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

Final Exam

Amy Murphy

7 May 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 **5 of the 7** questions to answer. For extra credit, you may answer **one** other question, 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 5 in your bluebook are for *normal* credit and the next one (if any) is *extra*. If you answer all 7 questions, only six of them will count toward your grade. Grades will be assigned 0-50 (with 10 possible extra credit).

CSC456: Students enrolled in 456 must choose **6 of the 7** questions to answer. For extra credit, you may answer the 7th question, but you must clearly identify which question is to be counted as *extra*. If you do not indicate any, I will assume the 7th answer (if any) is *extra*. Grades will be assigned 0-60 (with 10 possible extra credit).

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

   (a) (3) Name one advantage to a large scheduling quantum. Name one disadvantage.

   A large quantum means fewer context switches (less overhead) for compute bound processes. On the other hand, interactive processes may be less responsive when mixed on the same processor with compute bound processes.

   (b) (3) Name one advantage to a large memory page size (assume the frame size and the page size match). Name one disadvantage.

   A large page size means less overhead to track to the mapping from virtual page to physical frame. On the other hand, there will be more internal fragmentation.

   (c) (4) Name one advantage to a large size disk block. Name one disadvantage. How does the Unix Fast File system (FFS) overcome this disadvantage?

   A large block size means it takes fewer independent accesses to to read/write an entire file. Alternately, it may cause internal fragmentation. FFS allows a block to be broken up into smaller pieces, and each one allocated (possibly) to a separate file.

2. **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.

3. **Multiprocessor Scheduling.** When porting an operating system from a uniprocessor system to a multiprocessor system, several of the core components must be simultaneously protected and ensured to scale. One such component is the scheduler.

(a) (3) Define processor affinity, and why it is important in multiprocessor systems.

Processor affinity means that once a process has executed partially on a processor, it is usually worthwhile to keep processing it on the same processor. In other words, it would be more costly to switch it to another processor to continue executing.

(b) (4) If we assume a single run queue for the entire system, then scalability issues may arise because of (1) an increase in the number of processors accessing the queue or (2) an increase in the lock holding time when removing a process from the run queue. Why would creating multiple run queues, one per processor help the scalability issue?

This eliminates the contention over a single shared queue, making it faster to get the next process to execute.

(c) (3) Despite processor affinity and having a separate run queue for each processor, why is it still necessary to provide a locking mechanism around each queue?

The processors will still need to interact in order to balance the processing load.

4. **Synchronization.** The following is a proposed barrier synchronization procedure using only atomic reads and writes. Assume that the only shared state among the processes is the array `mem[MAXPROCS]` which is initialized to all zeros.

```
void barrier(int myID, int *sense, int nProcs) {
   int count, idx, cur;
   mem[myID] = *sense;

   do {
      count = 0;
      for (idx = 0; idx < nProcs; idx++)  {
         cur = mem[idx];
         count += (cur == *sense);
      }
   } while (count < nProcs);

   /* rotate sense for next time */
   *sense = *sense + 1;
   if (*sense == 3)
      *sense = 0;
   return;
}
```

(a) (5) This procedure is broken. In other words, it will not guarantee barrier synchronization in all cases. Describe a case where it will fail.

There is a race condition where one thread will get through the barrier, detecting that all processes are at the barrier. It will then proceed through the next concurrent section and enter the next barrier, changing its sense before some other thread exist the first barrier. Then, neither thread will be able to continue.

(b) (5) How can you fix this procedure, using only atomic reads and writes. In other words, you cannot use any advanced synchronization mechanisms.

The easiest mechanism is to check if the sensed value is equal to my sense value or equal to my sense value plus 1. Because we cycle through three different sense values, there will never be any race conditions.

5. **Memory Management.** There are several page replacement algorithms available for managing the memory inside the operating system. The *clock* pageout algorithm maintains a pool of candidate victim pages using reference and dirty bits.

   Suppose we have a system where pages are paged either to a local disk or to a remotely mounted disk. Remote data is more expensive to save and retrieve.

   (a) (6) Describe a modified clock algorithm that tends to prefer local pages when selecting a victim for pageout.

   Each page is modified to indicate whether it is local or remote. The triple of values used by the clock to determine whether to page contains bits for (ref, dirty, remote). As the clock hand rotates through the pages to select victims, the decreasing preference order is as follows: (0,0,0), (0,0,1), (1,0,0), (1,0,1), (0,1,0), (0,1,1), (1,1,0), (1,1,1).

   (b) (4) Describe an extension of your algorithm that can handle more than two levels of paging store.

   Instead of keeping a single bit for the remote bit, keep multiple bits which represent the weight.

6. **Device Management.** A observant OS student was looking at the utilization of the departmental machines, and realized that the mix of applications was interesting. Usually, there was one I/O intensive data mining application running in the background, and several user applications with a moderate to large amount of disk I/O. The data mining application was running at the same nice level as the other processes, and because it was constantly performing disk read operations, the performance of the other processes was significantly affected.

   (a) (3) In a traditional disk system, such as the one running on the departmental machines, what is the delay cost associated with a single disk operation?

   disk access = seek time + rotational delay + operation time

   (b) (7) Thinking about this cost, and in particular the wasted time spent on each access, the student came up with a way to get some data to the data mining application basically *for free*, without significantly affecting the I/O behavior that the interactive processes would see if the data mining application was not running. While the performance of the data mining application degraded, the loss was acceptable. Another benefit was an increase in the overall bandwidth utilization of the disk. What was the student's idea? Assume that the data mining application could be easily modified to make simultaneous, block-size read requests to the disk and accept any block-size data returned to it.

   The idea is to prioritize disk requests so that "normal" users receive priority so long as they exist. Whenever possible, however, we take advantage of the (unavoidable) rotational delay associated with the next user access to read any data-mining block that may rotate past the read heads. This is straightforward if the data-mining block is in the same cylinder as the

last or next user access. It may even be possible when the data-mining block is in a different cylinder, so long as the relevant seek times are smaller than the relevant rotational times.

7. **Distributed deadlock detection.** One mechanism for detecting deadlock on a centralized system is for the operating system to maintain a resource allocation graph where the nodes represent the processes, and an edge from one node to another indicates that one node is waiting on a resource held by another process. In other words, if R_1 is held by process P_1 and P_2 requests R_1, an edge is added from P_2 to P_1.

   In a distributed system, we have resources and processes spread across multiple hosts, and processes that require resources on multiple hosts. While it would be possible to manage all resources in a centralized way (keeping a single resource allocation graph), consider a more distributed solution such as the following.

   Each host keeps its own resource allocation graph of the local resources. When a process (P_1) requests a remote resource and is made to wait, a symbolic node (P_{ex}) is added to the local graph, and an edge added from P_1 to P_{ex}. Similarly, when a resource held locally by P_2 is requested and a remote process must wait for that resource, an edge is added from P_{ex} to P_2. There is only one P_{ex} node per local host, representing *all* other hosts.

   (a) (5) Assume the system is globally deadlocked. Is it possible for a process which is part of that deadlock to detect the deadlock looking only at the local resource allocation graph? If so, how? If not, why not?

   Yes, deadlock can be detected. We are guaranteed that if deadlock exists in the system, there will be a cycle on at least one host.

   (b) (5) Is it possible that such a local detection algorithm will erroneously detect deadlock? If so, give an example. If not, sketch a proof for why not.

   Yes, it is possible. Suppose P_1 on host 1 and P_2 and P_3 on host 2. If P_1 is waiting on P_2, and P_3 is waiting on P_1, then host 1 will have a cycle from P_1 to P_{ex} to P_1, but as long as P_2 makes progress and releases the resource, the system is not deadlocked.