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

Fall 1999 Anthony D. Joseph

Midterm Exam #1 *Solutions*

September 29, 1999
CS162 Operating Systems

Your Name:	
SID and 162 Login:	
TA:	
Discussion Section:	

General Information:
This is a **closed book** examination. You have two hours 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	12	
2	15	
3	9	
4	24	
5	30	
6	10	
Total	**100**	

1. Threads – Good or bad? (12 points total):
   a. (6 points) List two reasons why ***overuse*** of threads is bad (i.e., using too many threads for different tasks). Be explicit in your answers.
      i)

      3 points. "Threads are cheap, but they're not free." Too many threads will cause a system to spend lots of time context switching and not doing useful work.

      Each thread requires memory for stack space and TCBs. Too many threads and these memory uses will dominate overall memory use.

      ii)

      3 points. Having many threads makes synchronization more complicated as you have more and more simultaneous tasks. Also, debugging is more difficult due to non-deterministic and non-reproducible execution.

      We did not give credit for answers that were scheduling issues (e.g., too many threads could cause starvation), since these issues also affect processes.

   b. (6 points) List two reasons why threads are useful/important.
      i)

      3 points for each of two reasons. Many reasons, including:
      - *More efficient operation: overlap I/O and computation*
      - *Easier to share data between threads in the same address space than between processes*
      - *Modularity: A complex task can be decomposed into smaller pieces*
      - *Reduced latency on multiprocessors*
      - *Faster to switch between threads than between processes*

      We did not give credit for saying that threads give you concurrency, since processes also provide concurrency.

      ii)

2. (15 points) For each of the following thread state transitions, say whether the transition is legal ***and*** how the transition occurs or why it cannot. Assume Mesa-style monitors.

   a. Change from thread state **BLOCKED** to thread state **RUNNING**.

   5 points, 3 for correct answer and 2 for justification. Illegal. Threads must first be inserted into the ready queue (i.e., moved to the RUNNABLE state) before they can be executed (moved to the RUNNING state). With Mesa-style monitors, threads are placed onto the ready queue, while with Hoare-style monitors, they are immediately given the CPU (moved to the RUNNING state).

   b. Change from thread state **RUNNING** to thread state **BLOCKED**.

   5 points, 3 for correct answer and 2 for justification. Legal. A thread that sleeps, joins with another thread, or waits on I/O will transition from the RUNNING state to the BLOCKED state.

   c. Change from thread state **RUNNABLE** to thread state **BLOCKED**.

   5 points, 3 for correct answer and 2 for justification. Illegal. A thread can't change from the RUNNABLE state to the BLOCKED state without first being scheduled and taking an action that causes it to block (e.g., the actions listed in b).

3. (9 points) You're hired by AB Computers to improve the performance of their system. They point out that their applications only use 10 of the CPU's 32 registers; so to improve the performance of the applications, they suggest that you change the OS's context switch routine so it only saves the 10 registers used by the applications. Assume that you can correctly change the context switch routine. Is this a good or bad idea? Why?

   This is a very bad idea (correct answer 3 points). Up to 6 points for justification. Examples: Long-term implications, such as problems with changes to applications or the use of future applications and the intermittent problems that will result; also, issues with legacy applications.

   We subtracted 3 points for justifications that only focused on the operating system and ignored the long-term issues.

4. (24 points) Concurrency control:

You've been hired by the HAL computer company to review their code for a large application. In the application you find the **atomic swap** procedure. Say whether it either (i) works, (ii) doesn't work, or (iii) is dangerous that is, sometimes works and sometimes doesn't. If the implementation does not work or is dangerous, explain why (there may be several errors) and rewrite the code to fix it so it does work.

The problem statement is as follows: The **atomic swap** routine pops an item from each of two stacks and pushes the items onto the opposite stack. If either stack does not contain an item, the swap fails and the stacks are left as before the swap was attempted. The swap must appear to occur atomically: there should be no interval of time during which an external thread can determine that an item has been removed from one stack but not yet pushed onto another one. In addition, the implementation must be highly concurrent – it must allow multiple swaps between unrelated stacks to happen in parallel. You may assume that stack1 and stack2 never refer to the same stack.

```
void AtomicSwap (Stack *stack1, *stack2) {
       Item thing1; /* First thing being transferred */
       Item thing2; /* Second thing being transferred */

       stack1->lock.Acquire();
       thing1 = stack1->Pop();
       if (thing1 != NULL) {
              stack2->lock.Acquire();
              thing2 = stack2->Pop();
              if (thing2 != NULL) {
                     stack2->Push(thing1);
                     stack1->Push(thing2);
                     stack2->lock.Release();
                     stack1->lock.Release();
              }
       }
}
```

This routine is (iii) dangerous (correct answer worth 3 points) for three reasons (each reason is worth 3 points):

1. *Deadlock could occur because there is no resource ordering during lock acquisition.*
2. *The routine fails to release the locks if either stack is empty.*
3. *The routine does not leave stack1 unchanged if stack two is empty (it fails to repush thing1).*

We subtracted 3 points if you claimed that the routine is not atomic – an outside application cannot see whether something has been removed from one stack and not yet pushed onto another stack (this includes the error that fails to repush

thing1. It does not release the lock on stack1, so the error is not externally visible).

(Additional space for question 4)

The code solution is worth 12 points total: 3 points for fixing each error and 3 points for the overall implementation quality (if deadlock is fixed). We subtracted 6 points for solutions that used a global lock (where the lock was held for the entire swap routine) and subtracted 3 points for solutions that used a global lock around lock acquisition (where the lock was only held while each lock was acquired).

```
void AtomicSwap (Stack *stack1, *stack2) {
        Item thing1; /* First thing being transferred */
        Item thing2; /* Second thing being transferred */

        if (stack1 > stack2) {
                stack1->lock.Acquire();
                stack2->lock.Acquire();
        } else {
                stack2->lock.Acquire();
                stack1->lock.Acquire();
        }
        thing1 = stack1->Pop();
        if (thing1 != NULL) {
                thing2 = stack2->Pop();
                if (thing2 != NULL) {
                        stack2->Push(thing1);
                        stack1->Push(thing2);
                } else {
                        stack1->Push(thing1);
                }
        }
        stack2->lock.Release();
        stack1->lock.Release();
}
```

5. Concurrency problem (30 points total):

A particular river crossing is shared by both Linux Hackers and Microsoft employees. A boat is used to cross the river, but it only seats **four** people, and must always carry a full load. In order to guarantee the safety of the hackers, you cannot put three employees and one hacker in the same boat (because the employees would gang up and convert the hacker). Similarly, you cannot put three hackers in the same boat as an employee (because the hackers would gang up and convert the employee). All other combinations are safe.

Two procedures are needed: *HackerArrives* and *EmployeeArrives*, called by a hacker or employee when he/she arrives at the river bank. The procedures arrange the arriving hackers and employees into safe boatloads; once the boat is full, one thread calls *Rowboat* and only after the call to *Rowboat*, the four threads representing the people in the boat can return.

You can use either semaphores or condition variables to implement the solution. Any order is acceptable and there should be no busy-waiting and no undue waiting – hackers and employees should not wait if there are enough of them for a safe boatload. *Your code should be clearly commented, in particular, you should comment each semaphore or condition variable operation to specify how correctness properties are preserved.*

a. Specify the correctness properties for your solution:

1. *People enter boats either based upon the number already in the boat (resulting in undue waiting) or they enter the boat in groups of two or four.*
2. *The boat is launched when it contains four people (this action is done by one of the people threads).*
3. *Only one person (or representative of two or four people) can check the state of the boat (i.e., the number of people).*

We awarded five points for the correctness properties (a base of two points plus one point for each correctness property). We subtracted one point for each property that was a performance and not a correctness property (e.g., saying that there should not be undue waiting

b. Your solution:

The solution is worth 25 points. An entirely wrong solution was awarded no points.

- *We subtracted 8 points if there were no comments, four points if the comments were weak or useless.*
- *For solutions that did not use semaphores or condition variables, we subtracted 15 points.*

(Additional space for question 5)

- *We subtracted 2 points for each minor error, up to 8 points.*
  - *Not defining or initializing variables*
  - *Gratuitous checks. For example, checking if the boat contained four people, even if the condition is never visible to other threads.*
- *We subtracted 4 points for each major error, up to 12 points:*
  - *Signaling/waiting outside a critical region (or using a semaphore inside one)*
  - *Busy waiting*
  - *Failing to wait for Rowboat to be called before a thread returns*
  - *A non-symmetric solution where the HackerArrives and EmployeeArrives procedures were different*
  - *Undue waiting caused by loading one person at a time instead of two or four: loading one at a time is bad because you could have a string of three hackers followed by a long string of employees, all of whom would be forced to wait for another hacker. Loading two or four at a time would avoid such a situation.*

```
int WH = 0, WE = 0;          // shared variables tracking waiting people
Lock mutex = FREE;           // Lock to enforce correctness property
Condition Hacker;            // Used to wait for enough people to cross river
Condition Employee           // Same as Hacker

void HackerArrives() {
        mutex->Acquire();    // Enter the critical section so we can check state vars
        if (WH == 3) {       // We've already got three waiting hackers. Lets go!
                Hacker->signal(mutex);       // Wake three hackers (any three)
                Hacker->signal(mutex);
                Hacker->signal(mutex);
                WH -= 3;     // Decrement state vars (this must be done here, not later)
                Rowboat();   // Cross the river
        } else if ((WH >= 1) && (WE >= 2) {        // 1 other hacker, 2 employees
                Hacker->signal(mutex);       // Wake up one hacker, two employees
                Employee->signal(mutex);
                Employee->signal(mutex);
                WH - -;      // Decrement state vars
                WE -= 2;
                Rowboat();
        } else {
                WH++;        // Wait for more Hackers to arrive
                Hacker->wait(mutex);         // No need to check state vars
        }
        mutex->Release();    // Done with critical section
}
```

The solution for EmployeeArrives is symmetric.

Although condition variables are the preferred mechanism for solving the problem, semaphores can also be used:

```
Semaphore Hackers = 0;          // Start with zero hackers
Semaphore Employees = 0;        // Start with zero employees
int HackerCount = 0;            // Count of waiting hackers
int EmployeeCount = 0;          // Count of waiting employees
Semaphore Mutex = 1;            // Semaphore for critical region (checking counts)
Semaphore Rowing =0;            // Semaphore to prevent riders from returning from
                                //   Hacker/Employee Arrival call until Rowboat call

void HackerArrives() {
        Mutex.P();              // Acquire lock for rider variables
        if (HackerCount == 3) { // Three other waiting hackers
                HackerCount –= 3;  // Decrement count of waiters
                Mutex.V();         // Release lock
                Hackers.V();       // Wake up three other waiting hackers
                Hackers.V();
                Hackers.V();
        } else if ((HackerCount >= 1) && (EmployeeCount >=2)) {
                HackerCount –= 1   // Decrement count of waiters;
                EmployeeCount –= 2;
                Mutex.V();         // Release lock
                Hackers.V();
                Employees.V();
                Employees.V();
        } else {
                HackerCount += 1;  // New waiting hacker
                Mutex.V();         // Release lock
                Hackers.P();       // Go to sleep until other riders arrive to fill boat
                Rowing.P();        // Wait for Rowboat, once we get in the boat
                return;
        }
        // Only the rider that fills the boat (didn't sleep) makes it to this point
        RowBoat();
        Rowing.V();             // Wake up waiting boat occupants
        Rowing.V();
        Rowing.V();
}
```

The solution for EmployeesArrive is symmetric.

No Credit – Problem X: (000000000000 points)

Runner up for the 1999 Darwin Awards (*don't try this at home*)

In rural Carbon County, PA, a group of men were drinking beer and discharging firearms from the rear deck of a home owned by Irving Michaels, age 27. The men were firing at a raccoon that was wandering by, but the beer apparently impaired their aim and, despite the estimated 35 shots the group fired, the animal escaped into a 3 foot diameter drainage pipe some 100 feet away from Mr .Michaels' deck.

Determined to terminate the animal, Mr. Michaels retrieved a can of gasoline and poured some down the pipe, intending to smoke the animal out. After several unsuccessful attempts to ignite the fuel, Michaels emptied the entire 5-gallon fuel can down the pipe and tried to ignite it again, to no avail. Not one to admit defeat by wildlife, the determined Mr. Michaels proceeded to slide feet-first approximately 15 feet down the sloping pipe to toss the match.

The subsequent rapidly expanding fireball propelled Mr. Michaels back the way he had come, although at a much higher rate of speed. He exited the angled pipe "like a Polaris missile leaves a submarine," according to witness Joseph McFadden, 31. Mr. Michaels was launched directly over his own home, right over the heads of his astonished friends, onto his front lawn. In all, he traveled over 200 feet through the air. "There was a Doppler Effect to his scream as he flew over us," McFadden reported, "followed by a loud thud." Amazingly, he suffered only minor injuries. "It was actually pretty cool," Michaels said, "Like when they shoot someone out of a cannon at the circus. I'd do it again if I was sure I wouldn't get hurt."

6. (10 points) Deadlock:

Consider a computer system that runs 5000 jobs per month with no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice a month, and the operator (a salaried employee) must terminate and rerun about 10 jobs per deadlock. The operator notices immediately when deadlock has occurred because their screen shows the computer's running processes. Each job is worth about $2 (in CPU time), and the jobs terminated tend to be about half-done when they are aborted.

A system programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase in the average execution time per job of about 10 percent. Since the machine currently has a 30 percent idle time, all 5000 jobs per month could still be run.

Should the company install the algorithm? List the reasons why or why not.

The answer to this question was subjective; as such, we looked for answers that showed that you thought through your decision.

Arguments against the algorithm:

1. *The cost of deadlocks is:*
   $2/job x 10 jobs x ½ complete = $20/month
   versus the cost of additional CPU time used the deadlock-avoidance algorithm:
   10% of 5000 jobs x $2/job = $1000/month
2. *There is less available idle time for future use when the deadlock-avoidance algorithm is used.*
3. *Finally, the algorithm delays the common case for only a few deadlocks.*

Arguments in favor of the algorithm:

1. *More reliable execution times (since no jobs are aborted and re-run).*
2. *Less need for the operator (the operator could be assigned additional / alternate tasks)*
3. *Running more jobs in the future may result in more deadlocks.*

Overall, the arguments against the algorithm are strongest.

We awarded 5 points if you provided a reasonable argument (e.g., providing an answer with a few of the listed arguments). We awarded more or less points based upon the number and strengths of your arguments.

(This page intentionally left blank)