S23 COM SCI 111 Final Review Notes

By Vincent Lin.

Kahoot Questions (with Explanations)

Question 1

Which of the following is NOT true about write buffering in the context of file systems?

- It can prevent writes to disk altogether
- It can be bypassed with the `fsync` system call
- **CORRECT:** It reduces the amount of data to write per disk operation
- It might lose your data upon power loss

▼ Explanation

With buffering, a sequence of redundant writes (e.g. writing something and then deleting it) can resolve to not having to write anything at all - this also means that when you create a quick temporary file for something, there's a good chance that file never actually existed in the persistent file system.

Write buffering is a trade-off. By working with RAM and only visiting disk once in a while, you speed up performance, but RAM is volatile, so if the power goes out, any data that was still in buffer is lost. You can use the `fsync` system call to forcibly flush the contents of the write buffer to disk. This is useful for applications that prioritize correctness, like databases, to eliminate this problem.

The purpose of write buffering is to make the MOST out of each write operation. Every operation with disk is expensive, so we use buffering to batch a bunch of writes before flushing them in one go at some later time. This also makes for more efficient scheduling.

Question 2

Suppose I want to figure out the entire set of resources a principal is permitted to access. Which authorization strategy is better suited for this?

- Access control lists (ACLs)
- **CORRECT:** Capabilities
- Public-key cryptography
- Journaling

▼ Explanation

In access control list systems, each object keeps track of what subjects have access to it. In capability-based systems, each subject keeps track of what objects they can access. It follows that capabilities are more suited for this job since it stores all objects the principal in question is permitted to access. With ACLs, you would have to visit *every* object in the system and query its control list to see if it includes the principal.

Public-key cryptography and journaling are irrelevant answers.

Question 3

Which of the following is true about DOS FAT file systems?

- **CORRECT:** Free blocks are indicated in the file allocation table
- Have significant external fragmentation
- Have significant internal fragmentation
- Can support sparse files

▼ Explanation

The essence of the FS is that the file allocation table (FAT) stores the number of the next cluster of the file it's a part of. If a cluster is not part of any file (i.e. is *free*), this is indicated by a special value in its FAT entry, conventionally the number 0.

As is generally true for any "block"-based strategy, there is limited fragmentation. There is zero external fragmentation because the space is uniformly divided into clusters. There is little internal fragmentation because the only situation is can arise is when the last cluster of a file is partially used, but this means that there is never more than the cluster size (512B) of space wasted per file.

FAT systems cannot support sparse files, which are files containing large regions of unallocated/empty data, like "holes" in them that can be used for other purposes until it actually gets populated. Due to the linked list nature of file clusters, without the help of some kind of metadata, it's difficult to mark clusters in the middle as unallocated because doing so would disconnect the list.

Also: https://web.cs.ucla.edu/classes/spring23/cs111/readings/FAT_files/dos.html

Question 4 (formerly Question 6)

Which of the following is a way(s) to reduce resource contention?

- Increase the time spent in critical sections
- Increase updates to shared data structures
- **CORRECT:** Use atomic instructions
- **CORRECT:** Use finer-grained locking

▼ Explanation

Firstly, we would want to reduce *everything* about a critical section - reduce the number of CS's, their length, how often we enter them, etc. Every time execution is inside a critical section, a thread had exclusive access to a resource with other threads contending for it. We would also want to minimize the use of shared data structures as they are just more things to be contended.

Using atomic instructions eliminates the need for a critical section entirely. Using finer-grained locking spreads requests out over more resources such that fewer parties contend each resource (lock). The classic example is your Lab 3, a thread-safe hashmap where you recognize that operations within each bucket will always be independent of those in other buckets, so we can assign each bucket their own lock.

Question 5 (formerly Question 7)

Might want to extend the time for this question.

In journaling file systems that use write-ahead logging, the order of operations in a transaction matters. Order them in a way such that both the data region and journal are left in a well-defined state even if power goes out at any step:

A. Write metadata to the journal
B. Write metadata to the data region
C. Write data to the data region
D. Commit the transaction to the journal

- ABCD
- **CORRECT:** CADB
- ABDC
- CDAB

▼ Explanation

Firstly, you want to write the data to the data region first. In general, writing before claiming we did is safe - the "data before pointer-to-data" tip. This is because that space in the data region was unused anyway, so we can put whatever we want there and the FS is still left in a defined state.

Writing metadata to the data region is more dangerous because that makes it "show up" in the FS - it gives the "pointer" to the data. Thus, writing metadata has to come after the data write no matter what.

But before we do that, remember that in write-ahead logging, we log what we're *about* to do before actually doing it. So we first journal the metadata write, and then once all the metadata is safe in the journal, we commit the transaction.

Finally, we write the metadata to the actual data region.

If power goes out after step 1 (C): It's as if nothing happened. The bytes in our allocated date region changed, but without metadata pointing to it, it's effectively garbage data with no way to get to it anyway.

If power goes out after step 2 (A): The transaction is not committed yet (no "end" block), so when recovering, we see the journal has an incomplete transaction and can retry the whole thing over.

If power goes out after step 3 (D): The transaction is committed, but the metadata was never actually written. When recovering, we see that the journal has a complete transaction that wasn't garbage collected, so we can just *replay* the transaction to completion.

Question 6 (formerly Question 8)

Fill in the blank to make this joke make sense:

> Interviewer: Explain to us ____ and we'll hire you.
>
> Me: Hire me and I'll explain it to you.

- Spin-locks
- Device drivers
- **CORRECT:** Deadlock
- Cache validation

▼ Explanation

You and the interviewer are depending on something from the other, locked in a stalemate because both of you think you can't proceed without acquiring the "resource" the other has. This is precisely what a deadlock is, where instead of people we have processes/threads and the "resource" is locks.

The other answers don't really make the joke applicable.

Question 7 (formerly Question 9)

Which of the following is true of direct memory access (DMA)?

- Can access RAM without CPU intervention
- Controlled by a scheduler that determines who uses the bus at any moment
- More efficient than memory-mapped I/O for bigger transfers
- **CORRECT:** All of these

▼ Explanation

DMA is a mechanism that allows peripherals to access memory without CPU intervention. This makes it so that the device speed is not limited by the CPU speed, and it lets the CPU focus on other tasks while DMA takes charge of data transfer. Thus, accessing RAM without CPU intervention is in its very definition.

DMA can only be performed when the CPU is not using the memory bus, so there's a scheduler that determines when DMA can make the transfer.

A useful point to remember is that DMA is better for bigger transfers or high bandwidth operations, as there is an initial overhead where the CPU helps set up the DMA before saying "you're on your own now", so it's better to make the most out of a transfer. This is better compared to memory-mapped I/O, which has a cost for *each* memory access and is better suited for lower bandwidth operations.

Question 8 (formerly Question 10)

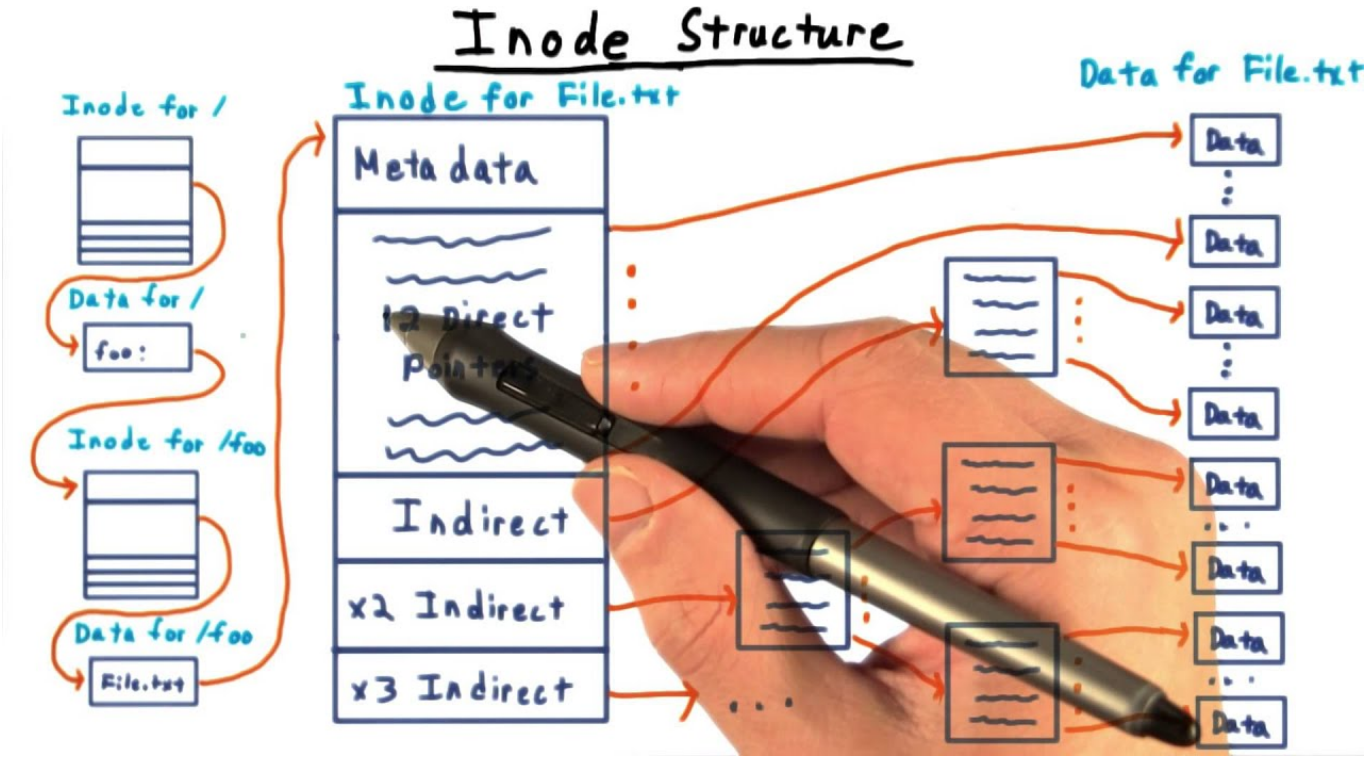
How are Unix file systems capable of supporting massive files when needed?

- Every inode is a dynamic length vector of block pointers
- Every data block is a dynamic length vector of bytes
- Modeling files as dynamic length linked lists of data blocks
- **CORRECT:** A hierarchical structure with pointers to blocks of pointers

▼ Explanation

Both inodes and data blocks are fixed size. Files are not modeled as linked lists of data blocks unlike FS's like linked extents systems or the DOS FAT FS.

Instead, Unix uses pointer indirection. The last few block pointers of an inode point to blocks of pointers, which themselves can point to even more blocks of pointers, ultimately pointing to actual storage blocks. By having this hierarchical structure of pointers to blocks of pointers, the Unix FS is able to support massive files with fixed sized inodes/blocks, and only as needed.



Question 9 (formerly Question 11)

A fun freebie!

Suppose you're designing a new role in your Discord server. Discord doesn't give out "Moderator" by default as that lets users bypass all other permission checks. You agree, so you grant specific permissions, like "Manage Messages" and "Manage Channels", but NOT ones like "Manage Server" and "Manage Roles". The Discord permission system seems to exhibit which of these security principles? (multiple answers)

- **CORRECT:** Separation of privilege
- **CORRECT:** Fail-safe defaults
- **CORRECT:** Least privilege
- **CORRECT:** Open design

▼ Explanation

"Fail-safe defaults" is hinted by:

> Discord doesn't give out "Moderator" by default as that lets users bypass all other permission checks.

This doesn't mean everything else has sensible defaults (one can argue they do though, with knowledge about the Discord permission system), but with what's given in this question, it should be enough.

"Separation of privilege" and "Least privilege" are hinted by:

> you grant specific permissions, like "Manage Messages" and "Manage Channels", but NOT ones like "Manage Server" and "Manage Roles".

We see that there isn't one giant "Manage" permission but rather separate specialized permissions for more granularity. This granularity also lets admins mix and match the permissions for a role, giving it only what it needs to do its job.

"Open design" isn't really hinted anywhere in the prompt, but I didn't know what to put as the 4th option and it still makes sense anyway - the permission system is well-documented, and every member can easily find out what they can/can't do, even if they don't have access to the admin control panel.

Question 10 (formerly Question 12)

Which of the following is true about threading?

- Increasing the number of threads increases parallelism
- Threading is only available in kernel space
- **CORRECT:** Every thread has its own stack in the process' memory space
- Only one thread per process can be on a core at any given time

▼ Explanation

When done *right*, increasing the number of threads increases parallelism, but this isn't always true. Notably, if there is significant resource contention and no way for threads to work *independently*, threading can actually be even slower due to the overhead of context switching. Furthermore, increasing the number of threads beyond the number of logical cores won't do anything since by definition, only that number of threads can be executed concurrently at one time.

Threading is available in both user and kernel space.

Each thread has its own exclusive region of the process' memory space, which is its own stack for things like local variables. Threads of the same process still share the rest of the memory space, like the code and data regions.

Threads can be independently scheduled, so there's no rule saying that only one thread per process can be on a core at any given time. If anything, smart schedulers try to have threads of the same process executing together to make better use of the shared cache.

Question 11 (formerly Question 13)

Which of the following is the best solution to preventing order violation errors?

- Locks
- **CORRECT:** Condition variables
- Leases
- Polling over interrupts

▼ Explanation

Condition variables are synchronization primitives used to efficiently wait for a specific condition to become true before proceeding instead of wasting CPU cycles polling/spin-waiting. These can enforce the order between two operations (like memory accesses) - "only if A then B".

The other answers are also useful synchronization strategies, but more suited for other problems:

- Locks more generally prevent atomicity violation errors
- Leases prevent deadlock through preemption
- Polling over interrupts help prevent livelock

Question 12 (formerly Question 15)

Which type of Client/Server model does this sound like? No set server, each node can potentially act as a client or server.

- Thin server
- Cloud services
- **CORRECT:** Peer-to-peer
- Andrew File System (AFS)

▼ Explanation

From the UPE W23 111 Final review slides:

> Types of Client/Server models
>
> - **Peer-to-peer:** No set server, each peer can potentially act as a client or a server
> - **Thin client:** Client is very lightweight, relies on a server to do most of the work and store the files.
> - **Cloud services:** Servers are opaque to clients, clients access services provided by the servers

I randomly threw in AFS for a 4th option. It doesn't even make sense.

Question 13 (formerly Question 16)

Consider this definition for some data structure. What kind of synchronization primitive does this likely represent?

```
typedef struct __Unknown_t {
    int value;
    pthread_cond_t  cond;
    pthread_mutex_t lock;
} Unknown_t;
```

- Mutex lock
- Condition variable
- Thread control block (TCB)
- **CORRECT:** Semaphore

▼ Explanation

From the OSTEP textbook (31.1):

> A semaphore is an object with an integer value that we can manipulate with two routines; in the POSIX standard, these routines are `sem_wait()` and `sem_post()`.

The most important detail is that it has an `int value`, which can hold some arbitrary signed integer. This is what makes semaphores more general then things like locks, which are binary in nature.

Secondly, we see that it's built *with* a condition variable and mutex lock, so the `struct` as a whole can't represent those but rather a composite structure, like that of a semaphore.

TCB is a random 4th option that isn't even a synchronization primitive. A TCB looks like a PCB, which would have a much larger `struct` definition with a bunch of fields related to the state of a thread.

This code is actually taken from the code that comes with the OSTEP textbook at: https://github.com/remzi-arpacidusseau/ostep-code/tree/master/threads-sema

The author provides a custom implementation of a semaphore called a "zemaphore".

Question 14 (formerly Question 17)

Which of the following is NOT an advantage of using RAID over a single disk?

- Reading from multiple disks in parallel speeds up I/O times
- Large disks means it can support large data sets
- Spreading data across redundant disks means it can tolerate the loss of a disk
- **CORRECT:** Simplicity makes it trivial to pick a RAID and its parameters for a particular workload

▼ Explanation

From the UPE W23 111 Final review slides:

> Advantages of RAID over a single disk
>
> - **Performance:** read from multiple disks in parallel to speed up I/O times
> - **Capacity:** large data sets demand large disks
> - **Reliability:** spreading data across multiple disks without RAID techniques makes data vulnerable to the loss of a single disk
>   - RAID uses redundancy to tolerate loss of a disk and keep operating

From the OSTEP textbook (38.10):

> There are many possible RAID levels to choose from, and the exact RAID level to use depends heavily on what is important to the end-user... Picking a RAID and setting its parameters (chunk size, number of disks, etc.) properly for a particular workload is challenging...

Question 15 (formerly Question 18)

What is the purpose of a password salt?

- **CORRECT:** To make dictionary attacks more expensive
- To increase the number of characters in a password
- To increase the types of characters used in a password
- To make up for a weak cryptographic hashing algorithm

▼ Explanation

A dictionary attack is where the attacker tries common words in a dictionary under the assumption that users use them in their passwords. Without salting, identical passwords have identical hashes. This means that if the database of hashes is leaked, cracking a password for one account cracks all accounts with that same

password. Salting makes it so that identical passwords still have different final hashes, making the attack much harder.

A salt is only used during encryption and decryption of the password. It's often stored *with* the password but is not part of the password, so it doesn't change its length or variety of characters.

The "strength" of a cryptographic hashing algorithm isn't relevant here.

This question was from a previous 111 Reiher final (S20).

Question 16 (formerly Question 19)

Which of the following is NOT true about remote data access?

- RPCs provide a mechanism for invoking functions on remote machines
- **CORRECT:** Remote data access should appear different from local data access
- Remote file transfer requires no OS support
- Remote disk access can't support file sharing amongst clients

▼ Explanation

Remote procedure calls (RPCs) turn procedure calls/functions into a message sent to another computer to compute and return its result, so the first option is is a true statement.

Perhaps the most important goal of remote data access is that it should appear *indistinguishable* from local data access - this is called *transparency*. The second option is a false statement and thus the correct answer.

The options about remote file transfer and remote disk access come straight from the UPE W23 111 final review slides:

> Remote file transfer
>
> - Relying on non-OS based protocols to query for data from the server
> - Pros
> - Requires no OS support
> - Cons
> - Latency
> - No transparency

> Remote Disk Access
>
> - The remote server is seen as a local disk
> - Typical architectures:
> - Storage Area Network (SCSI over Fibre Channel): fast, expensive, moderately scalable
> - iSCSI (SCSI over ethernet): moderate performance, cheap, scalable
> - Pros
> - Transparency
> - Leaves performance/reliability/availability problems to the server
> - Cons
> - Inefficient fixed partition space allocation
> - Can’t support file sharing amongst clients

- Message losses due to network errors become file system errors