University of Rochester **Computer Systems (CSC2/456)**

Midterm Exam

Amy Murphy

6 March 2002

Read before beginning: Please write clearly. Illegible answers cannot be graded. Be sure to identify all of your answers in your blue book (e.g., 1a, 2b, etc). All questions are worth 10 points, numbers in parenthesis indicate relative point values.

While I have tried to make the questions as unambiguous as possible, if you need to make any assumptions in order to continue, state these as clearly as possible as part of your answer. In the name of fairness, I would like to avoid answering questions during the exam.

CSC256: Students enrolled in 256 must choose **6 of the 8** questions to answer. For extra credit, you may answer the other 2 questions, indicating clearly on the front of the bluebook which questions are to be counted as extra credit. If you do not specify, I will assume the first 6 in your bluebook are for *normal* credit and the remaining 2 (if any) are *extra*. Grades will be assigned 0-60.

CSC456: Students enrolled in 456 must choose **7 of the 8** questions to answer. For extra credit, you may answer the 8th question, but you must clearly identify which question is to be counted as *extra*. If you do not indicate any, I will assume the 8th answer (if any) is *extra*. Grades will be assigned 0-70.

1. **Short answer.** Your responses for each part should be at most 3-4 lines of text:

   (a) (3) Operating systems make the distinction between between user level operations and kernel level operations. Decide whether the following should be user or kernel, and briefly justify your answer: (i) Disable all interrupts, (ii) read the time of day clock, (iii) set the time of day clock, (iv) kill a process.

   (b) (3) To a programmer, a system call looks like any other call to a library procedure. Is it important that a programmer know which library procedures result in system calls? Why or why not?

   (c) (4) Polling a device for the completion of an operation is typically a bad idea because while polling, the CPU is not doing useful work. Nonetheless, this is not always true. Describe a circumstance when polling might be better than interrupts.

2. **Processes and threads.** Suppose that three processes exist in a system, as described table below. Suppose that the system uses preemptive, round-robin scheduling, and that T_{11} is running when the quantum expires.

Process	Threads within the process
P_1	T_{11}, T_{12}, T_{13}
P_2	T_{21}, T_{22}
P_3	T_{31}

   (a) (4) If the threads are implemented entirely at the user level (with no support from the operating system), which threads might possibly execute at the beginning of the next quantum?

   (b) (4) If threads are supported by the operating system (i.e., lightweight processes), which threads might possibly execute at the beginning of the next quantum?

   (c) (2) How does your answer change if the system has two processors?

3. **Device management.** Network cards are just one of the many devices which the operating system must manage. For this problem ignore all other devices and assume infinite bus bandwidth. Consider a video stream that sends a unicast, 100 MB/sec stream to as many machines as possible. Under each of the following conditions, what is the *maximum* number of destinations that can receive the stream? Explain your answer (this helps me give partial credit if you get the wrong number).

   (a) (3) Programmed I/O, 100MHz machine, 1GHz network.

   (b) (3) Programmed I/O, 10GHz machine, 1GHz network.

   (c) (4) DMA, 100MHz machine, 1GHz ntework.

4. **Scheduling.** Given the five processes below with their indicated number of run time units, answer the questions that follow. Assume processes arrive in numerical order at time 0.

Process ID	**CPU requirements**
P_1	5
P_2	4
P_3	2
P_4	1
P_5	8

   (a) (4) Show the scheduling order for these processes under first-come-first-served, shortest-job first, and round-robin scheduling (quantum = 2).

   (b) (4) For each process in each schedule above, indicate the wait time and turnaround time.

   (c) (2) Briefly explain why these two values are meaningful criteria for evaluating scheduling algorithms.

5. **Fair Scheduling.**

   (a) (3) Assume that when a process is scheduled, it always uses the full quantum which is assigned to it, and that the scheduler will only process interrupts at quantum boundaries. What does it mean for a scheduler to be perfectly fair? What does it mean for a scheduler to be probabilistically fair? (Caution: do not mix the terminology: probabilistic and proportional.)

   (b) (3) Suppose we have a lottery scheduler which choose the next process to run with equal probability (ignoring the number of tickets held by each process). However, instead of assigning all processes the same quantum size, the scheduler dynamically adjusts the quantum size to be proportional to the number of tickets held by each process. Processes with more tickets get longer quantums.

   Is the resulting schedule still probabilistically fair? Why or why not?

   (c) (4) Even if we relax the assumption made in part (a) to allow a process to yield the processor when it does I/O operations in order to not waste idle CPU cycles, why is the modification to the lottery scheduler in part (b) not always a good idea?

6. **Synchronization.** Tweedledum and Tweedledee are separate threads executing their respective procedures. The code below is intended to cause them to forever take turns exchanging insults through the shared variable X in strict alternation. The Sleep() and Wakeup() routines operate as follows: Sleep blocks the calling thread, and Wakeup unblocks a specific thread if that thread is blocked, otherwise it has no effect.

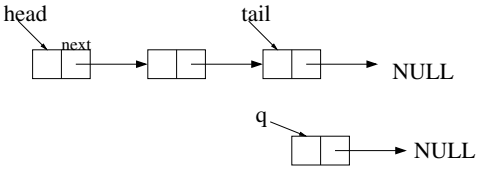
```
void Tweedledum() {
    while(1) {
        Sleep();
        x = Quarrel(x);
        Wakeup(Tweedledee thread);
    }
}
```

```
void Tweedledee() {
    while(1) {
        x = Quarrel(x);
        Wakeup(Tweedledum thread);
        Sleep();
    }
}
```

(a) (5) The code shown above exhibits a well-known synchronization flaw. Briefly outline a scenario in which this code would fail, and the outcome of that scenario.

(b) (5) Show how to fix the problem by removing the Sleep and Wakeup calls and instead using a binary semaphore.

7. **Lock free synchronization.** Suppose we want to use lock free synchronization mechanisms for managing a FIFO queue. Graphically, the queue is as shown in the figure below (ignore q for now). Basically, each node consists of some data and a next pointer. We also have global head and tail pointers. For simplicity, assume that we will ALWAYS have at least two nodes in the queue. In other words, head and tail will never be equal, and they will never be NULL.




```
dequeue()
    repeat
        p = head;
    until (CAS (head, p, p→next));
    return p;
```

(a) (3) The right hand side of the figure shows the lock-free dequeue operation which removes the element pointed to by head, and updates the head pointer. (Note the use of CAS: compare and swap. This atomic operation compares the first two parameters, if they are equal, it swaps the value of the first parameter with the third parameter and returns true. If the values are not equal, no swap is done, and false is returned.) What is *lock-free* about this dequeue operation? In other words, give a definition of the lock free approach. It may help to state one of the benefits of using lock-free synchronization mechanisms.

(b) (2) What happens to lock free synchronization mechanisms such as this one in environments with heavy contention for the resource?

(c) (5) Write the enqueue operation to enqueue node q (shown in the figure) to the tail of the queue. Hint: you need to update both the next pointer of the original tail node, as well as the pointer to tail.

8. **Deadlock prevention.** Consider the following allocation of a resource. Assume that there are a total of 10 instances of this resource.

Process	Num Used	Maximum
P_1	1	6
P_2	1	5
P_3	2	4
P_4	4	7

(a) (5) If P_4 requests one more instance of the resource, does this lead to a safe or unsafe state? Why?

(b) (5) What if the request for another resource comes from process P_3 instead? Why?