Final Exam
CS 111, Principles of Operating Systems
Fall 2017

Name: Cody Hubbard

Student ID Number: 004 843 389

This is a closed book, closed note test. Answer all questions.

Each question should be answered in 2-5 sentences. DO NOT simply write everything you remember about the topic of the question. Answer the question that was asked. Extraneous information not related to the answer to the question will not improve your grade and may make it difficult to determine if the pertinent part of your answer is correct. Confine your answers to the space directly below each question. Only text in this space will be graded. No question requires a longer answer than the space provided.

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?

The DOS FAT file system is unable to efficently store sparse files because of the way the file allocation table is set up. Since sparse files are mostly empty the table entries will not be able to handle blocks pointing to the empty space easily as empty blocks have their own convention in the table.

Unix systems do not have such problems with this because of the way the Inode block allocation list thing works. Instead of a linked list type setup it has a structure that holds pointers to blocks that belong to the file. This it can just put the blocks belonging to the Sparse file in this structure regardless of their contents.

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 allocates a new inode and has it point to an existing files inode.
A soft link does not do this. Instead, a soft link gives directions to a file using its full path.

As far as metadata goes, a hard link increases a files link counter, a soft link does not. A hard link's own metadata pretty much just tells you that it is a hard link & the INode of the linked file while a soft link has no such metadata

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?

When using capabilities in a single machine environment it is easy for the OS to give out and revoke capabilities at will. It is also easy to make sure that no capabilities get into the wrong hands. This is not the case with dist. systems. It may be very difficult to revoke a capability and give it to a different client. The original client could go down or become compromised.

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

Horizontal scalability is being able to expand a Distributed system by simply adding more machines to it. This is good because machines (servers) can be cheap and this can make scaling the Distributed system cheap as well. These machines can also fail or break individually and be replaced without much impact to the system.

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

Full disk encryption is effective against the type of attack where someone steals your HDD & tries to access it on a different machine. It is effective because without the key/software that is on the fully encrypted drives' home machine the disk will be usually completely unreadable. The malicious user would have to brute force the drives encryption before anything on it would be usable.

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 is issuing virtual addresses when it issues RAM addresses for its own use. It does this so it can take advantage of the mmu and its TLB & other virtual memory optimizations. The OS can also therefore take advantage of pages & page frames and the nice optimizations they provide (specifically not needing to find pure contiguous physical segments for the memory it needs).

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

Disk defragmentation helps with External fragmentation.
It helps with external fragmentation and not internal fragmentation because disk defragmentation is a reordering of drives occupied space. This reordering can consolidate free space eliminating small "fragments" between allocations. This however has no impact on the amount of unused space in allocated segments, which is the cause of internal fragmentation, which is why it cannot help with it.

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

User level threads can achieve preemptive scheduling by making their reads/operations well known to each other. So when a thread is going to preform critical or costly actions that may block they can instead stop and wait for other threads to complete less costly or nonblocking operations first.

Used w/ multithreading.

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

Sloppy counters solve the problem of contention & race conditions. They solve this problem by reducing mutal exclusion. That is each concurrent thread/process is given its own counter which is only added to the global counter once a certain threshold is met by the sloppy counter. This limits the amount of contention as multiple process/threads are unlikely to try and update the mutexlly exclusive global counter at the same time. The disadvantage of sloppy counters is that the global counter may be inaccurate at any single time it is measured during execution. The sloppy counters may hold value that has not been consolidated into the global counter when it is checked.

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 Create Semaphore → Create child → Parent stop → Child initialize → Parent continue/child continue

I DONT remember the exact name of the semaphore operations

So grab() = try and decrement/get semaphore & sleep if you cannot & post() = increment/release semaphore, wake the next in line to get it

The semaphore should be initialized to 0.

The parent should immediately grab() the semaphore once it creates the child. This will cause the parent to sleep as the semaphore has no value/resource to grab.

The child should then preform its initialization & then call post(). calling post will wake the parent and grab the semaphore allowing it to continue execution.

This use of semaphore achives the desired goal because the parent sleeps during the child's initialization & is promptly woken up once the child finishes & can then continue execution.

This approach works when the semaphore cannot have a value < 0. The book implements such a semaphore. In the case the semaphore must be able to go below 0 the child must simply grab the semaphore before the parents first grab() call, & the semaphore will have to be initialized to one.

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 is necessary for deadlock because it ensures that no contending processes/threads will release the resources/locks that they have already acquired, otherwise the deadlock may resolve itself. One method that can be used to avoid hold & wait is to force each process to acure all of its resources before it executes, if it cannot do so it releases everything & tries again later.

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

Asynchronous I/O is useful for systems using event based concurrency because event based concurrency is not multithreaded. That is, if an event flag is raised that requires blocking for I/O the entire process/system locks & waits for the event's I/O to complete. With asynchronous I/O the process can quickly return and resume normal execution without having to wait for the entire I/O to complete vastly reducing the amount of time spent blocked.

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?

Write caching/buffering is an optimization related to making writes to a file system preform better.

This optimization especially helps when several small changes are being made to a file, or when a file is quickly changed/created and then deleted. Caching/buffering ~~of~~ these writes can ensue that needless writes to disk are avoided, It can also help amortize write costs. Coalescing writes into contiguous data segments.

The complexities added is the need to keep and maintain buffers for multiple writes, possibly needing several buffers at once as well as the increased risk of data loss during a power failure and such.