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.

   (i) kernel, although this is a basic way to do synchronization, if a user does it wrong, then the OS will stop functioning and has no way out. (ii) user, this is a read-only operation, and a user can do no harm. However, some OSs keep this in the kernel. (iii) kernel, if the user changed the time, other users would be adversely affected. (iv) kernel, users can't kill processes arbitrarily. They can kill their own processes, but the kernel needs to check to make sure that the parameter is indeed owned by the calling user.

   (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?

   Yes, it matters. Kernel calls are expensive operations, and can affect the running time of the application.

   (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.

   If a process is waiting for an operation which will occur soon, then it is not worth the ovehead to do the context switch involved in blocking on an interrupt. Generally, when the time to wait is less than the overhead time for the context switch, it is better to poll.

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?

T_{21}, T_{22} or T_{31} The OS is unaware of the threads within process P_1

(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?

any except T_{11}

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

It actually doesn't change. In the first case, the kernel will not know about the user's threads, so no other choices can be made. In the second case, the kernel will have full knowledge of the user's threads and will make the same decision (of course, realizing that the OS should not run the same thread on simultaneously on multiple 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.

If we assume the CPU can transfer memory at 1B on each CPU cycle, 1 host, the processor is the bottleneck.

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

10 hosts, the network bandwidth is the bottleneck

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

10 hosts, the network bandwidth is the bottleneck, if we assume the DMA can transfer memory at bus speeds.

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).

scheme	order	wait time					turnaround time				
		P_1	P_2	P_3	P_4	P_5	P_1	P_2	P_3	P_4	P_5
FCFS	P_1 P_2 P_3 P_4 P_5	0	5	9	11	12	5	9	11	12	20
SJF	P_4 P_3 P_2 P_1 P_5	7	3	1	0	12	12	7	3	1	20
RR	P_1 P_2 P_3 P_4 P_5 P_1 P_2 P_5 P_1 P_5 P_5	0	2	4	6	7	16	13	6	7	20

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

See above.

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

Wait time gives an indicates the time until the process gets its first chance on the CPU. This can indicate the time that a user has to wait to get even partial results. Turnaround shows how long the user must wait for complete results.

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.)

In a perfectly fair schedule, each process will get exactly the same number of CPU cycles as the other processes. In a probabilistically fair schedule, over time, processes will get the same amount of CPU, but the scheduling is subject to some random perturbation.

perfectly fair: each process gets equal time or time exactly proportional to its priority.

probabilistically fair: same as fair, but averaged over time; short-term unfairness is permissible.

(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?

*Yes, still fair, for exactly the same reason as lottery scheduling: the product of (frequency chosen) * (length of quantum run for) is the same in both cases. In particular, since each process is equally likely to be chosen, over time, higher priority processes will run for more total time in an amount proportional to their priority.)*

(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?

Interactive processes and processes with real-time (soft or hard) scheduling requirements can easily be disrupted by a CPU-bound process of even medium priority; the system will not feel responsive at all – especially with greater numbers of processes present.

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

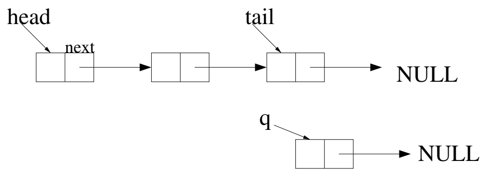
It is possible for Tweedledee to process up to, but not including its sleep. Tweedledum executes up to and including its sleep. Tweedledee is swapped back in and executes its sleep. The processes are deadlocked.

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

```
void Tweedledum() {
    while(1) {
        S.P();
        x = Quarrel(x);
        S.V();
    }
}
```

If we assume that both processes share the semaphore S, then Tweedledee will look the same as the Tweedledum shown above

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.

With lock free synchronization algorithms, if there is no contention, the process trying to perform the operation will never have to acquire/release a lock around the object being synchronized. Instead, the operation is optimistically executed, and if there is problem, the program will simply try again, hoping that the next time around, there is no contention.

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

The processes which fail will end up spinning a lot, trying to get their operation to succeed. This can be expensive both in terms of the CPU cycles spent doing redoing the operation, and any computation which must be done on each iteration through the loop.

(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.

```
enqueue()
    repeat
        p = tail;
    until (CAS (p→next, NULL, q));
    tail = p;
```

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?

This state is unsafe. After this allocation, there are not enough free resources to give the maximum number of resources to any of the processes.

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

This state is safe. A possible safe sequence of process execution is: P_3, P_4, P_1, P_2