University of California, Berkeley
College of Engineering
Computer Science Division – EECS

Spring 2008 Anthony D. Joseph

Midterm Exam *Solutions*

February 27, 2008
CS162 Operating Systems

Your Name:	
SID AND 162 Login:	
TA Name:	
Discussion Section Time:	

General Information:
This is a **closed book and notes** examination. You have 90 minutes to answer as many questions as possible. The number in parentheses at the beginning of each question indicates the number of points given to the question; there are 100 points in all. You should read **all** of the questions before starting the exam, as some of the questions are substantially more time consuming.

Write all of your answers directly on this paper. ***Make your answers as concise as possible.*** If there is something in a question that you believe is open to interpretation, then please ask us about it!

Good Luck!!

Problem	Possible	Score
1	33	
2	23	
3	17	
4	27	
Total	**100**	

1. (33 points total) Short answer questions:

a. (4 points) True/False and Why?

In the operating system environments that we've studied so far, an application programmer has to worry about deadlock occurring involving the CPU as a resource.

TRUE FALSE

Why?

FALSE. *The CPU is easily preempted which eliminates one of the four preconditions for deadlock. The correct answer was worth 2 pts and a reasonable explanation that mentioned the OS preempting the CPU was 2 pts. Some students incorrectly said there was no waiting while holding the CPU, but a program could busy wait to create a deadlock.*

b. (4 points) True/False and Why?

In the operating system environments that we've studied so far (i.e., no virtual memory), the operating system has to worry about deadlock occurring involving physical memory as a resource.

TRUE FALSE

Why?

TRUE. *Physical memory cannot be preempted. Some students mistakenly thought that the resource in question was a lock in memory, instead of simply chunks of memory necessary for process execution. We gave full credit for False answers that said as their justification that OSs we've studied do not worry about deadlock and it is the application programmers responsibility.*

c. (5 points) Disabling interrupts:

i) (3 points) Give a two to three sentence description of how interrupts can be used to implement critical sections.
Disabling interrupts prevents interrupts from involuntarily preempting the current thread (2 pts). If the thread does not voluntarily give up control, it can run the critical section to completion without interference (1 pt).
We only gave 1 pt for this problem if the answer described how the implementation works with little other explanation about the effects of disabling interrupts.

ii) (2 points) Briefly (2-3 sentences) state the problems with implementing critical sections using disabling of interrupts.
We gave 2 pts for answers that mentioned any of the following – disabling interrupts doesn't work on a multiprocessor can't be used by user-level applications, could allow infinite loops, arbitrarily delay other processes, may cause real-time systems to fail, and may cause interrupt buffer overflow.
We only gave 1 pt for answers that mentioned lesser issues – may lead to unfair processor allocations, may cause other processes to wait a while, disabling interrupts for a while is bad, and other issues that were either less realistic or less well defined.

d. (8 points) List the four requirements for deadlock.

Mutual exclusion, non-preemptable resources, hold and wait, circular chain of waiting.
Each requirement was worth 2 points.

e. (4 points) Briefly explain why when using a Hoare semantic monitors it is sufficient to use "if" statements when checking state variables, while most OS monitor implementations require the use of "while" statements.

With Hoare semantic monitors, the signaled thread is guaranteed to receive the lock and control immediately, thus the recipient is guaranteed to be able to proceed. Most OS monitor implementations only place the signaled thread on the ready queue; thus, other threads may run (and access the critical section) between the time that the thread is signaled and when it actually runs.
We deducted 1 pt for only saying something about Hoare and not comparing to Mesa/OS monitors, or not emphasizing that another thread can change the synchronization condition the initial thread slept on. If the answer was vague and didn't imply an understanding of the IfWhile issue, and was just a definition of a Hoare monitor, then it we deducted 2 pts. Extraneous, vague, or off-topic answers resulted in another 1 pt deduction.

f. (8 points) In lecture, we discussed using atomic instructions (e.g., test-and-set and swap) to implement spinlocks. With spinlocks, threads spin in a loop (busy waiting) until the lock is freed. We motivated the use of blocking locks and semaphores as an improvement over spinlocks because having threads spin can be very inefficient in terms of processor utilization. However, spinlocks are not always less efficient than blocking locks. In two to three sentences, briefly describe a scenario where spinlocks would be more efficient than blocking locks.

Spinlocks are more efficient when the critical sections are very short (e.g., one or a few instructions that increment a shared variable), or when the critical section is rarely contested (i.e., Acquire almost always will succeed). Spinlocks are more efficient in these cases than, say, semaphores, because the overhead to acquire the lock is incredibly low (one instruction). I also accepted the multiprocessor answers from the book.

We deducted 2 points for not mentioning how spinlocks have less overhead than other locks (like semaphore).

We deducted 4 points for not saying when to apply these spinlocks, or -2 points if it wasn't clear how your explanation applied to the locking issue, especially with respect to the "shortness" or "uncontestedness" of the lock.

Partial explanations that didn't answer the question were a 6 point deduction.

Completely off-topic answers received no credit.

2. (23 points total) Implementing and Using Synchronization Primitives.

a. (6 points) While working on the first project, one of your project partners proposes to use a modified version of Semaphore to implement the condition variable class as follows:

```
Semaphore.set(0);         // Initialize semaphore to 0
Wait(Lock lock) {
     Semaphore.P(lock);   // Special semaphore that
                          // releases lock if it waits
}

Signal() {
     Semaphore.V();
}
```

Decide whether you think their proposal will work or not, and briefly, in two to three sentences, explain your decision

This will not work because Semaphores are commutative, while condition variables are not. Semaphores have history (their counter), and repeated signal calls will increment the counter, causing subsequent wait calls to proceed immediately.

Several students said that this implementation would not work because Wait() does not reacquire the lock after P(lock) is returned. However, the re-acquisition of the lock was implicit in the P(lock) function, just as the release of the lock was implicit as well. No credit was given for this answer.

Other students claimed that the implementation wouldn't work because to call Signal() you first had to call Wait() and since the initial value of the Semaphore was 0, it would deadlock. However, it is not necessary that Wait() has to be called by a thread in order to call Signal(). It is only necessary that you possess the lock to execute either function. No credit was given for this answer.

Some students understood that the Signal() implementation was erroneous because it incremented the Counter and changed the state of the Condition variable. However, they failed to explicitly state why this was an issue – i.e., they didn't talk about commutativity or how calling Signal() would let another thread that called Wait() to immediately return from the call. We took off either 2 or 3 points depending on the correctness and quality of the explanation.

b. (9 points) Another one of your project partners proposes the opposite – to implement semaphores using an unmodified Mesa-style Monitor class. Show a possible implementation of a Semaphore class using monitors. You should provide the initialize(value), P(), and V() operations.

```
Semaphore {
    int value;
    Lock lock = FREE;
    Condition cond = NULL;

    initialize(int initialvalue) {
        if (initialvalue < 0) throw exception;
        value = initialvalue;
    }

    P() {
        lock.acquire();
        while (value == 0)
            cond.wait(lock);
        value--;
        lock.release();
    }

    V() {
        lock.acquire();
        if (++value == 1)
            cond.signal();
        lock.release();
    }
```

A correct P() implementation was worth 4 points. A correct V() implementation was worth 4 points. A correct initialize() implementation was worth 1 point.

-1 for not checking if the value passed to initialize was less than 0.

-1 for not explicitly stating that a lock is acquired/released or for not using the synchronized keyword in the method definitions. Any misuse of a lock (e.g., forgetting to acquire/release) also incurred -1.

-2 for using a "while" instead of an "if" in P(). [However, some people who used "if" kept an extra variable that kept track of the number of threads waiting in P() and only incremented the value of the semaphore if this variable was zero. This was acceptable]

-2 for releasing the lock and THEN trying to use a condition variable

-2 for failing to call ConditionVar.Signal() in V()

-2 for disabling/enabling interrupts instead of using locks.

-2 if your semaphore did not work for corner cases. Some student's implementations allowed P() to be executed twice without blocking, even if the initial value was 1.

-2 if your P() always blocked, regardless of the value.

-2 if, for some reason or another your V() call blocked.

-2 if your V() or P() didn't increment/decrement the value every time (unless you were one of the people who had a working implementation with "if")

-2 for any concurrency issues that existed even if locks were used.

c. (8 points) Nested Monitors.

i) (5 points) In this question, we'll consider the effects of nesting two monitors, M1 and M2:

- Inside the code in Monitor M1 that manipulates M1's state variables, there is a procedure call into Monitor M2
- M2 then waits on one of its condition variables (releasing the lock for Monitor M2).

Assuming that M2 does not call any other monitors, including M1, are there any problems with nesting monitors in this manner? Use a simple example to explain your answer.

Even though waiting in M2 releases M2's lock, this situation can result in deadlock as follows: a thread that calls from M1 to M2 will have to hold the M1's lock (since it is manipulating M1's condition variables). If the thread that will issue the signal on M2 executes in similar fashion (i.e., calling a function in M1 and then a signal function in M2), it will not be able to complete the function in M1 (since the original thread still holds the lock), and thus, it will not be able to signal the original thread in M2.

-5 if you said there is no problem.
-3 if you said that there is a problem, but do not give an appropriate explanation. The following quotes are valid examples, but are not relevant to the stated problem and as a result, no credit was given:

- *"M2 sleeps with M1's lock meaning that M1 cannot proceed until M2 wakes up."*
- *"What if M2 sleeps with M1's lock, and then another thread enters M2 and then attempts to acquire M1's lock"*

ii) (3 points) Suppose that the wait in M2 also releases the lock for Monitor M1. Does your answer for part (i) change or not? Briefly explain your answer.

Deadlock is no longer a problem, however, this solution would violate mutual exclusion for Monitor M1 and could result in an inconsistent state for the data being protected by M1.

-3 if you said there was no problem.
-2 or -3 if you said there was the a problem, but didn't give an appropriate explanation.

A common explanation was:

> *"It is only a problem if M2 doesn't reacquire M1's lock as well when it wakes up."*

Unfortunately, it ***is*** *a problem even if M2 did acquire M1's lock upon waking up. The reason is that if a thread is in M1's critical code and a call to a method causes the thread to lose its lock when it was not expecting it, then would allow multiple threads to execute in M1's critical section. Depending on your explanation, you received either 1 point or none at all.*

3. (17 points total) Concurrency problem: Dining Graduate Students.

The Dining Graduate Students problem is as follows. Six graduate students are seated around a table with a large deep dish pizza in the middle. Graduate students are very refined and so they eat pizza with forks and knives. But they don't have a lot of money, so they have the utensils. There are three forks and three knives in a pile next to the pizza. Each student uses the following algorithm to eat:

(1) Pick up a knife
(2) Pick a fork
(3) Cut out a slice of pizza and eat it
(4) Return the knife and fork to the pile

a. (4 points) Specify the correctness constraints. Be succinct and explicit in your answer.

A graduate student waits for a knife and then a fork (4 points).
We gave one point for some other type of related constraint (e.g., you can't steal utensils, there's one fork and one knife per diner, etc.)

-1 for not stating a student must have both knife and fork before eating.
-1 for not stating a student must pick up a knife before picking up a fork.
-1 for saying any implementation-dependent details (i.e. thread, lock, and etc.)
-1 for each statement that is wrong (i.e. only one student can eat at a time).

b. (5 points) Can deadlock ever occur? Explain your answer in terms of the conditions for deadlock to occur. If deadlock could occur, describe a reasonable deadlock avoidance algorithm. If deadlock cannot occur, explain why not.

Deadlock cannot occur, since the students' algorithm defines a partial ordering for resource acquisition (knives, then forks), and the number of knives and forks is equivalent, meaning that acquiring a knife guarantees that a fork can be acquired. Thus, there is no circular waiting while holding, which violates one of the four preconditions for deadlock to occur.

-5 for saying yes and not considering the order of picking up a knife before fork.
-4 for saying yes, but acknowledging the picking order has some sort of effect.
-4 for saying no, but for wrong reasons (i.e., require picking up a knife and fork atomically).
-2 for messing up the order (i.e., fork before knife).

c. (8 points) Implement the `Dine()` method using *only* semaphores (your solution may not use locks, monitors, or other synchronization primitives). Create a method `Dine()`, which waits until a student has fork and knife and can eat, then calls `Eat()`, and then releases the utensils before returning. Your solution should allow multiple students to eat at the same time (as long as there are sufficient utensils in a pile in the middle of the table). Assume you are given the variables, `forks` and `knifes`, which each starts off initialized to the total number of forks and knives available, respectively. *Based on your answer from part 3.b, your solution should avoid deadlock.*

The key insight is to set the initial value of the semaphore to the minimum of the total number of forks or knives. Then, each P operation effectively grabs a knife and fork and each V operation relinquishes a knife and fork.

```
Semaphore pairs = new Semaphore(min(forks, knives));

Dine() {
  pairs.P();            // This "acquires" a knife and fork
  Eat();
  pairs.V();            // This "releases" a knife and fork
}
```

-8 pts: using other synchronization primitives (i.e., locks, monitors, CVs, interrupts)
-7 pts: didn't enforce correctness constraint (need knife and fork to eat).
-7 pts: trying to use a semaphore in an illegal way (e.g., trying to check the value)
-7 pts: non-working solutions (i.e. non-synchronized code).
-6 pts: only one graduate student can eat at a time.
-5 pts: no call to Eat()
-5 pts: deadlock possible
-5 pts: busy waiting
-5 pts: solution fails if pairs is initially set to some value
-2 pts: messing up the order (i.e., fork before knife).
-1 pts: missing semaphore initializations or values
-1 pts: incorrect semaphore init values (i.e., some people did "Semaphore utensilPairs(floor(fork, knife)/2)")
-1 pt: extraneous/redundant semaphores
-1 pt: didn't show initialization of semaphore value
-5 – 7 pts: incorrect use of semaphores. Many students defined integer variables to explicitly keep track of the number of available knives and forks, and used an "if" or "while" statement to look up the variable values – this solution is basically trying to implement condition variables using semaphore. Some solutions come close to successfully implement a working condition variable using semaphore. However, we took off points because this shows a lack of understanding of semaphores. In particular, a semaphore already provides these counters implicitly. We took off 5 to 7 points depending on how badly semaphores were misused.

Another solution:

```
Int forks      = N;        // Total number of forks
Int knives     = N;        // Total number of knives
Semaphore Lock = new Semaphore (1);    // mutex
Semaphore SemF = new Semaphore (forks);  // scheduling
Semaphore SemK = new Semaphore (knives);  // scheduling
Dine {
  Lock.P();           // This "acquires" mutex
  SemK.P();           // This "acquires" one knife
  SemF.P();           // This "acquires" one fork
  Lock.V();           // This "releases" mutex
  Eat();
  SemF.V();           // This "releases" one fork
  SemK.V();           // This "releases" one knife

}
```

4. (27 points total) Operating Systems.

a. (6 points) What are the two roles of an Operating System. For each role, briefly (one to two sentences) explain the role.

i)

Coordinator and Traffic Cop – The OS manages all resources, settles conflicting requests for resources, and prevent errors and improper use of the computer.

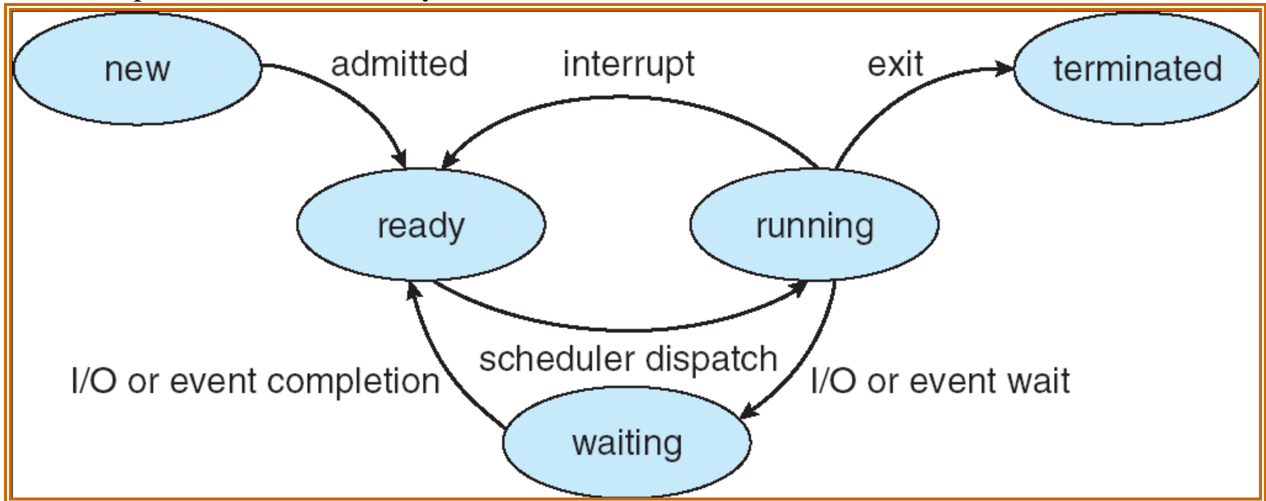
ii)

Facilitator – The OS provides facilities that everyone needs, such as standard libraries, windowing systems, etc., and helps make application programming easier, faster, and less error-prone.
Also, OS as Illusionist, where the OS hides hardware constraints and gives the illusion of a dedicated machine.
We deducted 1 to 3 points depending on how much the answer for part (ii) overlapped the answer for part (i).

b. (5 points) What private (thread-private) and public (shared between cooperating threads) resources are allocated when a thread is created? How do they differ from those allocated when a process is created?

A thread is allocated a private Thread Control Block which holds a register set, including Program Counter and Stack Pointer (for some architectures, a Frame Pointer), and it is allocated memory for a stack. The thread is also has access to shared memory within an address space, containing code, initialized data, uninitialized data (heap), and file descriptors.
A process is allocated a Process Control Block and an address space containing code, initialized data, uninitialized data (heap), file descriptors, and space for threads.
We deducted ½ point for each error in how resources are shared or allocated

c. (10 points) Draw the life cycle of a thread and label all the transitions:



We deducted one point for each missing state or transition.

d. (6 points) We discussed the Therac-25 in lecture and in an assigned reading.

i) What is the Therac-25, and how does it differ from the Therac-20?

The Therac-25 is a machine for radiation therapy that produces electron beam and X-ray radiation. It differs from the Therac-20 in that it uses only software control and safety interlocks in the electron accelerator and electron beam/X-ray production process, and software control of dosage. The Therac-20 included hardware safety interlocks.

ii) Explain the Therac-25's problems and the underlying causes.

Software errors caused the deaths and injuries of several patients. There were a series of race conditions on shared variables and poor software design that lead to the machine's malfunction under certain conditions.

No Credit – **Problem X** (000000000000 points)

A 2007 Darwin Award Runner Up

> "Named in honor of Charles Darwin, the father of evolution, the Darwin Awards commemorate those who improve our gene pool by accidentally removing themselves from it." (http://www.darwinawards.com/)

The Laptop Still Works!

[Confirmed True]

(26 February 2007, California) 29-year-old Oscar was driving on Highway 99 near Yuba City, when his Honda Accord crossed into oncoming traffic and collided with a Hummer. The occupants of the Hummer were not seriously injured. California Highway Patrol officers found Oscar's laptop still running, and plugged into the car's cigarette lighter. Investigators believe that he was using it when his car crossed the center line.

"Driving is not a time to be practicing your multitasking skills," remarked CHP spokesman Tom Marshall.

Oscar is not alone. Last year, 510 California drivers were charged with reckless driving because they were using a TV, video, or computer monitor. A 2001 CHP study cites cell phone use as the top cause of crashes involving distracted drivers, followed by fiddling with music. "Anything that distracts you can kill you, whether it's eating lunch or working on a computer," an AAA spokesman said.

Oscar was a computer tutor. Hopefully his fatal lesson will teach others to surf on the information superhighway, not the asphalt superhighway.

This page intentionally left blank