

# CSCI 1670 Midterm Exam Solutions

Fall 2007

## Do all of questions 1 through 3.

1. Assume we have a uniprocessor system. Each thread has both a user-mode stack and a kernel-mode stack. Interrupt handlers execute on the kernel-mode stack of the current thread. In the following, please mention in whose context (thread or interrupt handler) actions are taken and which stack is being used. If you use a term such as APC, explain what it means and how it works. A thread may call *SchedYield* (in the kernel) to yield the processor to another thread.

- a. Our system is to be preemptible in user mode, but not preemptible within the kernel. The clock interrupt handler determines if the current thread's time slice is over and, if so, requests that it be preempted. Describe the mechanism used to preempt the thread. Note that keeping interrupts masked the entire time a thread is in the kernel is not feasible.

A global variable in the kernel, *preempt*, is used to request the kernel thread to give up the processor. When the clock interrupt handler, running in the interrupt context on the kernel stack of the current thread, determines that the thread's time slice is over, it sets *preempt* to 1. Whenever the system is about to return to user mode it checks *preempt* and, if it is 1, it resets *preempt* to 0 and calls *SchedYield* to have the current thread yield the processor to another thread. This return to user mode can happen in two ways. If the interrupt occurred while the thread was executing in user mode, then the system returns to user mode directly from the interrupt context. If the thread was interrupted while executing in the kernel, then the return to user mode will not be until the thread returns from the trap that took it into the kernel in the first place. At this point the system will be executing in the context of the thread. In both cases, the system is executing on the thread's kernel-mode stack.

- b. Our system is now to be preemptible both in user mode and in the kernel. Describe the mechanism used to preempt a thread.

Again the clock interrupt handler sets *preempt* to one when the current thread's time slice is over. However *preempt* is checked whenever the system is about to return from the interrupt context to the thread context — if it is 1, it is cleared and it calls *SchedYield*. In this case the system will necessarily be in the interrupt context and on the kernel-mode stack of the thread.

2. In the Fortran programming language, arrays are stored in column-major order, meaning that adjacent entries in each column are stored in adjacent memory locations. For example (using C notation for matrices), *Matrix[i][j]* is stored just before *Matrix[i+1][j]*. However, most scientific programs written in Fortran access arrays using loops similar to the following (again, using C notation rather than Fortran):

```
int Matrix[M][N];
for (i=0; i<M; i++)
    for (j=0; j<N; j++)
        Matrix[i][j] = somefunction(i, j);
```

For example, if  $M$  is 6 and  $N$  is 4, the elements of the matrix are stored in the order shown below:

0	6	12	18
1	7	13	19
2	8	14	20
3	9	15	21
4	10	16	22
5	11	17	23

This caused no problems until such programs were run on machines supporting virtual memory, where performance (particularly for large values of  $M$  and  $N$ ) turned out to be much worse than on machines that didn't support virtual memory. Explain why performance was worse. (Hint: consider the page-replacement policy.)

Suppose that each column of the matrix occupies exactly one page. Then each iteration of the inner loop touches a new page, forcing it to be brought into primary memory. If there are more columns than available page frames, then each iteration of the inner loop will probably result in a page fault, since by the time a column is accessed again, the page frame containing it will have been replaced. Thus, there will be a page fault for each iteration of the inner loop, for a total of  $MN$  page faults. However, if the matrix were stored in row-major order, then the iterations of the inner loop would walk through each page of entries of the matrix sequentially, and each page would be faulted in only once. Thus there would be  $MN/P$  page faults, where  $P$  is the number of matrix entries in a page.

3. *The layout of the file system on disk is an important factor in file-system performance. Since files are often accessed sequentially, the performance of sequential access is important.*

- a. *In the early implementations of the FFS file system, the kernel was constrained to accessing files one block at a time — combining two one-block requests into one two-block request was not possible. (To keep your answer relatively simple, assume that files are allocated in whole blocks and that fragments are not used.) You would like to eliminate this constraint, allowing multiple one-block requests to be combined into a single multiple-block request. What constraints would be necessary on the layout of the blocks for this to be possible? How could the file system determine that these constraints are satisfied? (Hint: think about metadata.) What modifications would be necessary to the kernel's buffer-management code to support such multi-block transfers?*

It would be necessary for the blocks to be contiguous on disk. To determine this, the file system code would have to check the block locations, stored in the inode's disk map and the indirect, doubly indirect, and triply indirect blocks to make sure they are contiguous. The buffer-management code would need to be modified so as to allocate space for variable-size buffers.

- b. *Would block-interleaving still be necessary if this approach were used? Explain. (Assume that the disk controllers employed do not buffer tracks internally.)*

No. With this scheme it would be feasible to lay out blocks contiguously around a track and then to read a track's worth of blocks in a single disk rotation.

- c. *Modern disk controllers read the contents of entire tracks into internal buffers, from which the data can be transferred directly to primary memory in the computer. (Assume*

*the transfer from the buffer to computer memory takes negligible time.) Would this make unnecessary the scheme described in part a? (We are concerned only with reading data, not with writing data.)*

Yes. Regardless of the order in which the kernel issues read requests and the sizes of these requests, all of a track will be transferred to the internal buffer in a single rotation. Then the blocks contained in the buffer can be transferred to memory quickly in response to kernel read requests.

**If you do all of the following correctly, you'll get an A regardless of how well you do on the first three problems. If you miss any of the following, your grade will be based solely on how well you do on the first three problems.**

4. *What is a Rhinopias (other than a kind of disk)?*  
It's a rather spectacular (and rare) genus of scorpionfish.
5. *Why is the Sun Lab called the "Sun Lab"?*  
Because it was originally filled with computers made by Sun Microsystems.
6. *In the early days of the Brown CS department, most instructional computing was done on computers made by Apollo and most research computing was done on computers made by Digital. Neither company currently exists. What one company now owns what remains of both of them?*  
Hewlett Packard
7. *The CS Department purchased an academic license for Unix in 1979. How much did it cost? How much would a commercial license have cost?*  
\$350 and \$20,000.
8. *What did the original Unix license prohibit the teaching of?*  
The internals of Unix to students who were not employees of the university and thus not covered by the terms of the license.
9. *Two CS faculty members hold endowed chairs named for computer-industry pioneers. Who are the faculty members and who are the pioneers?*  
Franco Preparata is the An Wang Professor of Computer Science. Andries van Dam is the Thomas J. Watson Jr. Professor of Technology and Education. Wang was the inventor of core memory and the founder of Wang Laboratories, now defunct. Thomas J. Watson Jr. was the son of the founder of IBM and president of IBM from 1952 to 1971.
10. *Your professor has been teaching the operating systems course at Brown for a long time. Which current Brown faculty member taught it before he did?*  
John Savage.