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

Midterm Exam

Amy Murphy

19 March 2003

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 up to 10 extra credit points, you may answer **one** of the other 2 questions, indicating clearly on the front of the bluebook which question is 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 next one (if any) is *extra*. If you answer all 8 questions, only 7 of them will be counted toward your grade. Grades will be assigned 0-60 (with 10 possible extra credit).

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 (with 10 possible extra credit)

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

   (a) (3) What is the difference between deadlock and starvation?

   In a deadlock situation, none of the involved processes can possibly make progress. In a starvation situation, a process is ready to execute, but it is not being allowed to execute.

   (b) (3) What is the difference between a process and a thread?

   Every process has its own address space, but multiple threads inside a process share part of the memory with their parent process. There is less context switching overhead to switch among threads when compared to switching among threads.

   (c) (4) What are two types of low-level operations that higher-level synchronization operations (e.g., semaphores and monitors) can be built upon?

   test-and-set. compare-and-swap. atomic reads and writes. Other atomic operations. enabling and disabling interrupts.

2. **Device management.** You were just hired to work on a research project with a graphics professor. He has a program that takes an input image and searches through a bunch of images in a database in order to find matching images. It does not need to look at every image, but it must look at a large percentage of the images. Further, not all of the images will fit into memory, so the program spends most of its time doing disk read operations. You are given a single-threaded version of this program that works and it is your job is to make it run faster.

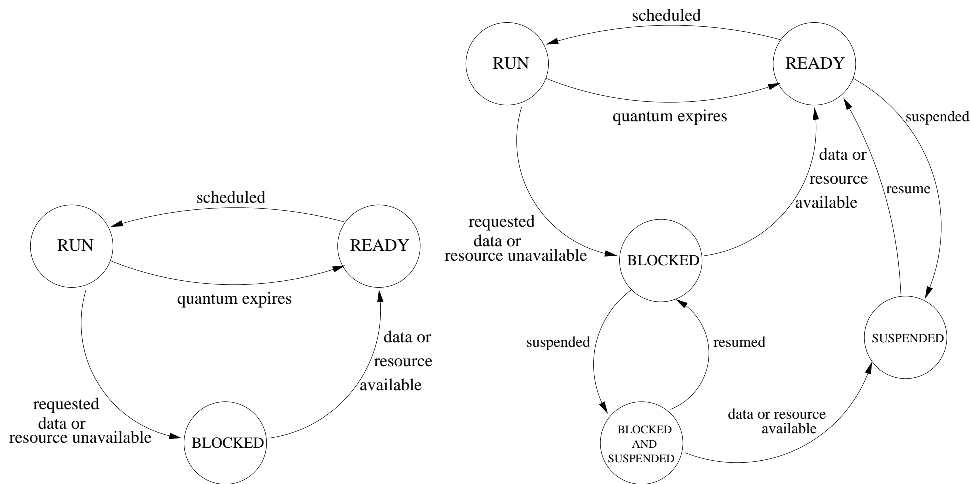
   (a) (5) Your first idea: arrange the images on the disk so that the ones that are queried tend to be located together on the disk. This requires you to change the file management portion of the OS in order to control the data layout, but you're an intelligent OS student (who has been all the way through the course, including the part on file management) and this is an easy task. After the modifications, the program runs faster! Why?

This has the effect of reducing the seek time to move the head to the desired data. By reducing the head movement, the I/O accesses are faster and the program, which is I/O bound, runs faster.

(b) (5) Even though you achieved some speedup, the professor isn't happy because you modified the OS. This means that his program now runs fast only on specific machines, and he wants it to run on any machine. So, back to the drawing board. Your second idea: add multiple threads. Although all the machines you are working on are single-processor, you amazingly still achieve a speedup. Why?

A minor speedup is obtained because you now have a I/O and CPU operations occurring in parallel. A larger speedup is gained because the disk manager now has a larger set of disk locations to choose from for retrieving data, and it can optimize the order in which they are served to minimize head movement and maximize the disk bandwidth, again speeding up the program.

3. **Processes lifetime.** (10) The figure below shows a simple state transition diagram for sequential processes. Suppose that the operating system wishes to implement the two system calls SUSPEND and RESUME. The SUSPEND call can be used by one process to suspend the execution of *another* process. (A process cannot SUSPEND itself.) The suspended process remains suspended until it is RESUMEd by the process that originally suspended it. While a process is suspended it cannot run. Redraw the figure in your bluebook, adding transitions and/or states to account for the SUSPEND and RESUME system calls. Label any new transitions to indicate what caused the transition to occur, much as the existing transitions are already labeled. Label any new states you add *and* briefly describe their purpose.



The `suspended` *state holds any process that is not waiting on any resources, it is only waiting to be resumed. The* `Blocked and Suspended` *state holds any process that was both suspended by another process AND it is waiting on some resource. It is possible for a resource to be granted to a suspended process, so the link between* `Blocked and Suspended` *and* `Suspended` *is necessary.*

4. **Scheduling.** Suppose a processor uses a prioritized round robin scheduling policy. New processes are assigned an initial quantum of length q. Whenever a process uses its entire quantum without blocking, its new quantum is set to twice its current quantum. If a process blocks before its quantum expires, its new quantum is reset to q. For the purposes of this question, assume that every process requires a finite total amount of CPU time.

   (a) (5) Suppose the scheduler gives higher priority to processes that have larger quanta. Is starvation possible in this system? Why or why not?

   No, starvation is not possible. Because we assume that a process will terminate, the worst that can happen is that a CPU bound process will continue to execute until it completes. When it finishes, one of the lower priority processes will execute. Because the I/O bound processes will sit on the low priority queue, they will eventually make it to the head of the queue and will not starve.

   (b) (5) Suppose instead that the scheduler gives higher priority to processes that have smaller quanta. Is starvation possible in this system? Why or why not?

   Yes, starvation is possible. Suppose a CPU bound process runs on the processor, uses its entire quantum, and has its quantum doubled. Suppose a steady stream of I/O bound processes enter the system. Since they will always have a lower quantum and will be selected for execution before the process with the doubled quantum, they will starve the original process.

5. **Nice.** The other day I was using my Linux workstation in the CS department, and suddenly, everything I was doing started to slow down. Sometimes it took several seconds to change from an emacs window to a web browser, or even to a terminal window. When I finally ran `top`, I saw that Myrosia (one of the graduate students in AI) was running an application on my machine, and it was taking nearly 99% of the CPU. I could see that she had increased the `nice` value of her process significantly, but my work was still being significantly slowed.

   Another day, I was working, and for no particular reason, ran `top` and found that Grigoris (a graduate student in systems) was using my machine. His process was also taking nearly 99% of the CPU and he had increased the `nice` value of his process, but my work was not being affected.

   (a) (6) What do you think was the difference between the two processes? In other words, how could Myrosia's process be affecting me while Grigoris' did not? Justify your answer.

   Nice only affects CPU usage, not memory usage. It turns out the Myrosia's process is a memory hog, and her processes were taking all of my memory. Grigoris' process was using only a small amount of memory. While both were using almost all of the CPU, my tasks were so small that they easily fit into the remaining 1% of the CPU, but the lack of memory control slowed me down.

   (b) (4) How can the situation be fixed? In other words, what mechanism can be used to still allow Myrosia's process to run without affecting me "as much"? You need only describe the mechanism.

   A mechanism needs to be added to limit the maximum amount of memory for a process. Nice does not do this.

6. **Synchronization.** The following Java code samples describe two Lock classes with two methods each: `acquire()` and `release()`. You can assume that the application calls `lock.acquire()` before entering a critical section and `lock.release()` after exiting the critical section. For the implementations that require a tid (i.e., thread id), you can assume that the tid of each thread is either 0 or 1.

```
class LockA {
  private int turn = 0;

  public void acquire(int tid) {
      while (turn == (1 - tid));
  }
  public void release(int tid) {
      turn = (1 - tid);
  }
}
```

```
class LockB {

    public void acquire() {
       disableInterrupts();
    }
    public void release() {
       enableInterrupts();
    }
}
```

(6) For each lock, answer the following three questions. Be sure that your answers clearly indicate whether you are referring to LockA or LockB.

(a) Does the code guarantee mutual exclusion? (Simply answer yes or no)

(b) Does the code guarantee progress? (Simply answer yes or no)

(c) List all other limitations that exist for each implementation. Issues you might consider include (but are not limited to) the following: generality, efficiency, and fairness. (Note: You can skip this part of the question when the implementation fails to provide mutual exclusion or progress.)

LockA (a) Yes, guarantees mutual exclusion; only the thread with tid matching turn is able to acquire the lock. (b) No, does not guarantee progress; if thread 0 never tries to acquire the lock (it is executing other code), thread 1 will not be able to acquire the lock. (c) Limitations (Not required): Only works with two processes, uses busy waiting.

LockB (a) Guarantees mutual exclusion on a uniprocessor, but not on a multiprocessor. On a uniprocessor, once the timer interrupt is disabled, the scheduler won't be able to switch to another process (assuming the scheduled process doesn't voluntarily relinquish the CPU). However, on a multiprocessor, it is possible for the other CPU to be running a process that also acquires the lock. (b) Yes, guarantees progress; once a process is scheduled, it is able to acquire the lock without incident. (c) Limitations: Only works on uniprocessors; allows user processes to disable interrupts for an arbitrary long period; cannot service other important interrupts during critical section (e.g., I/O); cannot schedule any other processes when lock is held, even those not contending for the lock.

(4) Also answer the following general question: Locks are often implemented by maintaining a list of processes (or threads) that are waiting to acquire the lock. What are all of the advantages of this approach (compared to a correct implementation of a lock that does not use a list)?

fairness and efficiency. Fairness comes from the fact that the queue can provide the lock to the processes in the order they request it. Efficiency comes from the elimination of the spin lock, and reliance on a notification mechanism to wake-up a process.

7. **Scheduling and Synchronization.** Suppose two processes compete for access to a critical section using simple spin locks. Prior to entering the critical section, the process executes the following: `while (TAS(t));` and after exiting the critical section it executes `t=1`.

   (a) (5) Suppose a priority scheduler where high priority processes always execute before any lower priority processes. Can the described scheme lead to deadlock? If no, why not. If yes, describe a case.

   Yes, deadlock is possible. If a low priority process acquires the lock. Then a high priority process starts, gets the processor and requests the lock. The high priority process will keep the processor, spinning on the lock that the low priority process is holding. Because the low priority process cannot get the processor, it cannot release the lock, and we have a deadlock scenario.

   (b) (5) Suppose a round robin scheduler. Can the described scheme lead to deadlock? If no, why not. If yes, describe a case.

   No, deadlock is not possible. A round robin, preemptive scheduler will alternate among processes, so every process will have a chance to execute, and eventually the holding process will release the lock.

8. **Deadlock prevention.**

   (a) (6) What are the necessary conditions for deadlock?

   (1) mutual exclusion - at least one resource is non-sharable (2) hold and wait - a process is holding some resource while waiting for another (3) no preemption - cannot take resource away (4) circular wait - P0 waits for P1 waits .... Pn waits P0

   (b) (3) Fix the following code to avoid the possible deadlock:

```
acquire(L1)        acquire(L2)
acquire(L2)        acquire(L1)
release(L2)        release(L1)
release(L1)        release(L2)
```

   You could have changed this many ways. One possibility is to remove circular wait and make both processes acquire the resources in the same order. Another possibility is to remove hold-and-wait and have the processes release the first resource before they acquire another. Although this changes the semantics of the program, I accepted it as a solution.

   (c) (1) What condition from (a) did you remove by making your change.