CS 111 Final exam, short

Nate Nateerawut Sookwongse

TOTAL POINTS

108 / 130

QUESTION 1

1 sparse files 7 / 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 10 / 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 9 / 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 0 / 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 10 / 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.

1. Why is the DOS FAT file system unable to efficiently store sparse files? Why can the Unix System V file system store such files much more efficiently?

DOS FAT file system allocates chunks to files. So if sparse files (much smaller in size than the chunk size) are to be stored, there would be a lot of internal fragmentation, and unutilized storage space.

Unix System V file system on the other hand tackles this issue by having smaller fixed-sized allocations for cases of sparse files. And larger files also have larger fixed-sized allocations sort of like a "hybrid" to account for small and large files.

2. What is the difference between a hard link and a symbolic link in a Unix-style file system? What implications does this have for how metadata about the file referenced by the link is stored?

A hard link associates a new file name that points to the same inode of the file you are linking to.

A symbolic link simply contains a path name to the file you are linking to. The metadata of the file reference would be found in the inode for that file.

A symbolic link can point to directories and files of other file systems (hard-link can't). But if the file is deleted, it could point to nothing.

3. If you use capabilities to provide access control in a distributed environment, what extra challenges do you face that you do not face when using them in a single machine environment?

One challenge is security. Because you are not on a single machine environment, there will have to be communication between machines, which can be compromised. Since a capability is just a stream of bits, someone could potentially capture this data.

Another challenge could be authentication, due to the fact it's dealing with various machines.

In addition, the ability to generate a list of who can access a particular resource does not scale well.

4. What is meant by horizontal scalability in a distributed system? Why is it good?

Horizontal scaling is achieved by adding more resources, such as adding another machine/server to the distributed system.

It is good because it is usually a relatively cheap way to add processing power and increase performance.

With horizontal scaling, you are not limited by hardware, but rather by the management of added machines.

5. What kinds of attacks does full disk encryption protect against? Why is it effective against these attacks?

Full disk encryption protects against maliscious attacks and attempts to extract data from the disk. Even if people are able to acquire the disk data, if it is encrypted, then they won't be able to exploit or make use of the data.

6. When the operating system issues addresses for RAM locations for its own use (such as accessing a process control block or finding a particular buffer in the block cache), is it issuing virtual addresses or physical addresses? Why?

The OS virtualizes the address space to make it easier for and give the illusion that each process has the entire space.

It'd be more efficient if the OS utilizes physical addresses for its own use, because it saves the cost of having to look up the Page Table or TLB to translate a virtual address to physical address.

7. What form of fragmentation does hard disk defragmentation help with? Why does it help with this form and not the other form?

Hard Disk defragmentation helps against the issue of external fragmentation. For instance, as blocks are freed, there may be smaller free blocks scattered, as opposed to a long contiguous sequence, and defragmentation can help by rearranging the blocks. Since blocks are fixed-sized allocations, internal fragmentation is unavoidable, and defragmentation can't help that (i.e. 2 different files can't "share" part of a block)

8. How can a user level thread package achieve preemptive scheduling?

Preemptive scheduling would have to be an additional feature that's implemented since the OS can only view user-level threads as a single process running.

One way could be to cause the user-level threads to yield so that a different thread can be scheduled to run.

11. What problem is solved with sloppy counters? How do they solve this problem? What is the disadvantage of using them?

Sloppy Counters helps solve the issue of performance and scalibility for using a counter with multiple threads. It solves this problem by giving each thread it's own local count variable. So as threads want to access the count resource, there is no contention or waiting necessary. A global count variable is periodically updated, which combines the local counts. A disadvantage is the count (global count) may not be accurate (depending on the update period)

12. Consider the following use of semaphores. A parent thread creates a child thread. The parent should not run until the child thread has performed a set of initialization operations. What should the semaphore's counter be initialized to? Which semaphore operations should the parent and child thread call, and when? Why does this use of a semaphore achieve the desired goal?

Parent-Thread Pseudocode

)init Semaphore count to 0
) create child thread
3) wait (semaphore)
4) continue other operations

Child-thread Pseudocode

1) Run initialization operations
2) post (semaphore)

* wait() decrements the semaphore's count and puts thread to sleep if count < 0

By initializing the sem's count to 0, and placing the wait/post operations as shown above, 2 things can happen:

1) The parent creates child, calls wait(), and the semaphore count is -1, so parent goes to sleep. Once child finishes initialization, and then calls post(), the count is incremented to 0, and the parent can wake up and continue.
) If by the time parent calls wait(), and after decrementing, the count is 0, that means the child already finished initializations, and the parent can continue.

13. Why is hold-and-wait a necessary condition for deadlock? Describe one method that can be used to avoid the hold-and-wait condition to thus avoid deadlock.

Hold-and-wait happens when starting to acquire multiple locks. It could be "holding" onto locks while "waiting" for additional locks. This can lead to a deadlock if A is waiting for a lock that B is holding, but B is holding onto that lock and waiting for another lock as well. Without this condition, deadlock would not occur, because eventually all locks should be released, and there's not infinite-waiting.

One method to avoid this is to release previous locks, as subsequent locks are found to be unavailable. For instance:

```
acquire Lock A;
if can't acquire B
    release A;
```

14. Why is asynchronous I/O useful for systems using event-based concurrency?

In the case of a single-core machine, asynchronous I/O makes event-based concurrency possible. Since there's only a single event-loop, if there's an I/O request, the async I/O can be requested without blocking, and the remaining events can then be processed. Without async I/O, there would be no concurrency, because the I/O request would block and nothing else would get done.

15. Describe an optimization related to making writes to a file system perform better. When does this optimization help? What complexities does this optimization add to the operating system and to expected file system behavior?

One type of optimization is the use of a buffer and cache, so that every "write" call doesn't necessarily result in an I/O operation.

One way this can help is by merging I/O requests and thus reduces the overall number of I/O operations required. Since I/O requests are slow, this can greatly improve performance.

Another case this could help is if a "temporary file" is being created and deleted shortly after. With this optimization, it saves time by not having to issue the I/O request and actually write that temp file to disk.

An added complexity is now the need for a policy to determine when to actually write to the disk. In addition, this may bring more issues in the case of a system crash (there could be more data that wasn't able to be written to disk before the crash)