EECS 343: Final Exam - Sample

From threads to file systems.

Fall 2014

Name (NUID):

Some words of advice:

- Read all questions first.
- Start from the easiest one and leave the harder ones for later.
- Approximate results are almost always a valid answer; for sure I do not need 5-decimal precision answers!
- Write clearly; **if I can't read it, I can't grade it.**

Question	Total	Credited
1	?	
2	?	
3	?	
4	?	
5	?	
6	?	
7	?	
8	?	
9	Extra 15	
10	Extra 15	
Total	130	

Good luck!

Problems

1. **(? points)** Can the priority inversion problem discussed in class happen with user-level threads? Why or why not?

   Answer: *The priority inversion problem occurs when a low-priority process is in its critical region and suddenly a high-priority process becomes ready and is scheduled. If it uses busy waiting, it will run forever. With user-level threads, it cannot happen that a low-priority thread is suddenly preempted to allow a high-priority thread run. There is no preemption. With kernel-level threads this problem can arise.*

2. **(? points)** In a system using user-level threads, is there one stack per thread or one stack per process? What about when kernel-level threads are used? Why or why not?

   Answer: *Each thread calls procedures on its own, so it must have its own stack for the local variables, return addresses and so on. This is equally true for user-level as for kernel-level threads.*

3. **(? points)** Give a sketch of how an operating system that can disable interrupts could implement semaphores.

   Answer: *To do a semaphore operation, the OS will first disable interrupts. Then it can read the value of the semaphore. If it is doing a* $wait$ *and the semaphore is equal to zero, it puts the calling process on a list of blocked processes associated with the semaphore. If doing an* $post$*, it must first check to see if there are processes blocked in the semaphore. If there are, one of them is removed from the list and made runnable. When all these operations have been completed, interrupts can be enabled again.*

4. **(? points)** Synchronization within monitors uses condition variables and two special operations `wait` and `signal`. A more general form of synchronization would be to have a single primitive, `waituntil`, that had an arbitrary Boolean predicate as parameter. Thus, one could say, for instance

   ```
   waituntil ((x < 0) || ((y + z) < n))
   ```

   The signal primitive would no longer be needed, of course. This scheme is clearly more flexible than then one discussed, but it is not used. Why not? *Hint:* Think about the implementation.

   Answer: *Basically it is very expensive to implement. Each time any variable that appears in a predicate on which some process is waiting changes, the run-time system must re-evaluate the predicate to see if the process can be unblocked. With the model discussed in class, processes can only be awakened on a* `signal` *primitive.*

5. *(? points)* In which of the five I/O layers (top to bottom: user-level I/O software, device-independent OS software, device driver, interrupt handler, hardware) is each of the following done:

   (a) Computing the track, sector, and head for a disk read.

   (b) Writing commands to registers.

   (c) Checking to see if the user is permitted to use the device.

   (d) Converting binary integers to ASCII for printing.

Answer: *a. device driver, b. device driver, c. device-independent software, d. user-level software*

6. *(? points)* A key issue when designing a file system is deciding how to keep track of which disk block goes with which file. Of the approaches discussed in class, which one would you choose to maximize efficiency in terms of speed of access, use of storage space, and ease of updating (adding, deleting, modifying) when the data is:

   (a) Updated very infrequently and accessed frequently in random order.

   (b) Updated frequently and accessed in its entirety relatively frequently.

   (c) Updated frequently and accessed frequently in random order.

   Answer:

   (a) Continual allocation

   (b) Linked list

   (c) Either i-node or FATS

7. *(? points)* Discuss the pros and cons of using large blocks to store files?

   Answer: *Large blocks may result on better performance, but worst space utilization. Having a large block size means that small files will waste a large amount of disks space. On the other hand, a small block size will result on files potentially spanning multiple blocks with the corresponding performance overhead (from the multiple seeks and rotational delays needed) to access them.*

8. *(? points)* In UNIX System V, the length of a block is 1 Kbyte and each block can hold a total of 256 block addresses. Using the i-node scheme, what is the maximum size of a file? (Remember UNIX System V uses three levels of indirection)

   Answer: *The i-node itself stores the addresses of the first 10 data blocks. For larger files, the last three addresses point to a single, a double and a triple indirect block. With each block holding 256 block addresses, the maximum file size is* $(10 + 256) * 1Kbyte + 256^2 * 1Kbyte + 256^3 * 1Kbyte \approx 16Gbytes.$

9. **(***Extra* **15 points)** Paper-related question.

10. **(***Extra* **15 points)** Paper-related question.