers. Marks will be deducted for any incorrect statements in an answer. Please make your handwriting legible!

MARKING GUIDE

1: ______/ 8

2: ______/12

3: ______/ 8

4: ______/10

5: ______/12

TOTAL: ______/50

CONTINUED ON PAGE 2

8 marks

Question 1. Very Short Answer

Each of the questions below should be answered with a couple of words or one sentence.

1 mark **Part (a)** What is the primary hardware mechanism that allows operating system kernels to be protected from user processes?

mode bit (or some mention of privileged / unprivileged modes of execution)

We gave 1/2 for MMU or memory protection, but you also need a way of preventing user code from modifying the memory permissions.

1 mark **Part (b)** The operating system maintains a data structure to represent each thread. List two items (give a general description, not the OS/161 variable names) that would appear in this structure.

thread / process id, current state (running, blocked, etc), scheduling priority, pointers to link thread onto queues... lots of other possibilities

1 mark **Part (c)** What is the main problem with using an atomic hardware instruction like "test-and-set" to implement a `lock_acquire` function on a uniprocessor?

busy waiting [we also gave part marks for noting that this leads to non-portable code, but that is not the main problem.]

1 mark **Part (d)** TRUE/FALSE: Some synchronization problems that can be solved with a monitor cannot be solved using only semaphores.

FALSE. Monitors and semaphores have equal power for solving synchronization problems. (only need to say TRUE or FALSE)

1 mark **Part (e)** What does it mean for the `exec` system call to return?

It means the exec failed. On success, a new program starts executing.

1 mark **Part (f)** Name two scheduling algorithms that can cause starvation.

Shortest Job First, Priority

1 mark **Part (g)** Define the term *conflict-serializable.*

concurrent transactions appear to complete in some serial order

1 mark **Part (h)** TRUE/FALSE: A dynamic partitioning system with load-time address binding can use compaction to deal with external fragmentation.

FALSE (load-time address binding does not permit dynamic relocation, which is needed for compaction).

CONTINUED ON PAGE 3

12 marks

Question 2. Short Answer

Each of the questions below should be answered with a paragraph or a diagram.

3 marks

Part (a) What is the main difference between an interrupt (e.g. a timer interrupt) and an exception (e.g. a system call)? In what way are they the same?

Interrupts are caused by something outside the currently executing thread/process. In other words, they are asynchronous with respect to the thread's execution and could happen at any arbitrary point. Exceptions are caused by something the thread/process did itself and happen at the precise point when the causing instruction was executed (e.g. syscall instruction, divide-by-zero, bad address reference). They are the same because both cause the executing process to enter kernel mode, save its state, and handle the interrupt or exception.

3 marks

Part (b) In Exercise 1 you created several threads that concurrently ran a loop in which they first incremented and then decremented a shared variable `counter` without any synchronization, as follows:

```
for (i = 0; i < niters; i++) {
      counter++;
      counter--;
}
```

We observe the following on a Linux system: When `niters=1000`, on a uniprocessor the final value of `counter` is almost always the same as its initial value regardless of the number of threads used, but on a multiprocessor different final values for `counter` are frequently observed. Explain this result.

On a uniprocessor, we can only get inconsistent results if the scheduler switches to another thread while a thread is in the middle of an update. With a small number of iterations, the thread is likely able to complete its entire loop in a single scheduling quantum without interruption. On a multiprocessor, threads truly run concurrently so there are many more opportunities for bad outcomes.

CONTINUED ON PAGE 4

3 marks **Part (c)** One strategy for deadlock prevention is to break the "no pre-emption" condition. Identify one type of resource for which this works, and one type of resource for which it does not. **Explain briefly.**

Works: CPU, memory. Does not work: printer, CD burner. In general, it works if we can save the state of the preempted resource and restore it later when it is returned to the original owner. It does not work if the resource has some state that cannot be saved and restored or the operation cannot be rolled back and restarted.

3 marks **Part (d)** Suppose we have a dynamic partitioning system with 15 units of memory. Sketch the memory space after the sequence of events below. Label (i) allocated space showing the request it belongs to, (ii) free space, and (iii) external fragmentation. Assume that a first-fit allocation policy is used:

1. Allocate 4 units for P1
2. Allocate 4 units for P2
3. Allocate 3 units for P3
4. Free P2's allocation
5. Allocate 3 units for P4

P1
P1
P1
P1
P4
P4
P4
free (external frag)
P3
P3
P3
free
free
free
free

continued on page 5

8 marks

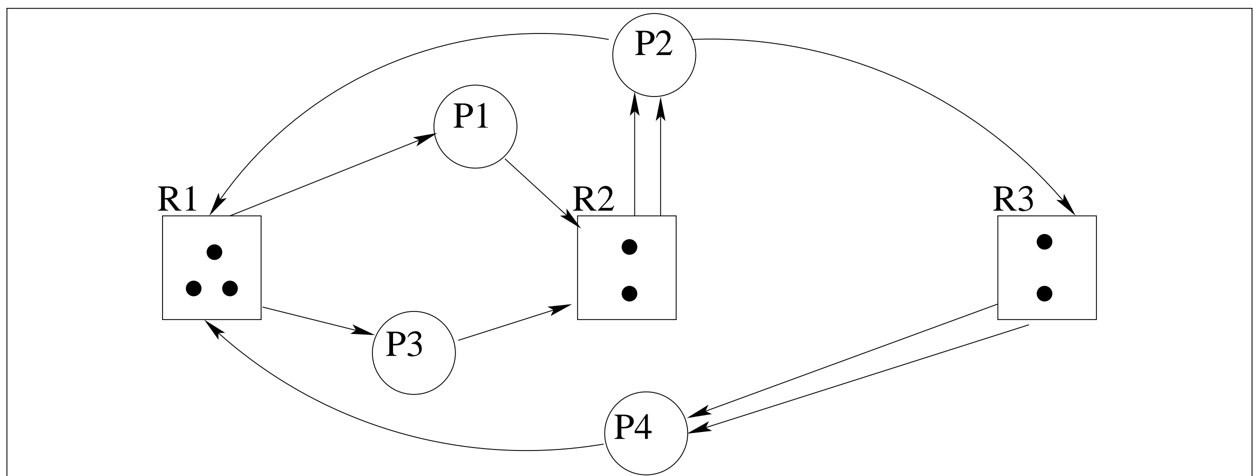
Question 3. Deadlock

A system has 4 processes (p1, p2, p3, p4) and 3 types of resources (R1, R2, R3). There are 3 units of R1, 2 units of R2 and 2 units of R3. At a particular point in time the following requests and allocations exist:

- p1 holds 1 unit of R1 and requests 1 unit of R2
- p2 holds 2 units of R2 and requests 1 unit of R1 and 1 unit of R3
- p3 holds 1 unit of R1 and requests 1 unit of R2
- p4 holds 2 units of R3 and requests 1 unit of R1

4 marks

Part (a) Complete the resource allocation graph below. Nodes for the resource types have been drawn already (dots within a resource node indicate instances of the resource type).



We were mainly looking for the conventions for drawing a resource allocation graph: (i) processes shown as circles, (ii) resource allocation shown as an arrow from resource to process and (iii) ungranted request shown as an arrow from process to resource. Finally, you needed to show all of the requests and allocations on the graph.

4 marks

Part (b) Both p2 and p4 have made a request for the last instance of R1. Does it matter which process is allocated this resource instance? **Explain your answer**, describing the outcome of both options.

Yes it matters. If p2 is granted the last instance of R1, then we have a deadlock since p2 is waiting for an instance of R3, both of which are held by p4, and p4 is waiting for an instance of R1. All instances of R1 are held by processes which are also blocked – p2 is blocked waiting for p4 to release R3 and both p1 and p3 are blocked waiting for p2 to release R2.

If p4 is granted the last instance of R1 the system can still make progress. since p4 has all the resources it needs, it will eventually complete, releasing its instance of R1 and both instances of R3. Then the outstanding requests for p2 can be granted, p2 will eventually complete and release R2, allowing p1 and p3 to finish.

CONTINUED ON PAGE 6

10 marks **Question 4.** CPU Scheduling

Schedule the three threads shown in the table below. Priorities are fixed with **higher numbers corresponding to higher priorities**. A timer interrupt occurs at the beginning of each time interval and the scheduler chooses the next thread to run (possibly allowing the current one to continue running). Scheduling decisions are *only* made as part of handling the timer interrupt. Assume that no time is required to choose the next thread and switch to it.

Process	Arrival Time	Service Time	Priority
A	0	10	7
B	2.5	5	15
C	4.5	2	3

3 marks **Part (a)** Round Robin (RR), time quantum = 2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	A	B	B	A	A	C	C	B	B	A	A	B	A	A

Common errors included starting a new process as soon as it arrived, rather than waiting for the expiry of the current process's quantum, and scheduling C before A in the 4th quantum (7th time interval). A entered run queue at $T = 4$ when its quantum expired, and C entered on arrival at $T = 4.5$, so A is ahead of C.

3 marks **Part (b)** Priority Scheduling (PRI):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	B	B	B	B	B	A	A	A	A	A	A	A	C	C

The only common problem was reversing the priorities (so that C had highest priority and B lowest).

3 marks **Part (c)** Shortest Remaining Processing Time (SRPT):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	B	B	C	C	B	B	B	A	A	A	A	A	A	A

The most common problem was using shortest-job-first, which is non-preemptive (so A runs to completion, then C, then B).

1 mark **Part (d)** If the time to perform a context switch is non-zero, which of these scheduling algorithms has the lowest overhead for this set of threads?

The lowest overhead is simply the one with the fewest context switches, which is PRI (3 switches). The most common error was to misinterpret overhead as average wait time.

continued on page 7

12 marks

Question 5. Synchronization and Threading

The Ontario government is considering allowing online gambling. The company WeNeedsCash, has decided to get into the marketplace quickly and establish a customer base. They have launched a web service that allows customers to create an account, deposit funds using a credit card, and then play the WeNeedsCash game by clicking a button labeled "SHOW ME THE MONEY!" Every button click debits their account by $1, until it reaches zero. Since there is no actual gambling, they do not need to wait for government approval.

The initial implementation of the server uses a single thread. Each click on the "SHOW ME THE MONEY" button results in a request that is handled by a function call to `show_money(account)`, where `account` refers to a data structure representing the user's account. C code for the account structure and the `show_money` function are shown below:

```
1.   struct account_s {
1.1   struct lock *l; /* exact syntax for locks used is not important */
2.        unsigned int balance;
3.        char *username;
4.   }

5.   int show_money(struct account_s *account) {
5.1       int new_balance;
5.2       lock_acquire(account->l);
6.        if (account->balance > 0) {
7.               account->balance = account->balance-1;
8.        }
8.1       new_balance = account->balance;
8.2       lock_release(account->l);
9.        return new_balance; /* modified return statement */
10.  }
```

2 marks

Part (a) You are hired as the software quality control expert for WeNeedsCash. Do you find any potential data races in the code? Explain your answer.

Since the server is single-threaded, there is no possibibility of a race condition. (Races happen when 2 or more threads access shared data without synchronization and at least one of the accesses is a write.)

The WeNeedsCash site becomes wildly popular, and the single-threaded server is taking too long to respond to client requests, limiting profits. The company rushes to fix the problem with the purchase of a 128-CPU multiprocessor for the server and a new multithreaded implementation. For each click request, the main server thread now creates a new thread which runs `show_money` and then exits. To save time, they don't bother with a software quality review before deploying the new system.

Jerry Jackpot has two computers and plays the WeNeedsCash game simultaneously on both using the same account. His account balance is nearly at zero. Clicking madly on "SHOW ME THE MONEY" on both screens, he notices that his balance has suddenly jumped to over 4 billion dollars!

Panicked WeNeedsCash executives now call for a code review.

continued on page 8

4 marks **Part (b)** Explain Jerry's good fortune. Show a simple scenario involving two threads, T1 and T2, simultaneously executing `show_money` that illustrates how a huge account balance could result.

Assume Jerry's account balance is $1. T1 and T2 concurrently evaluate the condition on line 6 on different processors, and since the balance is 1, both T1 and T2 proceed to line 7. T1 now decrements the balance and stores 0 as the result. T2 now reads the new balance of 0 and subtracts 1. However, since balance was declared as "unsigned int", it cannot store the value "-1". Instead underflow results and it gets the twos-complement bit pattern of -1 (all bits set to 1), which is $2^{32} - 1$*, or over 4 billion as an unsigned value.*

4 marks **Part (c)** Add synchronization to the code to prevent the problem. Show your changes beside the existing code, numbering your lines to clearly indicate where they should go (e.g. a line of code that should go between lines 2 and 3 would be numbered 2.1). Remember that WeNeedsCash wants to handle requests as quickly as possible, but without losing money!

Important things we were looking for were: (i) use of a per-account lock (or binary semaphore) so that operations on different accounts could proceed concurrently, (ii) treating lines 6-8 as a single critical section (so the lock is acquired before line 6 and released after line 8). To ensure each call to `show_money` returns the result of its own operation, there should really be a local variable assigned the result within the critical section, and a return of that variable rather than returning account-¿balance. However, this last point does not relate to the "huge account balance" race that you were asked to fix, so we did not award or deduct any points for it.

WeNeedsCash has heard complaints from customers that the new service is no faster than the old one. Measurements of customer response time show that the new system is actually SLOWER than the original. They repeat the tests with the new synchronization code removed and get similar results (slower than the original single-threaded implementation). They try both user-level and kernel-level thread implementations but still are unable to obtain a speedup over the original version.

2 marks **Part (d)** Explain why multithreading cannot speed up their code.

In the original single-threaded version, the main server thread had to make a function call that performed one test and one arithmetic operation for each request. In the multi-threaded version, the main server thread has to create a new thread for each request. The time to handle any request is now the time to create the thread plus the time to execute `show_money`. Although the `show_money` operations can be done in parallel, the thread creation is still handled by a single thread, and even the most efficient thread implementation will be more expensive than the simple `show_money` function. In other words, the overhead of thread creation outweighs the benefit of using multiple processors.

END OF SOLUTIONS