CS 111 Final exam, short

Cody S Hubbard

TOTAL POINTS

117 / 130

QUESTION 1

1 sparse files 10 / 10

✓ **- 0 pts** **Correct**

- 2 pts Did not mention block level allocation detail for FAT

- 1 pts Did not mention pointers in Unix Systems

- 2 pts Did not demonstrate understanding of FAT table entries

- 8 pts Did not answer the question

- 3 pts Incorrect understanding of Unix System V

QUESTION 2

2 Links 9 / 10

- 0 pts Correct

✓ **- 1 pts** **Did not go into detail about name storage and datablocks**

- 2 pts Did not talk about link count

- 3 pts Incorrect understanding about metadata comparison

QUESTION 3

3 Distributed capabilities 7 / 10

- 0 pts Correct

✓ **- 3 pts** **Did not go into detail about network attacks**

- 3 pts Did not show understanding of capability

- 2 pts Did not explain lack of challenges in single machine environment

- 5 pts Did not answer the question

QUESTION 4

4 horizontal scalability 10 / 10

✓ **- 0 pts** **Correct**

- 1 pts Missing some details.

- 3 pts Not very accurate.

- 9 pts Wrong.

QUESTION 5

5 Full disk encryption 10 / 10

✓ **- 0 pts** **Correct**

- 1 pts Missing some details.

- 4 pts Not very accurate.

- 7 pts Wrong in explaining attacks.

- 9 pts Wrong.

- 10 pts Empty.

QUESTION 6

6 Addressing in the OS itself 10 / 10

✓ **- 0 pts** **Correct**

- 10 pts Physical Addresses

- 5 pts The OS uses the same CPU as user processes do, and its instructions are passed through the same MMU as the instructions of those user processes.

- 5 pts Provide extra wrong answer

- 5 pts The concept is correct, but it doesn't answer the question directly.

QUESTION 7

7 Disk defragmentation 10 / 10

✓ **- 0 pts** **Correct**

- 1 pts Some details missing/wrong.

- 9 pts Wrong.

- 4 pts Not accurate.

- 10 pts Empty.

QUESTION 8

8 Preemptive scheduling with threads 3 / 10

- 0 pts Correct

✓ **- 7 pts** **Voluntary yielding is not preemptive scheduling.**

- 10 pts Totally wrong

- 8 pts Not very effective for real preemptive

scheduling.

- **8 pts** Putting a thread into an infinite loop will definitely not preempt it.
- **8 pts** The alternate interpretation is right, but then how does the library achieve preemption?
- **10 pts** OS knows nothing about these threads.
- **8 pts** How do you preempt a thread this way?
- **6 pts** Which interrupts/system calls?
- **10 pts** But how can you actual preempt?
- **10 pts** This is not a problem in mutual exclusion. It concerns mechanisms for controlling execution.
- **4 pts** More details on what system calls to use (i.e., timers).
- **10 pts** Use timer interrupts.
- **0 pts** Simpler to just ask the OS for a timer interrupt, but this probably would work, too.
- **5 pts** Close. The process hosting the threads can ask for a timer interrupt and use that opportunity to switch to a different thread.
- **8 pts** Without OS assistance, how could this be done?
- **0 pts** Clumsy, but workable. Scheduling timer interrupts would be easier.
- **8 pts** Vastly cumbersome to recast a local multithreaded program as a distributed program so you can gain control via RPC. Use timer interrupts.
- **10 pts** No answer

QUESTION 9

9 Sloppy counters 10 / 10

✓ **- 0 pts Correct**

- **3 pts** wrong answer for the first question
- **3 pts** wrong answer for the second question
- **4 pts** wrong answer for the third question
- **2 pts** missing the key point “inaccurate” for the third question
- **10 pts** wrong answer

QUESTION 10

10 Semaphores and thread initialization 8 / 10

- **0 pts** Correct
- **2 pts** Semaphore blocks on < zero.
- **3 pts** Parent runs wait before it can run.

✓ **- 2 pts Not considering both orders of operations**

- **3 pts** Counter should be zero
- **4 pts** Post increments, wait decrements.
- **10 pts** Totally wrong
- **3 pts** Must specify the parent's semaphore operation.
- **6 pts** No guarantee that child runs before parent, so this set of operations won't always work.
- **7 pts** Not how semaphores work, as described.
- **3 pts** Child just runs post, parent just runs wait.
- **3 pts** Child must do initialization operations before the post().
- **7 pts** As described, nobody ever gets to run. Child decrements semaphore on wait(), it's less than zero, he blocks, so nobody ever posts and wakes him (or the parent, who also blocks).
- **10 pts** Semaphores, not mutexes.
- **3 pts** Must specify the child's semaphore operation.
- **3 pts** Must not call wait before creating child, since then there will be no one to post.
- **10 pts** No answer

QUESTION 11

11 Hold-and-wait and deadlock 10 / 10

✓ **- 0 pts Correct**

- **2 pts** While this does avoid deadlock, it doesn't guarantee progress, since the two threads can each re-acquire one of the locks, then fail to acquire the other one.
- **4 pts** This is not a universal solution, since you cannot necessarily group all locks needing to be acquired together. Unless you put all locks under one lock, which serializes locking, which could be worse than deadlock. OK if you specify that the global lock can be released after acquiring other locks.
- **4 pts** Very hard, in general, since many reasonable operations require more than one resource to be updated atomically, and thus locked.

- **2 pts** "In advance" isn't enough. Must request them all at once.
- **10 pts** Totally wrong.
- **3 pts** If you seize someone's lock this way, you may run into atomicity problems.
- **3 pts** If you use one universal lock to control obtaining the others, either you must hold that universal lock as long as you hold any locks, or you must specify that all locks you need are obtained at once while holding the universal lock.
- **3 pts** Not allowing parallelism at all is not a great solution, and can't work for multi-process scenarios.
- **5 pts** Spin locks will never resolve the deadlock.
- **2 pts** Only viable if lock holders know they might lose their exclusive access. With leases, they inherently do. Otherwise, maybe not, which can cause problems.
- **5 pts** Not helpful unless someone releases a lock eventually.
- **3 pts** What is the condition that can't be done?
- **2 pts** Not clear what you mean by "preallocating". Correct for some reasonable meanings, not correct for others.
- **4 pts** Unclear explanation of why it causes deadlock.
- **4 pts** Lots of reasons events don't occur. Only a deadlock if the reason is someone else holds a lock preventing it.
- **5 pts** This approach attacks the mutual exclusion condition, not hold and wait. You can have mutual exclusion while still preventing hold and wait.
- **3 pts** Kills parallelism if you must hold global lock while holding any other lock.
- **5 pts** Priorities won't help.
- **5 pts** So how do you avoid it?
- **4 pts** Can happen with locks at the finest granularity.
- **5 pts** Far more drastic than necessary. Without further investigation, how do you know which waiting processes to even kill?
- **2 pts** How do you generalize this?
- **4 pts** Not helpful to only take locks away when a process is done with them.
- **5 pts** Incorrect description of hold-and-wait problem.
- **5 pts** If mutual exclusion is required on a particular piece of code, you can't move it out of lock, so this isn't a real solution.
- **5 pts** Preventing preemption doesn't help.
- **5 pts** Semaphores don't prevent deadlock.
- **5 pts** Waking a thread waiting on a lock doesn't help.
- **5 pts** Scheduling alone won't help. And not running any other process while one is holding some locks would kill performance.
- **4 pts** More details on what you mean by reservation here.
- **4 pts** Just checking doesn''t help.
- **3 pts** A bit more than that is necessary to take away a lock without causing problems.
- **5 pts** Would cause serious concurrency problems.

QUESTION 12

12 Event-based concurrency and async I/O 10 / 10

✓ **- 0 pts Correct**

- **2 pts** missing the point that there is only one thread handling all events.
- **10 pts** wrong
- **2 pts** missing the point that it affects performance and concurrency
- **2 pts** missing the point that synchronous I/O blocks other events

QUESTION 13

13 Optimizing file writes 10 / 10

✓ **- 0 pts Correct**

- **3 pts** Cylinder groups don't help that much on writes.
- **1 pts** Delayed writes also complicates ensuring file system consistency, especially in the face of crashes.
- **2 pts** Write buffering doesn't have anything to do with sequential writing. It's about delaying writes till they're cheap or unnecessary.

- **10 pts** Nothing to do with file system write performance.
- **4 pts** Not really a write optimization. More about reads.
- **3 pts** Require more details on exactly how this optimization interacts with writes.
- **2 pts** Introduces problems with on-disk consistency.
- **10 pts** Not a write optimization
- **1 pts** Write buffering usually handled in block I/O cache.
- **3 pts** Complexities?
- **1 pts** Journaling requires garbage collection.
- **10 pts** No answer.