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

Spring 2000 Prof. Michael J. Franklin

Midterm Exam #1

SOLUTIONS

February 28, 2000

CS 162 Operating Systems

Your Name:	Anne Cerky
SID and 162 Login:	123456789 cs162zz
TA Name:	E Djikstra
Discussion Section:	Section 13 - Sunday 7:30 am

General Information:

This is a **closed book** examination. You have 1 hour and 20 minutes to answer as many questions as possible. Partial credit will be given. 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 than others.

Write all of your answers directly on this paper. Be sure to **clearly indicate** your final answer for each question. Also, be sure to state any assumptions that you are making in your answers.

Please try to be as **concise as possible**.

GOOD LUCK!!!

Problem	Possible	Score
1. Thread Basics (3 parts)	**20**	**20**
2. Synchronization (5 parts)	**35**	**35**
3. Using Semaphores (3 parts)	**15**	**15**
4. Deadlock (3 parts)	**15**	**15**
5. Atomicity (2 parts)	**15**	**15**
TOTAL	**100**	**100!**

Question 1 [3 parts, 20 points total]: Thread Basics

(Assume Mesa-style monitors throughout unless explicitly stated otherwise)

a) (4 points) Because personal computers were intended to be used by a single person at a time, early PC operating systems such as MSDOS provided only a single thread of control and a single address space. Even assuming only one person uses the computer at a time, **briefly** state a problem or limitation resulting from this approach.

The two most popular correct answers were:

1. Can't do more than one task at a time which means either users have to wait for one task to finish before working on another (e.g., can't read email while printing) or resources would be used inefficiently (can't overlap I/O and CPU).

2. Single address space leads to protection problems --- user program can overwrite parts of the operating system.

Partial credit was given if no implication was stated.

b) (10 points) Given the three thread states: *running*, *runnable (i.e., ready)*, and *waiting*, state which of the six possible thread transitions are allowed **and** give an example of why a thread would follow such a transition.

Runnable (Ready) → Running; //Chosen to run by OS Scheduler

Running → Runnable (Ready); //Time-sliced out or called Yield()

Running→ Waiting; // Requests I/O, lock, or wait on another thread

Waiting → Runnable (Ready) //Event completion

1 point for each correct transition plus 1 point for each correct example. 1 Point each for not listing the two illegal transitions.

c) (6 points) What state information do you need to save/restore about threads when performing a context switch?

Program Counter, Registers, and Stack (2 points each)

-1 point if said "stack" instead of stack pointer; -2 points if listed both stack and SP; -2 points for additional unnecessary information (that does not need to be restored).

Question 2 [5 parts, 35 points total]: Synchronization

a) (5 points) Consider the following implementation of Locks:

```
Lock::Acquire() {disable_interrupts();}
Lock::Release() {enable_interrupts();}
```

For a **single-processor system** state whether this implementation is: *i)* correct and reasonable (i.e., is a good way to do it), *ii)* incorrect (i.e., it may produce incorrect executions), or *iii)* dangerous (i.e., it may work but it could cause problems in practice). **Briefly, but explicitly justify your answer.**

2 pts if answered Incorrect or Dangerous. 3 pts for a good reason; some examples:

- *does not work for multiple locks (enable_ints vs restore_ints)*
- *user program may not release for a long time.. effectively halts machine*
- *does not maintain the "acquired" state of the lock across a context switch (yield())*

b) (5 points) Answer the question in part "a" above, but this time for a **multi-processor system.** Again, briefly but explicitly justify your answer.

2 pts for Incorrect / Dangerous; 3 pts for a reason (any of the above, or:)

does not block other processors from accessing the critical section (if they said this would halt all processors without EXPLICITLY stating that disable interrupts affected every processor, they got no credit)

c) (10 points) Consider the following two threads, to be run concurrently in a shared memory (all variables are shared between the two threads).

Thread A	Thread B
`for i = 1 to 5 do` `  x = x + 1;`	`for j = 1 to 5 do` `  x = x + 1;`

Assuming a single-processor system, that load and store are atomic, that *x* is initialized to 0, and that *x* must be loaded into a register before being incremented (and stored back to memory afterwards), what are **all the possible values** for *x* after both threads have completed?

The correct answer is 2 through 10 inclusive. Grading was assigned as follows:

4 pts for 2, 3, 4

2 pts for 5

3 pts for 6, 7, 8

1 pt for 10

-1 pt for each extra number

-8 for listing all values, 1 through 10

d) (10 points) Show in pseudocode how to implement semaphores (just the P() and V() operations) using mesa-style monitors.

Solution:

```
P() {
    mutex.acquire();      //2 pts for correct acquire / release
    while (count == 0) {  //2 pts for while loop / cond. var.
        c.wait();
    }
    count--;              //1 pt for decrementing counter
    mutex.release();
}
V() {
    mutex.acquire();      //2 pts for correct acquire / releas
    count++;              //1 pt for incrementing counter
    c.signal();           //2 pts for signalling
    mutex.release();
}
```

No credit was given if monitors were not used as that was specifically required in the question. Any correct solution was given full credit.

e) (5 points) Recall that semaphores can be used to implement mutual exclusion or thread scheduling dependencies. Show in pseudocode how a semaphore can be used to implement the join operation on a thread. Be sure to indicate the initial value of the semaphore.

```
Thread::Thread() {
  …
  if (joinable) {
    _______________   //Your code goes here
  }
  …
}

void Thread::Join() {
  ASSERT(joinable);
  _____________   //Your code goes here
  delete this;
}

void Thread::Finish() {
  kernel->interrupt->SetLevel(IntOff);
  __________  //Your code goes here
  if (joinable)
  Sleep(FALSE);
else
  Sleep(TRUE);
}
```

First missing statement (2 pts) is "sema = 0" / must initialize the semaphore correctly

Second missing statement (1.5 pts) is "sema.P() / sema.wait()"

Thirds missing statement (1.5 pts) is "sema.V() / sema.signal()"

Question 3 [3 parts, 15 points total]: Using Semaphores

In class we discussed a solution to the bounded-buffer problem for a Coke machine using three semaphores (mutex, emptyBuffers, and fullBuffers):

```
Producer () {
     emptyBuffers.P();
     mutex.P();
     put 1 coke in machine;
     mutex.V();
     fullBuffers.V();
}
```

```
Consumer() {
     fullBuffers.P();
     mutex.P();
     take 1 coke from machine;
     mutex.V();
     emptyBuffers.V();
}
```

Given each of the following variations, say whether it is correct or incorrect. If you say correct, **explain any of the advantages and disadvantages of the new code**. If you say incorrect, **explain what could go wrong** (i.e., trace through an example where it does not behave properly).

The entire problem was worth 15 points Each of the 3 parts was out of 5 points:

5 points if for getting whether the code is correct/incorrect and with the appropriate justification/explanation. 3 points if for getting whether the code is correct/incorrect but without good explanation. 0 points otherwise.

a) (5 points)

```
Producer () {
     mutex.P();
     emptyBuffers.P();
     put 1 coke in machine;
     fullBuffers.V();
     mutex.V();
}
```

```
Consumer () {
     mutex.P();
     fullBuffers.P();
     take 1 coke from machine;
     emptyBuffers.V();
     mutex.V();
}
```

This code is incorrect. It can lead to deadlock. Consider the case where the coke machine is initially full. Suppose a Producer comes and grabs the mutex and then waits for emptyBuffers. A consumer then hangs on mutex and no one will ever consume a coke to empty a buffer. An analogous example is the case

where the coke machine is initially empty and a Consumer grabs the mutex and waits for fullBuffers.

b) (5 points)

Producer () {	Consumer() {
mutex.P();	fullBuffers.P();
emptyBuffers.P();	mutex.P();
put 1 coke in machine;	take 1 coke from machine;
fullBuffers.V();	mutex.V();
mutex.V();	emptyBuffers.V();
}	}

This code is incorrect. It can lead to deadlock. The problem is exactly as the first case of deadlock mentioned in part a. Consider the case where the coke machine is initially full. Suppose a Producer comes and grabs the mutex and then waits for emptyBuffers. A consumer then hangs on mutex and no one will ever consume a coke to empty a buffer.

c) (5 points)

Producer () {	Consumer() {
emptyBuffers.P();	fullBuffers.P();
mutex.P();	mutex.P();
put 1 coke in machine;	take 1 coke from machine;
fullBuffers.V();	emptyBuffers.V();
mutex.V();	mutex.V();
}	}

This code is correct. As mentioned in lecture, this code allows more concurrency which is the advantage to coding in this manner. Since the mutex immediately surrounds both the Producer and Consumer actions of putting a coke and taking a code from the machine, if we release the mutex quickly, we achieve better concurrency.

Question 4 [3 parts, 15 points total]: Deadlock

a) (5 points) Recall that there are four conditions that must hold in order for a deadlock to be possible. Name one of these conditions that is always present when using synchronization primitives for **mutual exclusion** and **state why the condition must hold.**

The limited access (mutual exclusion) condition must hold. This is by definition since limited access must be present for mutual exclusion to be achieved.

2 points for a correct deadlock condition

3 additional points for a correct explanation

If you said no preemption, you only lost 1 point for the correct deadlock condition.

b) (5 points) Name one of the deadlock conditions that is **not** necessarily present when using synchronization primitives for mutual exclusion and describe a way that deadlock can be avoided by preventing that condition from occurring in a system where locks are the only resource that can deadlock.

2 points for a correct deadlock condition

3 additional points for a correct explanation

Circular wait, hold and wait, and no preemption (assuming you didn't say this for part a) were all correct answers.

2 points for a correct deadlock condition

3 additional points for a correct explanation

For example, if you said circular wait, you could say that you use resource ordering to prevent this circular wait condition.

c) (5 points) Dining Aliens --- CS 162 meets the X-files

Consider a scenario where all philosophers have been mysteriously replaced by multi-armed beings from another planet. Initially there are *m* chopsticks in the middle of the table. An alien must pick up *n* chopsticks (where *n* is specified by each alien as an argument and may change each time that alien has dinner) before it can eat. After eating the alien puts the chopsticks back down. Describe a rule for taking chopsticks that avoids deadlock but allows concurrency and allows aliens to acquire their chopsticks incrementally.

*One answer is that we can use the modified Banker's Algorithm as presented in class. The rule is that we can allocate the chopstick if the total number of chopsticks - those chopsticks in use >= the number of chopsticks still needed by the alien. In other words, if the number of chopsticks in the middle of the table is greater than or equal to the number of chopsticks an alien *still* needs, then the alien is granted a chopstick.*

Grading was done as follows:

5 points for a plausible answer

3 if answer was incomplete/incorrect but had a glimmer of hope

1 not incremental solution, or any other attempt at a solution

0 no effort made

It is possible to score between these numbers if we felt your answer was in between two of these categories.

Question 5 [2 part, 15 points total]: Atomicity

a) (10 points) The original Nachos implementation of condition variables relies on semaphores. Recall that one of the requirements of Condition::Wait is that it atomically release the lock and wait on the semaphore. Is the following implementation of condition variables correct? If so, explain why Wait is effectively atomic. If not, state the smallest change you could make to make it atomic.

```
class Condition {
      Lock* conditionLock;
      List<Semaphore*>* waitQueue;
      void Wait() {
          waiter = new Semaphore(0);
          waitQueue->Append(waiter);
          conditionLock->Release();
          waiter->P();
          conditionLock->Acquire();
          delete waiter;
      }
      void Signal() {
          if (!waitQueue->IsEmpty())
             waitQueue->RemoveFront()->V();
      }
  };
```

This implementation is correct. Even if a context switch occurs after releasing the lock before waiting on the semaphore, the semaphore is already on the wait queue, so if a signal occurs, the semaphore will be V()'d. Then when the waiting thread callsP(), it will not actually sleep; i.e. it still caught the signal.

4 points for saying correct. 6 points for good explanation. 2 points for explanation on the right track, but missing key points about the semaphore wait queue. 0 points for explanations that do not talk about context switches between release() and P(). 5 points for saying not atomic but placing 1 disable-ints/restore-ints pair in wait(); -1 point for each additional pair.

b) (5 points) Why is it dangerous to acquire a lock in an interrupt handler?

The key thing to note here is that interrupts can occur in ANY thread. When a processor receives an interrupt, and interrupts are enabled, the current thread makes a forced procedure call to the interrupt handler. If an interrupt handler blocks while trying to acquire a lock, it could be forcing a high priority thread to sleep waiting for a completely unrelated thread. Worse yet, the current thread might already hold the lock, in which case you get a single-thread deadlock.

A lot of people said waiting in interrupt handlers is bad. This is true, but waiting a bit isn't as bad as blocking an arbitrary thread. It just results in inefficiency, and perhaps makes the associated I/O device unusable. Note that disabling interrupts does NOT prevent context switches. For example, in Nachos, Thread::Sleep() is always called with interrupts disabled, but that doesn't stop it from giving up the processor to another thread. Also note that each thread tracks its own interrupt state, so that when it gets to run, it sets interrupts to the state it wants (this is done in Thread::Yield and in Semaphore::P and ::V, for example). Disabling interrupts and context switching does not force the next thread to run with interrupts disabled

5 points were given for the following kinds of answers:

-lock already held => deadlock / could wait forever

-lock already held, handler sleeps => could get another instance of handler, atomicity lost

2 points for these:

-lock already held, another interrupt happens, deadlock

-interrupts off => deadlock

-lock never gets released => deadlock

-lock already held => interrupt handler waits/gets put to sleep

-lock already held by current thread => defeats purpose of interrupt handler

And no points for these (most of the class):

-might forget to release lock

-lock::acquire disables interrupts => interrupt handler stops

-lock already held and interrupts off => can't context switch

-lock already held, another interrupt might happen

-interrupts handled in kernel mode, security problem