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

Midterm Exam Solutions

Amy Murphy

28 February 2001

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 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 5 in your bluebook are for *normal* credit and the remaining 2 (if any) are *extra*. Grades will be assigned 0-50.

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 is *extra*. Grades will be assigned 0-60.

1. **Short answer**: your responses should be at most 3-4 lines of text:

   (a) (4) Name any three main components of most major operating systems (such as Unix or Windows-XX) and briefly describe their role.

   Memory management, process management, file management, I/O system management, networking, etc.

   (b) (3) Explain how the mixture of I/O bound processes with CPU bound processes maximizes system utilization. Why is this more important in batch systems than it is on most computers sitting around in our department?

   Want to keep all parts of the system busy, so if you can have some processes using the cpu while others wait on I/O, then the system will be more utilized. In general, our systems are underutilized (there are a lot of wasted cycles both on I/O devices as well as cpu), and user interaction is more important. Batch systems were expensive systems that were busy all the time.

   (c) (3) Define a counting semaphore, its initialization, and the two operations used to operate on it.

   A counting semaphore is a structure which can be used to give a specified number of processes access to a critical region. The number of concurrent accesses allowed is the initial value of the semaphore. wait() decrements this value and either causes the process to wait until the resource is available or allocates the process the resource. signal() increments the counter, releasing a waiting process (if any).

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

Process ID	CPU requirements
P1	7
P2	1
P3	2
P4	5
P5	3

(a) (5) Show the scheduling order for these processes under first-come-first-served, shortest-job first, and round-robin scheduling (quantum = 1).

FCFS: 12345
SJF: 23541
RR: 123451345145141411

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

	P1	P2	P3	P4	P5
FCFS wait	0	7	8	10	15
FCFS turn	7	8	10	15	18
SJF wait	11	0	1	6	3
SJF turn	18	1	3	11	6
RR wait	11	1	5	11	9
RR turn	18	2	7	16	12

The thing to remember here is that the wait time is the total time spent in the wait queue.

(c) (1) Briefly explain why these two values are meaningful criteria for evaluating scheduling algorithms.

Together they give a criteria to measure the effectiveness of a scheduling algorithm, specifically with respect to properties perceived by the processes themselves.

3. **Process Creation.** Modern unix operating systems use copy-on-write in order to implement the `fork()` operation for creating a new heavy-weight process.

(a) (3) Define the difference between a heavy-weight process and a thread (or lightweight process).

A heavy-weight process has a separate memory space for all aspects of the process, including data, instructions, and dynamic memory. Lightweight processes share data and instructions.

(b) (4) In class, we discussed copy-on-write for memory pages shared among multiple processes. We cannot apply this same concept blindly to process creation, but instead are forced to copy some parts immediately while other parts can be delayed. Knowing the components of general processes, which parts must be copied immediately, and which parts can be delayed and *copied-on-write*?

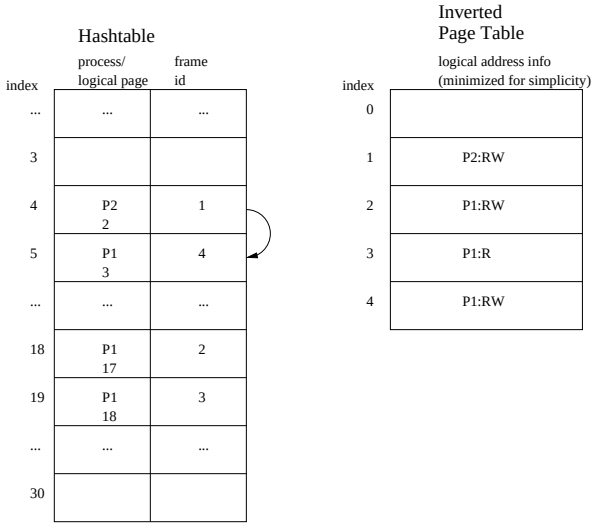
This question was referring to processes, not caching. When we create a new process, we must copy the stack space and registers, but need not copy the entire address space immediately. This is because often a call to fork creates a new address space into which a new process is immediately loaded, making the initial copy a waste of time and space because it is immediately overwritten. An example of this is when a command shell starts up a new process, first doing a fork to create a new copy of itself, but then replacing that copy with the new process which was executed on the command line.

(c) (3) Why is copy-on-write potentially better than copying the entire process immediately upon creation?

It can save time and space during the process creation (as mentioned above), avoiding the duplicate effort of making a copy, then immediately overwriting it.

4. **Inverted Page Tables.** Assume we have a simple, demand paging environment, with no segmentation. Processes $P1$ and $P2$ both have logical memory addresses in the range $0 \ldots 99$, inclusive. The page size is 5. The hash table portion of the structure uses a hash function which simply calculates the index by adding the numerical portion of the process id and the logical page id. (Note, this is a bad hash function because it requires a large hash table, but for our exercise, this

is good enough.) The hash table and the inverted page table are below. Arrows in the hash table indicate chaining of entries when a hash conflict has arisen. Empty entries in the IPT indicate the physical frame is not allocated.



(a) (6) Calculate the physical addresses for the following logical addresses where the number after the colon is the logical address (not the logical page). Assume the first page of a process has id 0: P1:17. P1:92. P1:111. Show you work for these calculations (in other words, what calculations/table lookups were necessary to get the actual address in memory from the logical address).

P1:17: Logical page $17/5 = 3$ offset $= 2$. hash table index $= 3 + 1$, follow chain to entry 5. From lookup, logical frame $= 4$. Address $=$ (frame)$*$(frame size) $+$ offset $= 4 * 5 + 2 = 22$

P1:92: Logical page $= 18$. offset $= 2$. Hash table index $= 19$. From lookup logical frame $= 3$. Address $= 17$

P1:111: exception, memory reference out of range.

(b) (4) When process P2 attempts to read logical address 97, what happens? Describe the changes in the data structures, and what the process perceives of these changes. Assume a mechanism exists to find the free frame in memory.

This causes a page fault. Frame 0 is allocated, and the entry in the IPT is updated with the logical address information for P2 frame 19. Next the hashtable entry at index 21 $(19 + 2)$ is mapped from P2:19 to frame 0. After this processing is complete, the P2 resumes execution as before.

5. **Synchronization.** In parallel programs (a single process with multiple threads), a common way to design the processing is in stages. All threads work independently during each stage, but must synchronize at the end of each stage. When all threads reach this synchronization point (called a barrier), they are notified and begin execution on the next phase of the computation.

Two potential complications: In many cases, there is no master thread watching over the children threads, waiting for each of them to get to the barrier, and then telling them to re-start. In other words, the children must monitor themselves! Second, it is often not known in advance how many children threads there will be during the lifetime of the parallel program. In other words, a child

can spawn another child (realizing that this new child will start in the same program stage as the child which created it)! In all cases, all children must synchronize at the barrier before the processing is allowed to continue to the next phase.

(a) (8) Your task is to provide the pseudo code for a monitor class called `Barrier` which enables this style of barrier synchronization. Things to take into consideration are the creation of a new child thread (one more process that needs to synchronize), processing when a thread reaches the barrier, and the releasing of waiting threads when the *last* thread reach the barrier. Remember that some of the standard monitor methods available to you are `wait()` and `signal()` and `signalAll()`. (Hint: if you know about Java synchronized objects, this concept is very similar.)

This was much easier than most people made it out to be. The main problem that people had was knowing the difference between monitors and semaphores. When you use a monitor, you need not use a semaphore (or any other atomicity-guaranteeing operation) to get atomicity of operations. That is the job of the monitor.

```
Class Barrier ()
    numThreads = 0;
    numThreadsAtBarrier = 0;
    newThreadCreated() {
        numThreads ++
    }
    barrierReached() {
        numThreadsAtBarrier++;
        if (numThreadsAtBarrier == numThreads)
            numThreadsAtBarrier = 0
            signalAll()
        else
            wait()
        }
```

You can also privide a method to be called just before a thread terminates. The synchronization is gurauranteed by the monitor.

(b) (2) Did you put the call(s) to `wait()` at the end of your method(s)? If so, why? If not, what possible complication might arise after your call to `signalAll()`?

Yes, it is there. As soon as the signal is received, all of the processes that were at the wait immediately fall out of the monitor, so there is no confusion about which thread actually gets to proceed through the monitor after the signal.

6. **Deadlocks.** Although deadlock is a real problem, it is often ignored during the design of the operating system due to the inefficiencies which deadlock prevention and/or deadlock detection introduce. One approach to reduce the overhead is to group resources into multiple classes, use a total ordering among these classes, and use a technique tailored to the class for resource allocation within that class. For example, four reasonable classes may be (1) swap space, (2) process resources such as files, tape drives, etc. (3) main memory, (4) internal resources such as communication channels among processes.

(a) (4) Is the sequence C1:R2, C1:R1, C3:R5, C1:R4, C4:R1 allowed (assuming that in Cx, the 'x' refers to the class number and Rx is a resource in that class as described above, and that once the resource is held, it is held for the duration of the process execution)? Is the sequence C1:R2, C1:R1, C2:R3, C4:R7 allowed? What general statement can be made about

the required ordering of resource acquisitions within a process? Is the above ordering of resource classes reasonable to expect of processes? Why or why not?

The first sequence is not allowed because there is an access to class C1 after an access to class C3, which violates the total ordering resource allocation assumption. The second sequence is allowed. The ordering of classes is probably as reasonable an ordering as you can get. Why? Because a process will generally proceed by first requesting swap space (the most that it will need), then will start getting access to resources that it will be accessing, then start running, using main memory, then begin to reach beyond the normal bounds of the process to other system resources.

(b) (3) Thinking about main memory as a resource is interesting, because much of the time, main memory can be preempted from the process it is allocated to. In most systems, what happens when a process' memory is preempted? Why is preemption not typically an option for other resource types (such as a file or tape drive) for handling deadlock recovery?

Preempted memory is typically swapped out to disk, but is still available and can be brought back in at a later time (in a demand paging environment) without disrupting the execution of the process. Typically preemption is not possible for a resource because preemption requires that a process know how to be stopped, restarted, and in some cases rolled back. It is difficult to roll back an operation on a device such as a CD burner.

(c) (3) If we want to use the banker's algorithm for handing the swap space, what information must be provided by every process before it can be allocated swap space? What can the operating system do with a running process that requests swap space when the banker's algorithm rejects the allocation as a potential deadlock situation?

We need to know the maximum amount of swap space that the process might use during its execution. If a new process arrives and we cannot satisfy it's request, the OS can terminate the process, or in a more gentle system, hold the process off the ready queue until the resources are available.

7. **Memory Management.**

(a) (5) Define spatial locality and temporal locality. If a program has a data space much larger than main memory, how will a lack of spatial locality affect its performance? In other words, will it slow down or speed up, and why. For this question, ignore caching, and consider only main memory management.

spatial locality: The principle that if a particular location in memory is accesses, the data physically near to it in memory will also be accessed.

temporal locality: The principle that if a particulare location in memory is accessed, that same piece of memory will be accessed again soon (in time).

In general the system would slow down because the amount of paging would increase greatly and more time would be spent pagin than doing real work.

(b) (5) Define both internal and external fragmentation. Draw a pictorial representation of a pure paging system (not paged segmentation, or segmented paging, but just plain paging) which demonstrates internal fragmentation. Draw a pictorial representation of a pure segmentation system which demonstrates external fragmentation.

internal fragmentation: space inside allocated memory which is wasted. (typically occurs in paging systems)

external fragmentation: space outside of allocated memory which is too small to be used for another process (typically occurs in segmented systems).

In general everyone's pictures were fine. See the book if you need a reference.