CS 111 Final Potential Questions

Terms and concepts not covered in doc yet:

- METRICS AND MEASUREMENT/LOAD AND STRESS TESTING
- POSIX file consistency (read after write consistency)
- Open after close consistency
- active/active, active/standby
- Copy on write (COW)
- Namespaces
- HTTP
- Remote disk access, data access, file access
- Loosely coupled vs tightly coupled
- Add more **or** remove if its added

Readings Not Yet Covered (pls contribute! ω it's all of us vs. Kampe)

- ❏ Metrics and Measurement
- ❏ Load and Stress Testing
- ❏ Authentication Services
- ❏ SMP Scheduling
- ✓ Eventual Consistency
- ✓ Multi-Processors
- ❏ Clustering Concepts
- ✓ Horizontally Scaled Systems

Formatting Guide

- Chapter/reading names are Heading 2
  - PC: Ctrl+Alt+2
  - Mac: Cmd+Alt+2
- Question names are Heading 3
  - PC: Ctrl+Alt+3
  - Mac: Cmd+Alt+3
- Question ANSWERS are Normal Text but indented to the right by one
  - PC: Ctrl +]
  - Mac: Cmd +]

Peep that document outline (Tools > Document Outline) for quick navigation

Week 6: Partho boi
Arpaci 32
Deadlock avoidance, Health monitoring, java synchronization, java intrinsic locks, Metrics and measurement, load and stress testing

Week 7: Tanzeela
Arpaci 33-38: Events, Intro to storage, devices, disks, RAID,
Arpaci 39-40: Files and filesystems, object storage, FAT file system, FUSE intro

Week 8: Jiwan
Arpaci 41-44: FFS implementation, crash consistency, Logging FS, Data integrity, Defragmentation

Week 9: Jeffrey Chan
~~Reiher intro to security: Authentication, Access control, cryptography~~
~~Arpaci 47: Distributed Systems,~~ DS security, ~~DS communications~~, RESTful interfaces
~~Sun's network FS~~, SSL, Leases(temp. locks), Two phase/three phase commits

Week 10: Partho boi 2.0
Arpaci 49: Andrew FS, Remote data consistency, robustness, security, ACID semantics
Virtual machines, distributed computing, tightly coupled systems, eventual consistency, horizontally scaled systems

Midterm Questions

Q1. What can cause a running process to be preempted?

The OS's timer interrupt occurs, indicating the end of the process' time-slice.
The processes' priority drops below anothers'
A higher priority process becomes runnable

Q2. What are two things can cause a running process to be blocked?

When a page has to be swapped in from disk into process's virtual address space.
The process initiates a disk read which may take a long time to complete.

Q3. How can one prevent a resource contention convoy with threads?

Allow "cutting in line" -- if a thread wants a lock and it's available, let it just take it instead of queueing behind other threads; optimizes for progress at the cost of fairness.
Use finer-grained locking -- divide the critical section into smaller critical sections each of which has its own lock.

C32: Concurrency Problems & Deadlock

Q1. Explain and fix the atomicity violation in the following code with a mutex.

```
// Thread 1
if (thd->proc_info != NULL){
   // #1
   fputs(thd->proc_info) // #2
   // #3
}

// Thread 2
thd->proc_info = NULL
```

Suppose thread 1 begins executing, but gets context switched at #1 to thread 2, which changes the value of thd->proc_info to NULL, which later causes an error at #2, even though the code accounted for a NULL value of proc_info.

This code assumed that the if block will be executed atomically, which often may not be the case. This can be solved by adding locks in the right places.

```
// Thread 1
pthread_mutex_lock(&lock);
if (thd->proc_info != NULL){
   fputs(thd->proc_info) // #2
}
pthread_mutex_unlock(&lock);

// Thread 2
pthread_mutex_lock(&lock);
thd->proc_info = NULL
pthread_mutex_unlock(&lock);
```

Q2. What is the order violation bug? What is one possible solution to this? What are the limitations to this solution?

```
Thread 1::
void init(){
   mThread = PR_createThread(mMain, …);
   …
}

Thread 2::
void mMain(){
   …
   mState = mThread->state;
```

```
}
```

This assumes that the struct mThread will be initialized before it is accessed, ie code block of T1 gets executed before T2's, which may not always be the case.

We can force the threads to execute in a certain order using locks and condition variables:

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
int mtInit = 0; // Could also use mThread as the state var

Thread 1::
void init() {
      ...
      mThread = PR_CreateThread(mMain, ...);
      // signal that the thread has been created...
      pthread_mutex_lock(&mtLock);
      mtInit = 1;
      pthread_cond_signal(&mtCond);
      pthread_mutex_unlock(&mtLock);
      ...
}

Thread 2::
void mMain(...) {
      ...
      // wait for the thread to be initialized...
      pthread_mutex_lock(&mtLock);
      while (mtInit == 0)
            pthread_cond_wait(&mtCond, &mtLock);
      pthread_mutex_unlock(&mtLock);
      mState = mThread->State;
      ...
}
```

Q3. What are the requirements for a deadlock to form?

1. **Mutual exclusion**
    a. Threads claim exclusive control of resources that they require (e.g. thread grabs lock)
2. **Hold-and-wait**
    a. Threads hold resources allocated to them (e.g. locks they have acquired) while waiting for additional resources (e.g. locks they wish to acquire)
3. **No preemption**
    a. Resources (e.g. locks) cannot be forcibly removed from threads that are holding them
4. **Circular wait**

a. There exists a circular chain of threads such that each thread holds one or more resources (e.g. locks) that are being requested by the next thread in the chain

Q4. Describe a scenario in which a deadlock bug arises.

Deadlock occurs when a thread is holding a lock (L1) and waiting for another (L2); at the same time, another thread is holding (L2) and waiting for (L1)

```
void thread_1() {
      pthread_mutex_lock(L1);
      pthread_mutex_lock(L2);
}

void thread_2() {
      pthread_mutex_lock(L2);
      pthread_mutex_lock(L1);
}
```

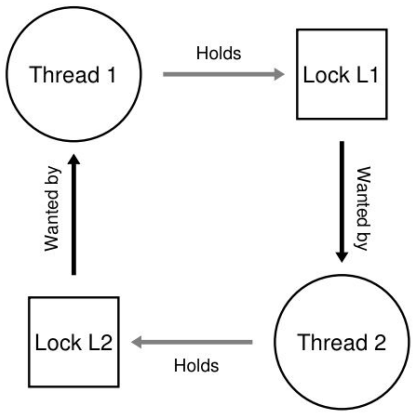


Figure 32.2: **The Deadlock Dependency Graph**

Q5. Describe a technique to avoid the circular wait condition of deadlocks.

Use an ordering system.
Total (strict) ordering -- have a strict order in which locks must be acquired.
Partial ordering -- have a non-strict but carefully designed order in which locks must be acquired.

```
if (m1 > m2) { // grab locks in high-to-low address order
      pthread_mutex_lock(m1);
      pthread_mutex_lock(m2);
} else {
      pthread_mutex_lock(m2);
      pthread_mutex_lock(m1);
}
// Code assumes that m1 != m2 (it is not the same lock)
```

Q6. Describe a technique to avoid the hold-and-wait condition of deadlocks.

Acquire all locks atomically.
Unfortunately, this requires knowing which locks must be needed ahead of time when calling modules. Also, this **decreases concurrency** since locks are acquired earlier than needed.

```
pthread_mutex_lock(prevention); // begin lock acquisition
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...
pthread_mutex_unlock(prevention); // end
```

Q7. Describe a technique to avoid the no preemption condition of deadlocks.

Take an all-or-nothing approach.
If you can't lock the second lock, unlock the first one.
This could lead to livelock -- two threads repeatedly attempting this and getting stuck (can be avoided with a random delay before trylock())

```
top:
      pthread_mutex_lock(L1);
      if (pthread_mutex_trylock(L2) != 0) {
            pthread_mutex_unlock(L1);
            goto top;
      }
```

Q8. Describe a technique to avoid the mutual exclusion condition of deadlocks.

Design various data structures without locks at all using powerful hardware instructions.
This can be very difficult for all but a few basic data structures.
An example: thread-safe linked list insertion without locks (using CAS)

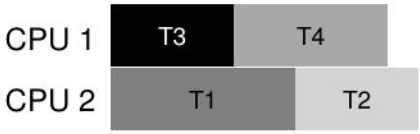
```
void insert(int value) {
      node_t *n = malloc(sizeof(node_t));
      assert(n != NULL);
      n->value = value;
      do {
```

```
            n->next = head;
        } while (CompareAndSwap(&head, n->next, n) == 0);
}
```

Q9. How can we avoid deadlock using scheduling?

Have the scheduler not run deadlock-creating threads together.
This can come at a steep cost of performance but is a tradeoff we may have to make to avoid deadlock in some situations.

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no



Q10. Why is strict ordering difficult to implement large code bases? Give an example.

One reason is that in large code bases, **complex dependencies arise between components**. Take the operating system, for example. The virtual memory system might need to access the file system in order to page in a block from disk; the file system might subsequently require a page of memory to read the block into and thus contact the virtual memory system.

Thus, the design of locking strategies in large systems must be carefully done to avoid deadlock in the case of circular dependencies that may occur naturally in the code.

Another reason is due to the nature of **encapsulation**. As software developers, we are taught to hide details of implementations and thus make software easier to build in a modular way. Unfortunately, such modularity does not mesh well with locking. As Jula et al. point out, some seemingly innocuous interfaces almost invite you to deadlock.

Example:

- main memory is exhausted.
- we need to swap some processes out to secondary storage to free up memory.
- swapping processes out involves the creation of new I/O requests descriptors, which must be allocated from main memory.

C33: Events

Arpaci C33-33.6 (events)

Q1. What is event based concurrency?

Wait for event to occur, check for the type of event, and process. The basic event loop looks like

```
while (1) {
      events = getEvents();
      for (e in events)
            processEvent(e);
}
```

Q2. What does the select() API do?

It examines the I/O descriptor sets whose addresses are passed in readfds, writfds, errofds. The descriptors from 0 through nfds-1 in the descriptor sets are examined. On return, select() replaces given descriptor sets with subset consisting of those descriptors. Lets you check whether descriptors can be read from and written to.

Q3. Why are no blocking calls allowed in event-based systems?

There are *no threads to run in the event-based approach*. The entire server will issue a call that blocks, resulting in the system being idle and wasting resources.
Instead, an event-based system uses *asynchronous I/O*.

Q4. What is asynchronous I/O?

OS interface that enables an application to issue an I/O request and return control immediately to the caller, before the I/O has completed

Q5. How do OS systems approach a program that has hundreds of I/Os issued at a point(repeatedly or wait checking?)

Some systems use interrupt-inspired *signals* to inform apps that an async I/O completes, removing need to repeatedly a sk the system.

Q6. When reading asynchronously, how does the event-based server know what to do?

Continuation: record the needed information to finish processing this event in some data structure. When the event happens, look up the needed info and process the event.

Q7. What problems does event-based programming face?

To take advantage of multiple processors, event server has to run multiple event handlers in parallel, so usual synchronization problems happen.

If **page faults**, it will block, and the server won't make progress until the page fault completes. Implicit blocking due to page faults hard to avoid and can lead to performance problems.

Programmers must always be on the lookout for changes in semantics and APIs that use **blocking** (which is disastrous for event-based handling).

Arpaci C35 (intro to storage)

C36: I/O Devices

[Arpaci C36 (devices)]

Q1. Describe the two important components of a canonical device?

The hardware interface -- allows the system software to control its operation.

The internal structure -- implementation specific.

Q2. What is the simplified device interface for a canonical device comprised of?

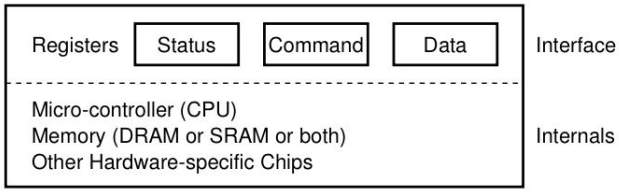


Figure 36.2: **A Canonical Device**

- status register read to see the current status of device
- command register tells devices to perform a specific task
- data registers passes data to the device

Q3. Describe the protocol of an interaction between an OS and a canonical device.

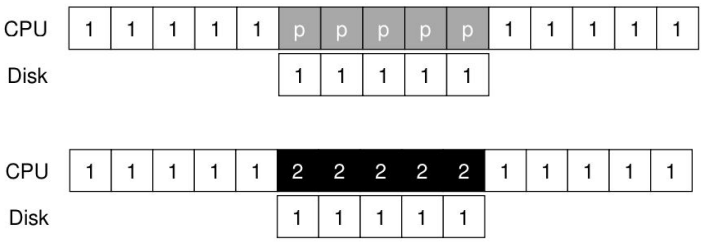
1. OS waits until device is ready by polling the device
2. OS sends data to data register. (When the main CPU is involved in data movement, it's called **programmed I/O**)
3. OS writes command to command register, implicitly letting device know that both the data is present and it should begin.
4. OS waits for the device to finish by polling

Q4. What's a better alternative to constantly polling the I/O device (step 1 above)?

Use an interrupt:

1. OS issues a request on behalf of the process
2. OS puts calling process to sleep
3. OS context switches to another task
4. When device is finished with operation, raises hardware interrupt
5. Causes CPU to jump to OS at pre-determined interrupt service routine (ISR) (i.e. interrupt handler)
6. ISR finishes request by reading data and waking the process waiting for I/O

Interrupts are better than just polling because they allow for computational overlap while waiting for I/O.



With polling

With interrupts

Q5. What are reasons not to use interrupts?

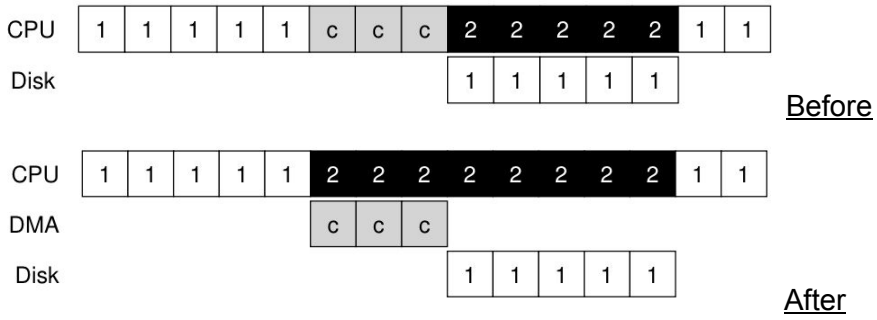
Interrupts are generally **only good for slow devices**. Depending on the situation, the cost of interrupt handling and context switching may outweigh the benefits interrupts provide.

If there is a flood of interrupts (e.g. incoming network packets), **livelock** is possible: OS only processes interrupts and never allows a user-level process to run actual service requests.

It may be better in some cases to use a **two-phased hybrid** approach: poll for a little while, then use an interrupt if device still not done.

Q6. Describe Direct Memory Access (DMA)

A very specific device that allows for transfers between devices and main memory without much CPU intervention. The *OS would program the DMA to know where the data lives in memory, how much data to copy, and which device to send it to.* OS can proceed with other work. When DMA completes, DMA raises interrupt and OS knows transfer is complete.



Q7. What are the two primary methods of communication between the OS and the device?

Explicit I/O: instructions that specify a way for the OS to send data to specific device registers. E.g. x86 “in” and “out” instructions. This is usually privileged.

Memory-mapped I/O: hardware makes device register available as if memory locations. OS simply performs writes to those memory addresses; hardware routes to device.

Q8. Describe the protocol of interaction between the device driver and the IDE.

1. Wait for the driver to be ready.
2. Write parameters (e.g. sector count, logical block address, etc.) to command registers.
3. Start the I/O (by issuing read/write to command register).
4. Data transfer to data port.
5. Handle interrupts.
6. Handle errors

C37: Disks

Arpaci C37 (disks)

Q1. What is the unwritten contract between clients and disk drives?

Accessing two blocks that are near one another on driver's space is faster. Accessing blocks in contiguous is the fastest access mode.

Q1. Describe the basic geometry of a modern disk.

A **platter** is a circular hard surface coated with magnetic material, bounded to other platters by a motor-based spindle.

Each platter has two sides, each of which is called a **surface** -- contains rings of tightly packed together **tracks**, where data is encoded. Each surface has a disk head attached to disk arm that reads and writes to the magnetic patterns on the disk.

The platters are all bound together around the **spindle**.

Q2. What is rotational delay?

Disk waits for desired sector to rotate under the disk head. If delay is R, the disk has to incur a rotational delay of R/2, on average, for a randomly selected sector.

Q3. What is a seek?

When the drive must move the disk arm to the correct track. This has many phases: acceleration, coasting (moving at full speed), deceleration (called the settling time).

Q4. What are the two types of write cache semantics for a drive?

In the case of **write back caching** (immediate reporting), the drive acknowledges the write has completed when it puts the data in its memory. This can be *faster* but *more dangerous* and lead to various problems.

- Most disks flush the cache anywhere between 5 and 30 seconds. This time represents a **performance/durability tradeoff**.

In the case of **write through caching**, the drive acknowledges write completion when it has actually been written to disk.

Q5. What is shortest-seek-time-first (SSTF)?

This is a form of disk scheduling that orders the queue of I/O requests by track, picking the requests of the nearest track first. This is a problem to the host OS because it sees the drive geometry as an array of blocks. In this case, the OS implements nearest-block-first (NBF). Starvation is also an issue, for example if there are continuous requests to the current rack then requests to another track (that requires head movement) will never get fulfilled.

Q6. What is Elevator/F-Scan/C-Scan?

The algorithm involves making sweeps across a disk, servicing requests across tracks. F-Scan freezes the queue to be sweeped. C-Scan sweeps from outer to inner and then resets at the outer track to begin again.

Q7. What is Shortest Positioning Time First (SPTF)?

If seek time is much higher than rotational delay, shortest-seek-time-first is fine. In the cases that seek is faster than rotation, would make sense to seek further to request service.

Q8. What is anticipatory disk scheduling?

Whenever the OS has to issue write requests to the disk, it has a few options. It can issue a single write at a time, whenever it receives a write request (or it can issue the “best” request, where “best” is determined by the scheduling algorithm). On the other hand, it can issue a small batch of requests that the disk could schedule with its internal scheduler (most modern disks have internal schedulers). Doing this may require waiting for a few more write requests from processes. This is anticipatory disk scheduling (the opposite of which is a **work-conserving** approach).

C38: RAID

[Arpaci C38 (RAID)]

Q1. What criteria do we use to evaluate a RAID (redundant arrays of inexpensive disks)?

You can remember them with acronym CPR

1. Capacity
    a. Given set of N disks with B blocks each, how much useful capacity available to clients?
2. Performance
    a. Depends heavily on workload presented to user
3. Reliability
    a. How many disk faults can the given design tolerate?

Q2. Describe RAID-0: striping.

Spread the blocks of the array across the disks in a round-robin fashion. This allows for greater parallelism when requests are made for contiguous chunks of the array. We call the blocks in the same row a **stripe** -- eg blocks 0, 1, 2, and 3 below.

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Q3. Evaluate RAID-0.

Capacity: perfect; N•B blocks of useful capacity
Reliability: Terrible, any disk failure will lead to data loss
Performance: Excellent, all disks utilized, often in parallel

Q4. Describe RAID-1: Mirroring

Simply make more than one copy of each block in the system and place each copy on a separate disk, thus letting us tolerate disk failures.

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

This example uses striping as well and is often called RAID-10

Q5. Evaluate RAID-1.

Capacity: Expensive, only obtain half of our peak useful capacity
Reliability: Worst case - loss of one data block, best case - loss of half the data blocks

Performance: For reads, it's the same. For writes, suffers worst-case seek and rotational delay of the multiple copies it must write to (twice as worse as RAID-0).

Q6. Describe RAID-4 with Parity.

Keep a block that contains the XOR of the blocks before it, indicating if the number of 1's contained in them was even or odd. This can later be used to detect corruption and even recover from a failure.

-

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Q7. Evaluate RAID-4.

Capacity: Uses one disk for parity for every group of disks it is protecting. Reduces useful capacity for a RAID group to (N-1)•B. Still pretty good.

Reliability: Tolerates EXACTLY one disk failure; if more, simply no way to reconstruct the lost data

Performance: Good for sequential large writes (bad for random writes) but updating the parity block poses an overhead that may be pretty large in the case of small random writes.

- This is because the parity blocks are all on the same disks, so these random small writes, even though they can complete it parallel, must still wait to access the disk where the parity block is.

Q8. Describe RAID-5 (rotating parity) and the motivation behind it.

To address the small-write problem (described above), RAID-5 rotates the parity block across drives.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Q9. Evaluate RAID-5.

Nearly identical to that of RAID-4. Only difference is random read performance is a little better because we can utilize all disks and random write performance improves noticeably as it allows for parallelism across requests.

- Now that the parity blocks are on different disks, the small-write problem is solved because (usually) they will be able to access their parity blocks in parallel.

C39: Files

Arpaci C39 (files)

Q1. How does open() create a file?

Passing it the O_CREAT flag creates a new file.. It also takes O_WRONLY, O_TRUNC. It returns a file descriptor, which is a pointer to an object of file type.

Q2. Describe lseek(). What is it useful for doing?

Lseek() changes the value of the file offset variable within the kernel (has nothing to do with seek). Useful for reading from random offsets w.i the document. It takes three arguments: filedes, offset, and whence (how seek is performed).

Q3. Describe fsync().

When called, the file system forces all dirty (not written) data to the disk. Returns when all of these writes are complete. This can be necessary because a write() call just tells the file system to write the data to persistent storage *eventually* but the FS usually buffers such writes in memory.

Q4. How does renaming work?

The function writes a the new version of the file under a temporary name, which is forced to disk with fsync(). When written, the temporary file is renamed. The new file is swapped into the correct place at which the old version is deleted. This last step is atomic and thus ensures all-or-nothingness.

Q5. Rewrite ls.

Such an example program may include opendir(), readdir(), and closedir(). A simple loop reads one directory entry at time, printing the name and inode number of each file in the directory. The program might stat() each file to get info.

Q6. What does link do?

It creates another name in the directory you are creating the link to, refers it to the same inode number. The file is not copied, but there are two names that refer to the same file. There is a reference count in the inode to track the files, used for unlink().

Q7. What does ln -s do?

It creates a symbolic link, which is actually a file of a different type. Removing the original file will lead to a dangling reference.

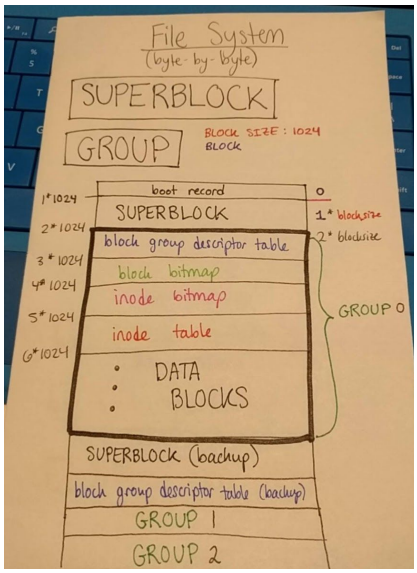
Q8. Which of the following are valid observations about hard links? (from Quiz 13)

- ✓ adding a hard link can prevent a file from being deleted.
- ✓ adding a hard link increments the reference count on an I-node.
- ❏ adding a hard link creates a new copy of the file, with a new name.
- ❏ a hard link is another directory entry (name) referring to another file (by name).
- ❏ a hard-link may be left dangling if the underlying file is deleted.
- ✓ a hard link is another directory entry (name) referring to the same I#.

C40: File Systems Intro

[Arpaci C40 (file systems)]

Q1. Describe the overall organization of file system.

The disk is divided into blocks. The data region is for user data. Metadata tracks information about each file. Inodes store access time and access permission, size and metadata. The disk has an inode table. A bitmap uses each bit indicate whether the corresponding object/block is free or in use. The superblock contains info about the file system. 

Example: ext2 file system -- shoutout Elena

Q2. How is the sector address of an inode block calculated?

```
blk = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

Node has info about its type, size, number of blocks allocated to it, protection info, time information, when the file was created and modified <---- metadata

Q3. What is the indirect pointer?

The pointer points to a block that contains more pointers, each of which points to data. An inode may have a fixed number of direct pointers and variable indirect pointers. For larger files, you may need a double or triple indirect pointer.

Q4. Describe the pre-allocation policy.

Some Linux systems will look for a sequence of blocks that are free when a new file is created by finding a sequence of free data blocks and allocating them to the newly-created file. This guarantees that the portion of the file be contiguous on the disk, improving performance.

Q5. What are the steps of reading a file from disk (ex: /foo/bar)?

The file system needs to find the inode for bar. It first traverses the pathname from the root directory, which has a known inode number of 2. The FS then looks inside of the root inode for pointers to data blocks. On disk pointers are used to read through the directory, looking for foo. Then the FS finds the inode number. This recursively continues until bar's inode is found. Once permissions have been granted, the read() is issued at offset 0, which will read in the blocks of the file.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			**read**			**read**				
				read			**read**			
					read					
read()					**read**					
					write					
read()					**read**					
					write					
read()					**read**					
					write					

Q6. What is dynamic partitioning?

OS integrates virtual memory and file system pages into a unified page cache. Memory can be allocated more flexibly across virtual memory and the file system. For file systems, this cache is key in preventing excessive reads from disk (just look at how many reads go into a simple open() call in the diagram above).

Q7. Why do files buffer writes?

By delaying writes for 30-50 seconds, the file system can batch some updates into a smaller set of I/O's. By buffering writes into memory, system can schedule the subsequent I/Os and increase performance. Some writes are avoided altogether if app deletes it within the delay time. On the other hand, databases force writes to disk with fsync().

File types

Q1. Describe the binary stream of varying ordinary files.

A text file is byte stream broken into lines, and rendered in characters. An archive (zip or tar) is a file that contains others. A load module has a different sections that represent different parts of the program. MPEG contains compressed program info that needs processing. These are all blobs of ones and zeros.

Q2. What are a few approaches to classing files?

The simplest is based on suffix (.png, .txt). Another approach is a magic number that determines the file type.

Q3. What's the difference between a directory and an ordinary file?

Directories contain name-spaces and names associated with blobs of data, similar to key-value stores. They are highly structured. Most directory operations implemented within the OS.

Q4. What type of I/O devices fit into a byte-stream access model?

Something comprised of thousands of GPUS will be mapped by gigabytes of display memory and control registers into our address space and manipulate them directly. Primitive devices will be dealt with directly (digital and analog signals).

Q5. What attributes do Unix/Linux files have?

Type, ownership, protection, when file created, and size

Key-Value Stores(Intro, types)

What are some examples of key-value databases?

Redis was very popular. Oracle NoSQL also provides such. A key has multiple components, specified as an ordered list. The major key identifies the entity and consists of the leading components.

Dynamo

Object Storage(Hist, Arch)

What is object storage?

This architecture manages data as objects, unlike a file hierarchy. Each object has data and metadata. It can be implemented at devices level, system level, and interface level. Allows for retention of unstructured data.

As opposed to fixed metadata in file systems, object storage provides for what?

Metadata can capture application-specific or user specific info for better indexing. It can centralize management of storage across many individual nodes and clusters and optimize metadata storage.

What applications for programmatic interfaces is object storage helpful for?

CRUD functions for read, write, and delete. They also support object versioning, object replication, and move between tiers of storage.

Describe OSD version 1.

Objects are specified with a 64-bit partition ID and 64-bit object ID. They are not fixed sizes, objects grow to physical limitations of device.

Describe OSD version 2.

This generation added support for snapshots, collections of objects. A snapshot involves all the objects in partition copied into a new partition. Collection contains the identifiers of other objects

FAT File System

What basic data structures do all file systems have?

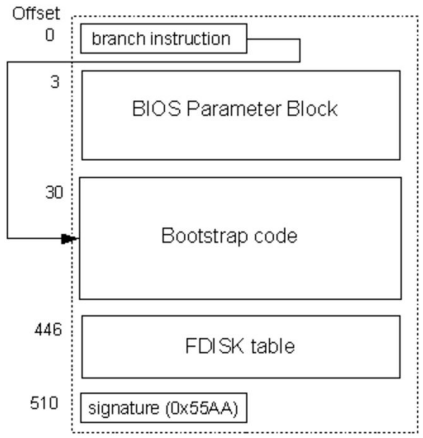
Bootstrap, code to be loaded into memory and exec when computer is on. Volume descriptors that describe size, type, and layout of the file system. File descriptors which describe a file. Free space descriptors which are blocks of unused space. File name descriptors are data structures that user-chosen name within each file.

What is a DOS FAT file system?

Comes with **file allocation table** which is a free list that keeps track of which blocks have been allocated to which files. The remainder of the volume is data structures, allocated to the files and directories.

Describe a boot record.

It begins with branch instruction, followed by a volume description, read bootstrap code, optional disk partitioning table, and a signature.



How do you find a free cluster in a FAT file system?

You need to search for an entry with the value -2. We can start our search with a FAT entry after the entry for the first cluster in the file.

How do we protect users from corruption of the FAT?

MicroSoft added support for Alternate FATs. The primary FATs would be periodically copied to on of the pre-reserved alternate FAT locations.

FUSE(Intro)

Q1. What is the Filesystem in Userspace (FUSE)?

This is a software interface for Unix OS that lets non-privileged users create file systems without using kernel code.

Q2. What is FUSE used for?

It is used for writing virtual file systems, which do not store data themselves. Any resource available of FUSE implementation can be exported as a file system.

C41: Locality and The Fast File System (FFS)

Q1. What is FFS's main organizing structure, and why does it make sense?

Fast File System tries to take advantage of disk geometry by dividing the disk into a number of cylinder groups. Modern drives don't export enough info for FS to understand whether a particular cylinder is in use so instead, modern FS's organize drives into **block groups**. Each **block** was also 4KB. This larger size made writing blocks to disk more efficient since disks like long sequential I/O.

It makes sense because, for example, files in the same directory can all be in the same block groups. Because files are commonly accessed sequentially and files in the same directory are commonly accessed together, putting them in the same **block group** can reduce disk seek time

Q2. How does FFS handle extremely large and extremely small files?

Both are exceptional cases. In the "Large File exception" we basically don't let the super large file take up the whole group. Instead, we let it take up the same amount as usual of the group, and write the remaining chunks to a different group. While that will make access to this file less efficient, it was shown in the book that in the average case it is more important to be able to access files in the same directory together rather than one large file quickly (also, most files are small anyways).

For extremely small files, there is the risk of **internal fragmentation** because a small file won't take up the entire 4KB block. There are two solutions to this. One is to use **sub-blocks** of a smaller size, and write these small files to the sub blocks. Once they add up to 4 KB write it to the normal block and free up the smaller sub blocks. This does raise some concern for efficiency again though, and so we often instead **buffer** these small file writes in the hopes that we can accumulate a 4 KB block.

Q3. What "nice to haves" did FFS introduce that were adopted by future filesystems?

Symbolic links and support for long file names are two things. FFS also introduced an atomic 'rename()' operation

C42: Crash Consistency: FSCK and Journaling

For Q1-Q3, an FS must write an updated data-block bitmap, updated inode, and data block to disk.

Q1. Describe what could happen if only a single of these writes succeeds.

Only data block is written

- Data on disk, no inode pointing to it, no bitmap saying block is allocated
- As if write never occurred
- Not a problem at all from crash consistency perspective

Only updated inode written

- If we trust the new inode pointer, we will read **garbage** data from the disk
- Bitmap says this data block has not been allocated but inode says it has (**file system inconsistency**)

Only updated data-block bitmap written

- Results in a **space leak**, the data block would never be used by the file system since it's believed to be allocated but no inode pointing to it

Q2. Describe what could happen if two of the three writes succeeds.

Inode and bitmap written, but not data

- File system is consistent but data block contains garbage data

Inode and data block written, but not data-block bitmap

- Inconsistency between inode and old version of bitmap, data-block may get overwritten

Data-block bitmap and data block written, not inode

- Inconsistency again, we have no idea which file the data belongs to since no inode points to the file

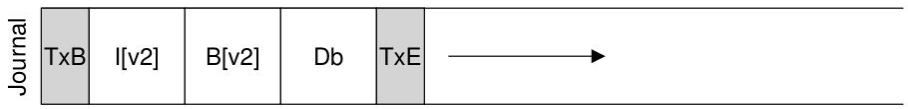
Q3. How can we check if a file system is corrupted? What problems are there with this approach?

Use the fsck UNIX tool to find and repair file system inconsistencies. Checks superblock, free blocks, inode state, inode links, duplicate pointers, bad blocks, and directories.

This approach requires meticulous programming. Furthermore, it may take prohibitively long to scan and repair an entire disk.

Q4. What is a better solution than fsck to the consistent update problem?

Use write-ahead logging (aka **journaling**). When updating the disk, before overwriting the structures in place, first write down a little note elsewhere on the disk describing what you are *about* to do (write-*ahead*). Allows you to later refer to the log if a crash occurs.



entry format

I: inode data, B: data block bitmap, Db: data block to be written
NOTE: Many file systems, including ext3, don't write the data block (Db) to journal, see Q6 below.

Q5. How can we solve the issue of a power failure occurring while journaling?

Issue the transactional write in two steps. First, write all blocks except the TxE (end marker) to journal at once. Once these complete, write the TxE block. This works due to the disk's atomicity guarantee of any 512-byte block (e.g. the TxE block). This implies that before the TxE transaction commit, if there is a failure then the update is lost.

Q6. What is the basic protocol for writing to disk in a journaling file system?

1. **Data write**: Write data to final location. This occurs first so we don't have to include the data block in the journal. Make sure you understand how this maintains FS consistency.
2. **Journal metadata write**: Write the metadata (TxB, I[v2], V[v2]) to log.
3. **Journal commit**: Write the transaction commit block (TxE), then wait for write to complete. At this point the actual data write can be considered committed, since we are 100% able to recover it in the case of a crash
4. **Checkpoint metadata**: Write contents of metadata update to their final locations within file system.
5. **Free**: Later, mark the transaction free in journal superblock.

We may also have to use **disk barriers** if our disk has its own intelligent scheduler, so that the order of this I/O does not get messed up.

Q6. Explain backpointer-based consistency

This is another way to check for file system inconsistency. An inode already points to data blocks, and then we add a backpointer to the data blocks back to the inode. If they don't refer to each other we know there's some inconsistency.

C43: Log-Structured File Systems

Q1. What is the unique approach taken by LFS to updating the disk?

Instead of overwriting files in place, LFS always writes to an unused portion of the disk, and then later reclaims that old space through a cleaning process. This process is often called **copy-on-write** and it enables highly efficient writing if updates are gathered into an in-memory segment and then written out together sequentially.

Q2. What is the key downside to LFS and how can this flaw be turned into a feature?

Old copies of the data are scattered throughout the disk and must be cleaned periodically, which can be a costly operation. Some file systems, like **WAFL**, turn this into a feature via **snapshots**, allowing users to access old files whenever they delete current ones accidentally. This is also known as a versioning file system.

Q3. How is the data in the LFS structured and updated?

The logging filesystem never overwrites data to update it, it instead writes a completely new segment for the sake of efficiency (large, continuous sequential I/O transfers are very efficient). For this reason, the LFS writes the user data and the metadata (like the inode) together, so the inode/other metadata and actual data are scattered throughout the disk.

To access the inodes then, we need a level of indirection known as the **imap** which maintains an inode -> disk address mapping (basically tells us where to find inodes). But since the imap needs to be regularly updated as well, it is also written in chunks and scattered across the disk. To find out where these imap regions are, a **checkpoint region** exists in some well known region of the LFS. This checkpoint region is only updated once every 30 seconds so we don't lose much performance.

Q4. What is the recursive update problem and how is it solved by the LFS?

The recursive update problem is a problem in all systems that don't overwrite data, instead choosing to write out a new segment whenever data is updated. Basically, whenever the inode is updated, its location also changes, so we would need to update the directory to reflect this change, but then the location of the directory would also change (since we can't overwrite that) but then the location of its parent would change … and so on until the root, causing a crazy amount of I/O having to be done for any update. The way the LFS solves this is through the **imap.** Since we update the imap each time the node is overwritten, we don't need to rewrite the directory - the imap will always hold the right address for the inode.

Q5. Why is garbage collection necessary in an LFS, and how does it work?

We need to garbage collect because otherwise all of our space will eventually become filled because we never overwrite, we re-write segments whenever updates happen. So to reclaim segments from which files have been updated (and deleted, if you're not doing

versioning FS) we need to run the garbage collector. The goal of the garbage collector overall is to read in M segments and write out N segments where N < M. Once it reads in a (partially used) segment, the garbage collector finds the dead blocks, reclaims them, and then writes out the non-dead blocks as part of another segment. The old segments are now free and can be used for subsequent writes. The specific mechanism to determine whether a block is **hot** or **cold** (unused) is by adding a **segment summary block** along with the data block. The Segment summary block contains the current inode number and an offset, the garbage collector can compare this with the current inode number for the block and see if they match.

C44: Data Integrity and Protection

Q1. Describe latent sector errors (LSE).

Occurs when a disk sector (or group of sectors) has been damaged in some way. One example is a head crash, when the disk head touches the surface for some reason (shouldn't happen in normal operation).

Q2. What are some examples of block corruption?

Bad disk firmware writing to wrong location, bus corrupts data during transfer to disk.

Q3. What is a silent fault?

A particularly insidious type of disk fault in which the disk gives no indication of the problem but returns faulty data. We want to avoid this at all costs

Q4. How can we combat LSE's?

Since they are easily detected, the storage can simply use whatever redundancy mechanism it has to return the correct data (e.g. alternate copy in a mirrored RAID).

Q5. How can we detect and combat corruption?

The primary mechanism used is called the **checksum** which takes a chunk of data and produces a small (4 or 8 bytes) summary of the data contents. This checksum is stored with the data and can later be compared with the data's current checksum. TANSTAAFL applies here because checksums need to make a tradeoff between how strong they are vs. how fast they can be computed.

For **misdirected writes** we also need to store a physical identifier (such as the block's block number and disk number) in addition to the checksum. Then we can validate this when reading data to ensure that there was no misdirected write

Q6. How are checksums stored on the disk?

Each data block gets its own checksum. One common approach is to pack N checksums into a block, then follow that with N data blocks. This is necessary because disks can only write sector-sized chunks.

Q7. What is disk scrubbing?

The system periodically (nightly or weekly) reads through every block of the system to check whether checksums are still valid. This reduces the chances that all copies of a certain data item have become corrupted.

Q8. What overheads does the use of checksums present?

Space overhead

- 0.19% on-disk space overhead with a 8-byte checksum per 4KB data block
- In memory when checksums are being read/checked (pretty small)

Time overhead

- CPU must compute each block's checksum when data is both stored and accessed

Reiher: Intro to Security

Q1. Why is security in Operating Systems important?

Everything you do runs on top of an operating system. if there's a breach/flaw in security there, it can break everything above it (pretty much everything).

There are relatively few operating systems to select from, meaning the vast number of computer systems out there are divided among these choices. A flaw in the security of one operating system can cause harm to many more lives than if there were many reliable choices to choose from.

Q2. How are interface guarantees related to and important to the security of operating systems?

OS abstractions usually have their own security methods, and present these security guarantees in their interfaces.

An example would be the file system and how it guarantees that you can't write to a file that has read-only permissions (the file descriptor **capability**).

It's important that these security guarantees are indeed enforced properly, so that systems / components built on top of the OS abstractions can utilize these guarantees in their design, and reduce the amount of complexity that comes with designing security implementations.

Q3. What are the three security goals common to any system?

*Useful (and relevant) acronym to remember: **CIA***

Confidentiality: keep secrets safe. If a piece of information is supposed to be hidden from others, don't let them find it. For example, if you don't want your credit card number to be known, it should be kept confidential.

Integrity: if a piece of component of a system is supposed to be in a particular state, don't allow an adversary to change it. For example if you order a pizza, don't let an adversary to change it to 1000 orders of pizza

- An important part of integrity is **authentication**. Not only do we want to make sure that data was not altered during transfer, but also we want to make sure the data was sent by the appropriate entity

Availability: if information or service is supposed to be available for use, make sure an attacker cannot prevent its use. For example if your store is having a huge sale, don't let attackers block off the streets to your store.

Another one is **non-repudiation**.

Q4. What are the eight common design principles when designing secure systems?

(Unlikely he'll make us list all eight. Maybe like list 2 or 3, or just describe a particular one. Just gonna put all eight here for your enjoyment)
(USEFUL ACRONYM: **"ALL OF CSE"**)

- **acceptability**: if your users won't use it, your system is worthless. A lot of promising and secure systems have been abandoned because they ask too much of their users.
- **least privilege**: give a process as little privileges necessary to complete the task they need. The more privilege you give someone, the more damage they can do
- **least common mechanism:** for different users of processes, use separate data structures or mechanisms to handle them (e.g. each process has a page table in a virtual memory system, ensuring that it can't access pages of another process)
- **open design:** assume that enemy knows every detail of the system's design
- **fail safe defaults:** default to security, not insecurity (e.g. make something private, and switch it to public later if your really need it to be)
- **complete mediation:** check if an action that needs to be performed meets security policies EVERY time it is requested
- **separation of privilege:** require separate parties or credentials to perform critical actions (e.g. two-factor authentication)
- **economy of mechanism:** keep your system as *small and simple* as possible and you'll have as little problems as possible (because increasing complexity and more lines of code increases the chances of security flaws)

Note that (as usual) many of these goals are (somewhat) at odds with each other. For example, complete mediation and acceptability.

Reiher: Authentication

Q1. Why is it important that the OS establishes a sense of identity for each process?

Each process should not be treated equally in terms of the amount of privilege it receives. For example, a program developed by a trusted company like Microsoft should be safer to grant privileges to than some random program that was downloaded from some random site.

To treat each process differently in terms of privilege, the OS needs to establish a sense of identity for each process.

One common method of establishing identity would be to associate a user's information (obtained when a user logs into a session) with each running process.

Q2. Briefly explain and give an example of each of the following: authentication based on what you know, based on what you have, and based on what you are.

Authentication based on what you know: the user is the correct user if they know a certain piece of information. The best example would be a password that the system stores, and uses to authenticate the user every time they log in.

Authentication based on what you have: the user can only obtain access and privilege if they can provide a specific item. An example would be like having an ID card to get access to a building, or having a USB drive that contains a special key that unlocks the system. Akin to a *capability*.

Authentication based on what you are: biological forms of authentication. Examples include facial recognition, fingerprints, and DNA.

Q3. During password authentication, the user's password isn't actually stored within the system. If an attacker got into the system, they would be able to just take the password. What's actually stored in the system and why is it designed so?

What's actually stored within the system is a hash of the user's password. When a user attempts to gain access via password, their input is hashed and compared to the stored hashed password.

It's designed this way because the system itself doesn't actually need to know what a user's password is, just that the user himself knows it
It's therefore important that the hashing algorithm will only make THAT specific password into the expected hash.

Q4. What are two problems that come with authentication based on what you have?

1. If you lose the object that gives you access/permission, you've lost it, and it's a huge inconvenience for the user

2. There's a possibility someone else has it, possibly a malicious entity who might try to perform malicious activities with the object now that they have it

Reiher: Access Control

Q1. Describe the access control problem.

Users make requests for valuable resources, and the OS needs to determine if such request will violate the system's security guarantees.

In other words, it's the process of determining whether a user is allowed to access a resource, and in what ways

Q2. Describe the two common mechanisms used by most access control systems.

Access control lists

- Specify precisely which subjects can access which objects in which ways. Presence or absence on the relevant list determines if access is granted.
- Analogy: Doorman at a special event that has a guest list

Capabilities

- Possession of the correct capability is sufficient proof that access to a resource should be permitted.
- Analogy: Each invited guest is given his/her own key to the door

Q3. The design principle of complete mediation states that we should check if a request meets security policies every time one is made. What are some ways to avoid running access control checks every time, and why do we do so?

One way to avoid having to run the access control algorithm every time would be to *grant subjects objects that only belong to them* (a capability). Doing so, we won't have to run checks every time because the *object is verified to belong to them from a previous access control check*. This is done with existing techniques such as virtual memory management; each process has its own object, a virtual address space.

We try to avoid running access control checks for every request for a critical resource because access control checking is expensive, and having a system run its algorithm every time results in a slow system.

(note: not sure if this is true, but i think this is related to how you store cookies in the browser to keep your authentication--this is also an example of stateful authentication (think back to statelesness/statefuleness))

Reiher: Cryptography

Q1. What are the advantages and disadvantages of public key encryption?

Advantage: Public key encryption solves the problem of how to distribute secret key to many users: you don't have to! With public key encryption, you no longer have to worry about how to transfer a secret key to a wide number of users. The system keeps a private key that is never shared, and distributes a public key to everyone else without having to worry about any security issues.

Disadvantage: Public key encryption is actually more expensive than traditional encryption performed with a single shared key. Therefore, it's important to be selective on when to use public key encryption.

(this probably gives some insight on why symmetric / session encryption is used. Public key encryption is used up until a session key is established, and every transfer done after that is done with a single key, making the transfer less expensive)

Q2. What authentication does the use of public key encryption provide?

If I encrypt a message with my private key, another user can authenticate my identity by decrypting the message with my public key, since it would only decrypt correctly if it had been encrypted with MY private key.

Q3. We can reduce the cost of integrity checking by hashing data with a hash function, and then just encrypting the hash function. What fundamental concept about hash function make them susceptible to attackers, and how do cryptographic hashes solve these problems?

Hash function, by their very concept, come with a sense of collision. It's possible for two inputs to get hashed to the same value, and that's a problem because it leaves an opportunity for attackers to provide an input that hashes to the same thing as the real thing.

Cryptographic hashes are a special type of hash functions that solve these problems by having the following characteristics:

- computationally infeasible to find two inputs that will produce the same hash value
- any change in input will result in unpredictable change in the hash value
- computationally infeasible to infer any properties of the input based only on the hash value

Q4. What is one reason why encryption would be used with capabilities in access control?

Recall that capabilities are like having a key to unlock and access the desired resource

Keys can be forged and given to other users for use, which is undesirable. In this case, cryptography can be used to create unforgeable capabilities

C47: Distributed Systems

L - latency is 0
I - infinite bandwidth
T - Topology doesn't change

A - Administrator (only one)
N - network is reliable
T - transport cost is 0
S - security (network is secure)

Q1. What are some challenges to designing a good Distributed System?

Handling Failure: This is the most important aspect of building a good distributed system. The goal is to build a system in which individual components can fail(which happens regularly), but the entire system must continue to function properly.

Performance: Performing tasks efficiently by limiting the number of messages sent over the network.

Security: Authenticating servers/clients, preventing messages from being intercepted/lost.

Q2. What is the central tenet of network communication?

The central tenet of communication is that messages get lost all the time.

Q3. What some common reasons for messages getting lost?

There are a multitude of causes for packet loss or corruption. Sometimes, during transmission, some bits get flipped due to electrical or other similar problems. Sometimes, an element in the system, such as a network link or packet router or even the remote host, are somehow damaged or otherwise not working correctly; network cables do accidentally get severed, at least sometimes.

More fundamental however is packet loss due to lack of buffering within a network switch, router, or endpoint.

Q4. Briefly describe an unreliable communication layer like UDP/IP.

- The UDP/IP networking stack is the most basic form of network communication platform.
- It employs no policies to deal with lost messages.
- Uses the socket API.

- Has some measures to check for packet corruption via checksums.

Q5. What is checksum, how does it check for packet corruption?

Checksum adds up that byte value of the entire message, then send the message and the checksum. The receiver, calculates the checksum on its own, and then compares it with the checksum sent by the sender. If the two don't match, we know that the message was corrupted during transmission.

Q6. Briefly describe a reliable communication network like TCP/IP

Uses acknowledgement to verify if the message is received:

- The receiver, upon receiving a message, sends an acknowledgement (ACK) to the sender.
- If the ACK is not received, the sender assumes that the message it sent got lost. After a certain amount of time (timeout), the sender resends the message.
- If the message was received, but the ack got lost, the sender will erroneously believe that the message was lost, and as a result will resend an already received message
- If that message is a simple assignment message, or a read request (idempotent message), it doesn't matter. But if the message is to increment a counter, it can cause problems.

Q7. Describe, in detail, two methods by which we can ensure repeated messages don't cause errors when the ack gets lost.

One way is to have a unique identifier for each message. That way, if the receiver receives a message that it already had, it can not pass it to the calling application. This, however, can be an expensive operation to have a unique ID for every message.

Another way is for the sender and receiver to have maintain a counter of message sent/received. For example...

- The sender sends a message with counter value 'n'. The receiver receives it, increments its counter from n to n+1, and sends the ACK.
- If the ACK gets lost and the sender times out, it resends the message again with counter ID n.
- The receiver sees that the counter values don't match and throw out the message, and resend the ACK.
- This time when the ack is received by the sender, the sender can increment its counter to n+1, and continue on with the next message.
- This method is known as **sequence counting**, and also has application in idempotent requests.

Q8. How is the timeout value determined?

The timeout value decides how long the sender will wait before it resends the message. If the value is too low, it the sender wastes network bandwidth sending a message that

hasn't failed to deliver, just waiting for the ACK, if the value is too high, the perceived performance of the server will be bad.

In more practical scenarios, the timeouts are calculated as an exponential function, increasing with every subsequent resend. This makes practical sense as multiple message losses implies the server is overloaded and needs time to properly send/receive all messages/acks properly.

Low timeout: The sender may think that the server is dead when really the network or the sender was just being slow. This could cause unnecessary restarts of the receiver

High timeout: The sender may not realize that the server had died for a long time (ie, the timeout length), so the restart of the server would be delayed. If the restart of the server is delayed this could result in lower perceived performance (as well as additional overhead when the server comes back to life and has to play a log)

Q9. Briefly describe Distributed Shared Memory, and outline some problems with it.

Distributed Shared Memory (DSM) is a mechanism to employ an OS like memory abstraction to distributed file systems.

Creates a virtual address space over the entire memory of individual computers in the network such that to any application running on any one of the machines would believe that it has access to the entire distributed system.

If a page of data that an application requests is already present on the local machine, the application will fetch it using the usual protocol (trapping to OS, TLB update etc.)

If the page isn't available on the local machine, the page fault handler sends a message to the machine that holds the page, fetch the page from that machine, install it in the local machine, and finally return to the calling application.

This method has a lot of flaws. Page faults are often frequent and fetching pages over a network system can be extremely slow and unreliable.

Generally to make distributed shared memory work, one tries to minimize page faults, so a node would like all the pages it needs on its machine. But doing this leads to a loss of the "distributed" property.

Additionally, in case one of the machines in the system fails, the pages on that machine are lost, unless replica machines exist or a write-log can be replated.

A benefit of DSM is that single-node failures do not affect the rest of the system's **correctness**, but they do perhaps impact performance. This method is obsolete.

Q10. Briefly describe what a Remote Procedure Call is.

Remote procedure calls allow client applications to *make calls to functions that don't exist in the client application*, but as if they were a local function. This is implemented through the two components of an RPC: **stubs** and **run-time libraries**

Stubs are generated both for the client side and server side, and allow the abstraction of external users seeing a normal procedure call, but under the hood implement the mechanisms needed for client/server message sending. The **run-time** library is responsible for handling things like **naming** and deciding the **transport-level protocol** to be used for message sending. See below for details.

Q11. Describe in detail the client and server side code algorithms of the Stub Generator.

Client Code:

- Create a message buffer: just a contiguous array of bytes
- Pack the needed info in the message buffer: info like an identifier for the function, function arguments. The process of putting all useful info in the buffer is called Marshalling or Serialization of the message
- Send message to RPC server: the communication with the RPC server is handled by the run-time library.
- Wait for reply (synchronous, so the RPC blocks until a reply is received)
- Unpack the return code and other arguments: the stub might need to unpack some of these results if they are complex data structures. -- this is called deserialization or unmarshalling.
- Return to caller

Server code:

- Unpack the received message: called deserialization or unmarshalling.
- Call into the function using the received arguments and execute!!!
- Package the results: marshalling, serialization into a single buffer.
- Send the reply: handled by the run-time library.

Q12. Discuss some challenges and limitations associated with RPC.

1. Previously there was an issue of how to communicate between machines of different endianness. In Sun's RPC package, endianness of the messages are well-defined through **eXternal data representation**
2. Interoperability challenge: ensure that different machines interact with each other well, despite running different OS's and having different ISAs. But this is a common challenge for any distributed system mechanism.
3. RPC *doesn't deal with connection/network issues*, threading, or handle failure. It is **only** an abstraction of the subroutine call. The user must deal with failure or retransmitting the call if needed. (However, the run-time library of the RPC can decide to use TCP/IP instead of UDP datagrams, which would take care of

message retransmission in the case of network failure, but obviously not server failure).

4. RPC's abstraction can be seen as **leaky** with respect to async. This is because you can't technically make an asynchronous RPC call, even if the server function you're calling into is going to do some extensive I/O or long computation. What you have to do is kind of implement this "async" yourself:
    a. Do a standard synchronous call to the server, and block until the server responds that it's either started the function or will start it sometime for sure (ie, queued it up)
    b. Once this reply from the server is received, periodically poll the server (with another RPC each time) to see if the computation is done (generally using interrupts where the server sends a message to client saying the computation is complete is not really in the RPC pattern)

Q13. Contrast Availability and Reliability.

- Availability = fraction of time that the service is available, as gauged through repeated random sampling.
- Reliability = probability of not losing my data.
- *Basically availability is concerned with the service itself, while reliability is concerned with the data that the service operates on.*

Q14. What does write-back caching do, and how does it work? How does it improve performance?

- The write back cache mechanism essentially works by taking a write (to server, or disk) request and instead holding it in a (large) cache, not flushing the write out immediately. By doing this, we can accumulate writes into the cache and do a longer batch write to the server or disk (which is more efficient than a small write). If we're writing to a remote file system, this will decrease the number of messages we need to send, giving us a huge performance win. We could also avoid some writes altogether if an application sends a write and then removes/overwrites the write. It also enables faster read-after-write consistency.
- One issue is that if crash consistency is not handled, then this is not reliable. If the client crashes before the write is actually flushed to disk or server, we've lost our data. So we need to use some sort of **log** where we write down what we're caching, and when we flush it out we write down that we flushed it. When a client restarts after crashing it can just replay the log.

Q15. Compare and contrast thin-client, P2P, and cloud service models.

- *The peer to peer model is a smaller network of devices that interact with each other and share resources.* The resources are local but still available to each other (for example, a printer). Microsoft work groups are an example of the peer to peer model. They are generally pretty small in size.

- *The thin client model is like your smartphone.* There are very few resources (ie, CPU, RAM, some storage and not much else). Most resources live elsewhere, and the thin client is good at exploiting these resources.
- *Cloud services abstract away the idea of servers and resources.* We only care about services offered (ie, data storage service with Amazon S3). Everything is exposed as a service which you call into, and when you call the service you don't really know what's happening (hence the term "cloud") - it could be talking to a million servers or not even one server.
    - Similar to the idea that we don't really care about having access to the generator that provides electricity for us, we just pay for the "service" of electrical power…

Q16. What are the benefits of a distributed file system over a remote file system?

- In a remote filesystem, the client talks to a single (primary) server where the data is located. This primary server may have one or several replicas for backups, but the operations on the data all go to one server. However in a distributed system, the client talks to many servers for operations on data/files. This improves performance since the data can be **striped** across many servers. This will increase throughput for large amounts of I/O. Striping data in general increases bandwidth and allows one to get the data faster. However, distributed filesystems are more complex and tough to implement.

Q17. Discuss remote data/file/disk access and their benefits.

- The main goal is **transparency**. It should be indistinguishable whether a file is from a remote or local machine. The efficiency/performance should also be the same. We may also want **globality,** where any client sees all of the files.
- For **remote data access** additional goals are reliability and scalability.
- Remote file transfer can be obtained through explicit commands such as scp or implicit protocols like HTTP/SMTP. The advantages of this are efficiency and no need for OS support, at the cost of latency and lack of transparency.

Reiher: Distributed Systems Security

Distributed Systems: Goals and Challenges

Q1. What are the four reasons we build distributed systems?

(Again, will probably not ask you for all four, and more likely to ask you about specific ones, but here's all the ones in the reading)

Client/Server Usage Model: there are many situations where we can get better functionality and save money by using remote/centralized resources rather than requiring all resources be connected to your computer. (E.g. Kampe bragging about his Amazon S3 bill being $5/year for using their computers/resources)

Reliability and availability: better reliability comes by combining multiple systems. If we have more resources than we actually need to provide, we might still be able to provide those resources even if a couple systems fail. If you're a company, you don't need to build out your own datacenters, redundancy and failover mechanisms, data synchronization mechanisms, or horizontally scalable systems. You can just write a check to smart people and use their better services

Scalability: as projects get bigger and require more computing power and storage, it is better to add additional systems with storage and computing ability as needed, rather than getting rid of your computer in exchange for a stronger and larger one

Flexibility: be able to choose whether to run different parts on different computers, or a simple part on multiple computers

Q2. Why is failure so much more difficult to deal with in a distributed system than in a single system?

In a distributed system our only indication that a component has failed might be that we are no longer receiving messages from it. Perhaps it has failed, or perhaps it is only slow, or perhaps the network link has failed, or perhaps our own network interface has failed. Problems are much more difficult to diagnose in a distributed system, and if we incorrectly diagnose a problem we are likely to choose the wrong solution.

Q3. How can a distributed system deal with the failure of a component that was holding resource locks?

We must find some way of recognizing the problem and breaking the locks. And after we have broken the locks we need some way of (a) restoring the resource to a clean state and (b) preventing the previous owner from attempting to continue using the resource if he returns.

Q4. Compare total ordering of operations between a single system and a distributed system. Specifically, why is it so difficult to achieve in a distributed system?

In a single system, all operations have a predictable total ordering. Distinct nodes in a distributed system, on the other hand, operate completely independently of one-another. Unless operations are performed by message exchanges, it is generally not possible to say whether a particular operation on node A happened before or after a different operation on node B. And even when operations are performed via message exchanges, two nodes may disagree on the relative ordering of two events (depending on the order in which each node received the messages).

Q5. How does the state of a distributed system differ from that of a single system?

Because of the independence of parallel events, different nodes may at any given instant, consider a single resource to be in different states. Thus a *resource does not actually have a single state*. Rather its state is a vector of the state that the resource is considered to be in by each node in the system.

Q6. What issues of heterogeneity arise when working with a distributed system?

In a distributed system, each node may be running a different ISA, OS, and/or versions of software and protocols. The components interact with one-another through a variety of different networks and file systems. The combinatorics and constant evolution of possible components render exhaustive testing impossible. These challenges often give rise to interoperability problems and unfortunate interactions that would never happen in a single (homogeneous) system.

Q7. What are Deutsch's 7 falacies of distributed computing?

(Unlikely to be asked for all 7, but Kampe did bring them up in lecture repeatedly, so worth understanding to some extent; just copied them in from the reading)

1. The network is reliable.
    - Subroutine calls always happen. Messages and responses are not guaranteed to be delivered.
2. Latency is zero.
    - The time to make a subroutine call is negligible. The time for a a message exchange can easily be 1,000,000x greater.
3. Bandwidth is Infinite.
    - In-memory data copies can be performed at phenomenal rates. Network throughput is limited, and large numbers of clients can easily saturate NICs, switches and WAN links.
4. The network is secure.
    - While not perfect, operating systems are sufficiently well protected that we (relatively) seldom have to worry about malicious attacks within our own computer. Once we put a computer on a network it becomes susceptible to penetration attempts, man-in-the-middle attacks, and denial-of-service attacks.

5. Topology does not change.
    - In a distributed system, routes change and new clients/servers appear and disappear continuously. Distributed applications must be able to deal with an ever-changing set of connections to an ever-changing set of partners.
6. There is one administrator.
    - There may not be a single database of all known clients. Different systems may be administered with different privileges. Independently managed routers and firewalls may block some messages to/from some clients.
7. Transport cost is zero.
    - Network infrastructure is not free, and the capital and operational costs of equipment and channels to transport all of our data can dramatically increase the cost of a proposed service.

In 1997, James Gosling (also of Sun) added an eighth falacy:

- The network is homogenous. This leads to the same problems as the consequences detailed in the first 3 falacies.

RESTful Interfaces

Q1. At a high level, describe what a RESTful interface is.

Representation state transfer (REST) web services is one way of providing interoperability between computers on the Internet. REST-compliant Web services allow requesting systems to access and manipulate textual representations (e.g. XML, HTML, JSON) of Web resources using a uniform and predefined set of *stateless* operations.

Q2. List and describe the constraints of a RESTful interface.

Helpful Acronym: **UCLA CS**

Uniform interface

- The idea that a RESTful interaction should clearly define the resource it is looking for, the manipulation desired, and a message containing any additional information.
- Allows independent evolution of different system components, because the sender can change the implementation of how it's generating requests and sending them to the server, and the server can change the implementation of how the request is handled, **but they still communicate in the same way**

Cacheable

- Clients and intermediaries may cache responses
- Responses must define themselves as cacheable or not to prevent clients from reusing stale or inappropriate data in response to further requests
- USEFUL for partially or completely eliminating some client-server interactions

Layered system

- A client may (transparently) be connected to an intermediary server which connects to the end server.
- Intermediary servers may improve scalability by enabling load balancing or offer other beneficial operations like security.

Client-server

Stateless

- No client context is stored on the server between requests
- Each request from any client contains all info necessary to service request

C48: Sun's Network File System (NFS)

Q1. What are the benefits of having a distributed client/server file system?

1. Provides easy **sharing** of data across clients
2. **Centralized administration**. For example, backing up files is now done by a few server systems instead of having to be done by a multitude of client computers
3. **Security** (that comes with having the servers in a locked machine room all together)

Q2. Why was statelessness important in developing distributed file systems?

Having a common state between the server and the client could cause potential problems if the server were to fail. For example, if the server stored a file descriptor for a certain client, then lost that file descriptor when it crashed, it would have no way of completing subsequent read/write requests because the server doesn't know what file the client is talking about.

The solution to that problem was having stateless communication, meaning that the client will always send enough information for the server to complete the request. That way, even if the server were to fail, any subsequent requests after recovery could be satisfied. Also it doesn't matter to which server the request goes to, making load balancing easier.

Q3. What role do idempotent operations play in NFS?

Recall that communication over an unreliable network leads to packets being dropped on the server side, or acknowledgments not making it all the way back to the sender.

Idempotent operations are operations where the effect of running the operation multiple time is equivalent to running the operation once. So in the case of failure where the client never hears back from the server, it can set a timeout and retry again without having to worry about something breaking.

Q4. Sending read/write system calls across a network causes performance issues because the network is simply not fast enough. To solve this problem, NFS caches requests to change a file, and sends them over the network all at once. What three problems arose from doing so? (The last problem arose after solving the first two… lol)

Update visibility: the timing in which updates from one client becomes visible to all the other clients

Stale cache: when a client's cached file is now inconsistent with the newly updated file, and is thus old and outdated

Server overwhelmed with GETATTR requests: After the first two problems were solved with **flush-on-close** and making GETATTR requests to the server, they found

that the server was being overwhelmed with GETATTR requests (eventually solved by implementing another client cache to refer to). Also, continually processing these GETATTR requests is a classic example of **livelock** because the server is busy but not making any progress, similar to a CPU becoming livelocked by servicing too many interrupts instead of executing user mode processes. We solved that by reducing the frequency of interrupts via interrupt coalescing, we solve this by caching the GETATTR request and keeping that cache valid for like 10 seconds (or your favorite number here).

Q5. How is the cache consistency problem solved in NFS?

To address update visibility, clients implement **flush-on-close** (aka **close-to-open**) consistency semantics: when a file is written to and subsequently closed by a client application, the client flushes all updates to the server. This ensures that a subsequent open from another node will see the latest file version. But what about nodes that already have the file in their own cache…?

To address the stale-cache problem, clients always issue a GETATTR request before using a file to check if the version of the file on the server has been modified more recently than the locally cached version they have. If so, the cached version is deleted and the next file access will go to the server.

Q6. What problems may arise out of keeping a timeout-based attribute cache in NFS?

Say machine A starts modifying file X and for the next N seconds its attribute cache tells it that it has the latest version. During A's modification of file X, machine B flushes its changes of file X, call it X(B), to the server. Then, once A finishes its modification of X and flushes X(A) to the server, machine B's changes, X(B), will be lost!

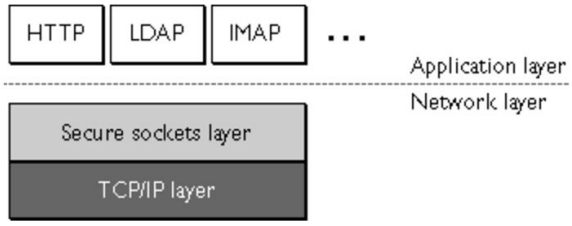
Q7. Why must writes to the server be forced to stable storage before returning success (aka write-through caching)?

If the client is informed of a write being successful, but the server crashes and restarts before completing the write, upon restart, the server will continue with the next write requested by the client but the previous write will not be reflected in persistent storage!

SSL

Q1. Where does SSL fit into a networked application that uses a TCP/IP layer?

The SSL protocol runs above TCP/IP and below higher-level protocols such as HTTP or IMAP. It uses TCP/IP on behalf of higher-level protocols, and in the process allows an SSL-enabled server to authenticate itself to an SSL-enabled client, allows the client to authenticate itself to the server, and allows both machines to establish an encrypted connection.

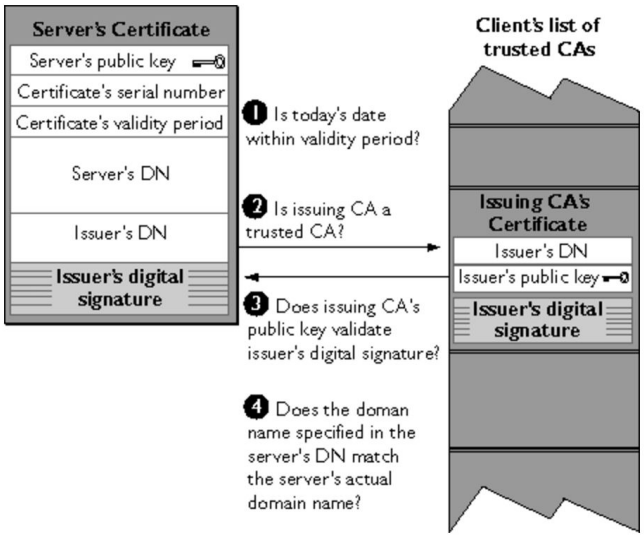


Q2. Describe the SSL handshake protocol.

1. Authenticate the server to the client.
2. Allow client and server to select cryptographic algorithms (ciphers) that they both support.
3. Optionally authenticate the client to the server. (Usually not done)
4. Use public-key encryption techniques to generate shared secrets.
5. Establish an encrypted SSL connection that uses a symmetric key.

Q3. What steps does an SSL-enabled client go through to authenticate a server's identity?

1. Check server certificate's validity period (date/time).
2. Check if the issuing certificate authority is a trusted authority.
3. Use the public key from the CA's certificate (from client's list of trusted CA's) to validate the CA's digital signature on the server certificate being presented.
4. Check if the domain name in the server's certificate matches the domain name of the server itself.
    a. Not part of the SSL protocol but is the ONLY protection against a man-in-the-middle attack
5. If everything above went well, the server is now authenticated.



Leases

Q1. Evaluate leases along the four axes used to evaluate single system methods of mutual exclusion plus the new axis of robustness.

Mutual Exclusion

- Leases are at least as good as locks, with additional benefit of potential enforcement

Fairness

- Depends if remote lock manager implements queued or prioritized requests

Performance

- Since we probably aren't going to use network leases for local objects, if lease requests are rare and cover large numbers of operations, this can be a very efficient mechanism.

Progress

- Automatic preemption makes leases immune to deadlocks! Only thing is, if a lease-holder dies, others must wait for the lease period to expire.

Robustness

- Clearly more robust than single system synchronization mechanisms.

Q2. What if a lease expires but the owner was part-way through a multi-update transaction?

The resource the owner was operating on should fall back to its last consistent state.

Q3. How do leases solve the problem of deadlock?

Progress will always be made with leases, or deadlocks will always be broken with leases, because leases can only be held for a fixed amount of time.

If it ever came down to two systems needing each other's leases to continue, they could just wait for each other's leases to expire, which is guaranteed to happen (although efficiency may be a concern, esp. in the case where a node with a lease fails and the resource the lease represents is going unused for some time.)

Distributed Consensus & (2 | 3) Phase Commits

Q1. Describe the consensus problem.

The consensus problem requires agreement among a number of processes for a single data value. Some of the processes may fail or be unreliable in other ways, so consensus protocols must be fault tolerant or resilient.

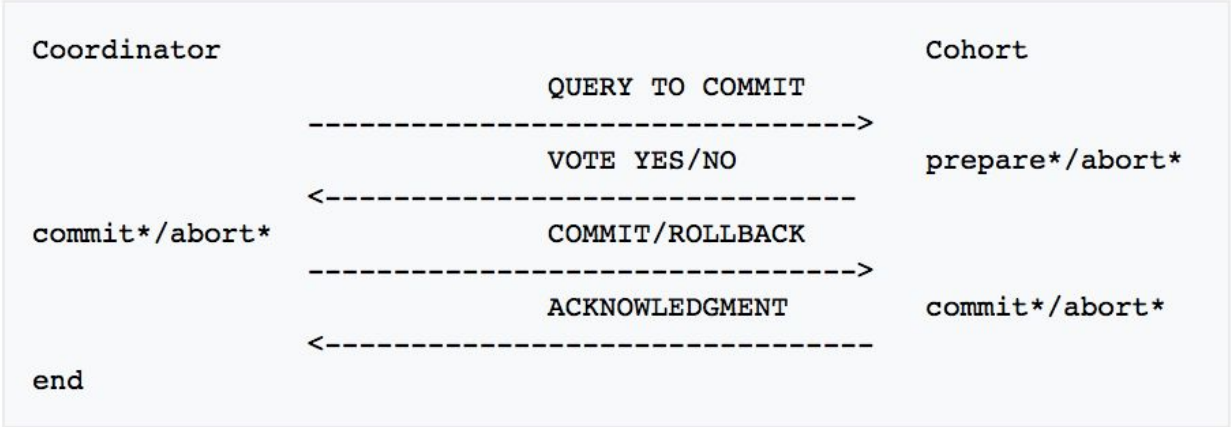
The processes must somehow put forth their candidate values, communicate with one another, and agree on a single consensus value.

Q2. Discuss how does the 2-phase commit addresses the consensus problem. What shortcomings does it have?

The 2-phase commit consists of… you guessed it, two phases:

1. **The commit-request (voting) phase**: a coordinator process attempts to prepare all the transaction's participating processes to either commit or abort the transaction. Each process votes either "Commit" or "Abort"
2. **The commit phase**: Based on the votes, if everyone voted "Commit", the coordinator commits the transaction (sending "doCommit" to the cohorts); otherwise, it aborts it and notifies the results to all the cohorts.

A two-phase commit cannot dependably recover from a failure of both the coordinator and a cohort member during the commit phase. It is also subject to unbounded wait times, in the case where the coordinator fails and does not send a message to the cohorts once it has received acks from them. Since it is a blocking protocol, after a cohort has sent an agreement message to the coordinator, it will block, and therefore hold resources and locks, until hearing back from the coordinator.



Q3. Discuss how a three-phase commit solves the issues faced by a two-phase commit.

The three phase commit introduces an additional preCommit stage which serves as an additional cohort/coordinator acknowledgement stage. If there is failure in the preCommit

stage, the cohorts assume that the coordinator died so they abort. But if all of the cohorts ack during this preCommit stage, then there is a go-ahead: the commit will occur regardless of the coordinator failing. After all preCommit acks are received, the only way the transaction can be aborted is if the coordinator sends an “abort” within the timeout frame. This eliminates the possibility of unbounded wait times.

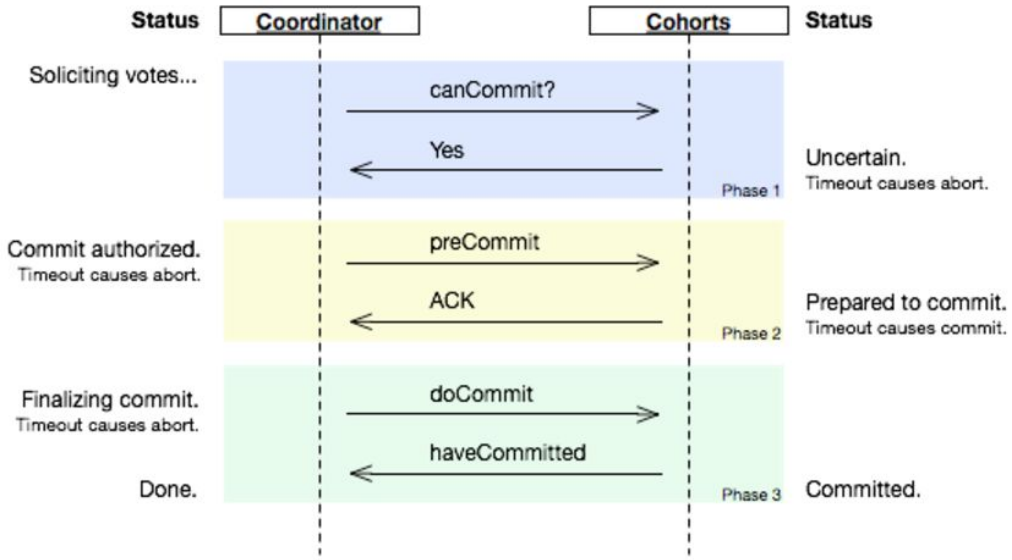
The 3 phase commit works as follows:

1. **First phase:** Coordinator asks cohorts “could you commit, like hypothetically?” If any cohorts say no then the transaction is aborted.

2. **Second phase:** If acks are received from all of the cohorts, the coordinator sends a preCommit (prepare to commit) message to the cohorts. If the coordinator fails to send this message then the cohorts abort the transaction. Now the cohorts once again have to ack the coordinator if they can do it, or nack if they can’t.

3. **Third phase:** If the coordinator has received acks from all of the cohorts, the commit is going to happen unless the coordinator sends an abort within the timeout frame. If the coordinator fails or the network is partitioned and no “abort” message arrives to the cohorts, then the commit happen anyways.

In the case of network partitions the three phase commit still does not guarantee consensus (CAP theorem). Also, the 3-phase commit can fail if one of the cohorts receives a preCommit message from the coordinator, and then both that cohort and coordinator fail.



Q4. Discuss the distributed system goals
Helpful acronym: SCARF

- Scalability
- Client/Server Model
- Availability
- Reliability
- Flexibility

C49: Andrew File System

Q1. What is the main goal of the AFS? Why doesn't the Network File System (NFS) achieve this goal?

The main goal of the Andrew File System is *scalability via minimization of client-server interaction*s (through whole-file caching and callbacks). In the NFS, the protocol forces the client to continuously poll the server if the file the client opened has changed. It does so by making the GETATTR call to the file on the server. This takes up resources like CPU cycles and network bandwidth, and this problem becomes severe as more clients access the server and hence is not very scalable.

Q2. Describe in detail the caching protocol in AFS V1. Describe the performance of a file that is frequently opened and closed by a client.

The caching on AFS works a bit differently from NFS. Instead of caching data blocks of the file that are frequently accessed by the calling application on the client, the client caches the entire file into its local disk (not memory).

Subsequent read()/write() calls are performed on the local copy of the file with no network interaction with the server.

The client employs the usual local-machine caching mechanism on the file, storing frequently accessed data blocks in an in-memory cache.

Since the local copy is preserved, if there are frequent open()s and close()es, the client works only on the local copy. Once a certain amount of time has passed and no new changes have to be made to the file, the client sends it back to the server where it replaces the old version of the file.

AFS v1 is similar to NFS in that it also required the client to issue a message, TestAuth, asking for the attributes of a file to check if it has been updated on the server. This substantially hurt performance since it forced servers to spend a lot of time telling clients it was OK to use their cached copies. This issue was fixed in AFS v2 with **callbacks**.

Q3. Briefly describe two drawbacks of AFS V1.

When the performance of the AFS was measured, these were the problems found:

- Traversing through directory after directory to reach the file was costly and time consuming
- When multiple clients accessed the same file, to maintain the consistency of their cached versions, the clients made a lot of calls to TestAuth on the file stored on the server to see if their cached version was still new (similar to problems faced by NFS). The servers spent too much time telling clients that their cached versions were still the latest versions of the file.

Q4. How are the aforementioned problems dealt with by AFS V2?

Stale Cache problem:

V2 implements **callbacks** to deal with the stale cache problem. It implements callbacks by assigning a state to each file. Consider this example where two clients have opened the same file from the server.

- Client 1 (C1) and Client 2 (C2) are both reading the file making no changes to the file on the server.
- C1 decides to write some changes to the file, and thus change the state of the file.
- The server is only notified of this change when C1 decide to close() the file, thus flushing it back to the server.
- Flushing the file from C1 back to the server changes the file's state. This triggers the callback on the server which looks for all clients that have cached that file; here its C2 that has the file cached.
- C2 deletes the file cached in it local disk, re-fetches the new version of the file from the server, resets the callback, and continues reading the new version of the file.

High path-traversal costs:

- On the first read(), AFS v2 caches not just the entire file, but also all the parent directories it is stored in. Callbacks are registered for these directories as well.

Q5. Briefly describe what happens when two clients try to modify the same file.

AFS employs the simple protocol of **last closer wins**, i.e. the last client to close() the file gets their version of the file to be persistent on the server.

Q6. Describe how Client crashes and Server crashes are handled in AFSv2.

Client crash: When a client crashes, it might miss important messages from the server. For example if a file in the crashed client gets changed by a different client and the server triggers the callback to inform the crashed client about it, that message gets lost. So after recovery, the client treats all its cached contents as suspect, ie it verifies with the server using TestAuth if its version of the file is still valid. If it isn't, it re-fetches the file, then continues regular functioning.

Server crash: Server crashes are more complicated to recover. If a client changes a file, flushes it to the server, and before the server can issue the callback it crashes, other clients will continue using the old version of that file. To deal with this, when a server recovers from a crash, it sends a message to all clients notifying them about the crash telling them to treat all their cached contents as suspect. Then, just like in the previous case, the client issues TestAuth on all files it has cached to check their validity.

The problem with that is if a server takes a long time to recover from a crash, clients can perform a large number of tasks with the old (possibly stale) versions of their files. To deal with this, the server-client often employ a heartbeat message that periodically tells each client that the server is still alive. When the clients don't receive the 'heartbeat' from the server, they stop doing any more work to avoid using old versions of files.

Q7. How is AFSv2's performance?

In a study, it was shown that AFSv2 can support up to 50 clients (~20 for NFS and AFSv1)

Since usually users of distributed file systems access the same files over and over again, and generally do operations on whole files (as opposed to sparse random access across blocks of different files), the performance of the AFS is similar to a local filesystem.

However, like all software, the performance of the AFS is limited by the assumptions that were made when designing it. AFS does do much better whenever sequential re-reads of very large files happen, because AFS has the whole file in cache while NFS does not.

However, AFS is worse than NFS when we need to do a **large file sequential overwrite** because it needs to fetch the whole file, cache it, … only to rewrite immediately (the caching was useless). AFS is also worse when we need to access a small subset of data from a very large file.

The I/O for AFS is proportional to its file access, while NFS can do less I/O if only a few blocks from a large file are needed.

Overall, the assumptions/choices for the workload are really important. For example AFS assumes that files are not frequently shared and are accessed in large, sequential batches. This is apparently usually the case. But AFS would be really bad, if for example it was used in a case where multiple clients were appending to the same log.

Q8: Several clients are writing weather-related data to a server, and are reading (from the server) weather data from other clients and displaying them to a user. We want to use a remote filesystem so the clients don't have to store any data of their own. Discuss reasons for using NFS, and reasons for using AFS.

NFS may be better if we are reading data from the server very frequently, and the data we are reading is changing a lot. This is because NFS caches on a per-block level - it would be less efficient to have to cache the whole file if we only have to update a few of the newer blocks. It would also be inefficient to have cached an entire file only to overwrite it and send it back immediately.

On the other hand, if we are not reading data very frequently it may be reasonable to cache the entire contents of the file and update that less frequently, so AFS would be better.

Basically if the updates are very frequent and small, then NFS is probably a better choice (for example, the data is being communicated once every few seconds). But if the updates are infrequent and large (for example, a summary is sent once every few minutes), but the files are commonly read, then AFS is probably a better choice.

This also depends largely on how the weather data is stored. If each client stores each data upload in its own file, AFS turns out to be a little better since it no longer faces the **file-overwrite** performance issue. However, if all of the weather data is stored in a single-shared file, NFS would excel due to its block-level caching.

Authentication Services

Q1. Discuss how Work Tickets can be used to implement authentication as a service without the client and server needing to exchange any secret keys with one another.

ACID

Q1. What is ACID?

ACID refers to properties that describe database transactions.

A sequence of database operations that follow the ACID semantics can be called a transaction.

ACID stands for Atomicity, Consistency, Isolation, Durability.

Q2. Describe each of the characteristics of and ACID operation.

Atomicity: database transactions have to be all or none. If one part of the transaction fails, then the entire transaction fails, and the database is returned to a state before the transaction started.

Consistency: characteristic that states any transaction will bring the database from one valid state to another. For example, A = 10 and B = 10 with the database constraint that A must equal B. There is a two-part transaction that first tries to increment A by 1, then increment B by 1, but this transaction fails halfway and B is never incremented. This puts the database in an invalid state, and an atomicity response will occur in which the entire transaction fails and the database is restored to a valid state.

Isolation: running multiple transactions at one time would be the same as running the same the transactions in sequential order. Basically means two transactions won't be run in parallel or else it'll cause issues of invalid state within the database and cause transactions to fail.

Durability: ensures that once a transaction has been committed, the changes will remain committed even in power loss or system failure.

Q3. What are write-ahead logging and shadow paging used for in terms of failure in ACID database operations?

Write-ahead logging: writing the data within a database into a separate database so that it can be recovered in the case of system failure.

Shadow paging: new updates are written to a partial copy of a database and aren't written into the actual database until the changes are committed

Both are techniques that use locks to try and achieve atomicity with ACID operations. Atomicity is important in handling failure with database systems.

Q4. Explain locking vs. multiversioning and how the service read/write requests in distributed database systems.

Locking is a mechanism meant to handle multiple writes to a database. Like concurrency, a system can only write to the database if it holds the lock, and this handles the case of having multiple writes to database.

Multiversioning: because the database is being written to and modified, doesn't mean that read requests should have to block or wait for the writes to finish. Multiversioning is a concept that read requests obtain a previous/unmodified version of the database, even though there are several transactions trying to modify it.

AppxB: Virtual Machine Monitors

Q1. What is the motivation behind Virtual Machine Monitors?

A VMM creates a layer of abstraction between the OS and the hardware:

- Allows the administrator to consolidate multiple OSs onto fewer hardware platforms.
- Makes it easier to test a software on multiple operating systems
- To achieve these goals, transparency is extremely important, because we still need to make the OS think it is an all-knowing-king-of-the-universe-saintly-saviour-only-thing-that-can-manipulate-hardware system

Q2. What are some things that need to be considered while virtualizing the CPU? How are these things accomplished?

- All operating systems must believe they have exclusive access to the entire hardware, much like a process running on one machine with virtualized memory has access to the entire memory.
- Since there are multiple operating systems on the machine, no one OS has complete unrestricted access to the hardware. In such a scenario, we must find a way to handle privileged instruction by an OS.
- To accomplish this, the VMM must perform a “Machine Switch”, just like an OS would perform a context switch.
- We also have to handle what happens when a privileged instruction is made by an application in one of the OSs

Q3. How does a VMM handle a system call trap from one of the OSs?

Process	Operating System	VMM
1. System call: Trap to OS		
		2. Process trapped: Call OS trap handler (at reduced privilege)
	3. OS trap handler: Decode trap and execute syscall; When done: issue return-from-trap	
		4. OS tried return from trap: Do real return from trap
5. Resume execution (@PC after trap)		

Table B.3: **System Call Flow with Virtualization**

Q4. How does a VMM handle a TLB miss?

When a process makes a VMM reference and misses, the VMM TLB miss handler handles it. Since the VMM TLB doesn’t know how to handle the miss, it jumps to the OS TLB miss handler (it knows this location at boot time). The OS TLB miss handler runs

and does a page table lookup, but this is a privileged operation, resulting in a trap to the VMM. The VMM then installs the desired VPN-to-MFN mapping, allowing the system to eventually get back to user-level code and retry the instruction, resulting in a hit.

Process	Operating System	Virtual Machine Monitor
1. Load from memory TLB miss: Trap		
		2. VMM TLB miss handler: Call into OS TLB handler (reducing privilege)
	3. OS TLB miss handler: Extract VPN from VA; Do page table lookup; If present and valid, get PFN, update TLB	
		4. Trap handler: Unprivileged code trying to update the TLB; OS is trying to install VPN-to-PFN mapping; Update TLB instead with VPN-to-MFN (privileged); Jump back to OS (reducing privilege)
	5. Return from trap	
		6. Trap handler: Unprivileged code trying to return from a trap; Return from trap
7. Resume execution (@PC of instruction); Instruction is retried; Results in TLB hit		

Table B.5: **TLB Miss Flow with Virtualization**

Q5. What is the information gap in VMMs? What problems does it cause?

The VMM often doesn't know too much about what the OS is doing or what the OS wants; this lack of knowledge is called the information gap. This can cause various inefficiencies -- for example, if the VMM is running underneath two OSes, and one of them doesn't have anything to do and just spinning idly, the VMM will still give equal CPU time to both. If the VMM knew that one of the OSes was idle, it would be able to give more CPU time to the OS that was doing useful work.

Eventual Consistency

Q1. Compare and contrast ACID and BASE semantics. In what situations can ACID semantics not be guaranteed, and why not?

ACID is generally not possible in eventually consistent distributed systems. The CAP theorem states that in the presence of a **network partition** where message delivery times are unbounded, we cannot guarantee both consistency and availability. These are where eventually consistent systems will come into play - after a series of writes, a read will eventually return the last write.

Eventually consistent systems are **basically available** (they guarantee availability) and have **soft state** (state is not everywhere constant, without input the state is subject to change). **Eventually consistent** means that given the system **does not receive any more input** it will eventually return the last written value.

ACID database transactions are not possible in cases where a bunch of cohorts must commit a transaction with uniform consensus in the presence of a network protocol.

Multiprocessor Scheduling

Q1. What are some solutions taken by a multiprocessor system to deal with cache coherency?

One way is to use **bus snooping**; each cache pays attention to memory updates by observing the bus that connects them to main memory. When a CPU then sees an update for a data item it holds in cache, it either invalidates or updates its own cache value for that data item. (Cache coherency problem is solved via bus snooping)

Q2. Does bus snooping solve all concurrency issues arising with MP parallelism? If not, explain a potential issue and a solution for it.

For example, if two threads (one on each CPU) or operating on a shared queue, the bus snooping alone won't solve the issue of making sure the data structure is modified correctly and consistently. Just like with single-CPU parallelism, we will need to employ locks around critical sections so that only one CPU at a time writes to sensitive data.

Q3. Discuss some important factors to consider while designing a multiprocessor scheduler. What are the two basic approaches to designing one?

There are two basic approaches to designing a multiprocessor scheduler:

1. Put all jobs that need to be scheduled into a **single queue** (i.e., just adapt our existing single-CPU policies).
    - (+) simple to implement! Just take existing one-CPU policy and adapt it slightly to work on two CPUs
    - (-) lack of scalability
    - (-) cache affinity becomes a problem; we make no attempt to run one thread on the same CPU each time
        - Taking advantage of cache affinity can be highly beneficial since it enables the reuse of the data cached on that CPU's L1/L2 caches, hits of which can make a 20-200x difference compared to a memory reference.
2. Opt for **multiple queues**, e.g., one per CPU
    - Apply load balancing to keep the queues balanced so that none of the CPU's wastes time inactive while others are working.
        - This can be done by migrating jobs around from one queue to another. (**work stealing**)
        - Need to be careful when doing this to avoid creating high overhead of looking for other jobs instead of doing useful work.
        - Also need to consider the effect of the migration on cache affinity.
    - (+) Scales better and handles cache affinity well
    - (-) Has trouble with load imbalance and is more complicated

Q4. Discuss the concept of hyperthreading.

The motivation for hyperthreading is similar to that of cache affinity: it's beneficial to run the same job on the same CPU. With hyperthreading, we give each core two sets of general registers, enabling it to run two independent threads. We ideally want the threads to have similar address spaces (e.g. because they are threads of the same process - the address spaces would be the same in this case) on the same CPU. This can result in huge performance wins since the L1 and L2 caches will be highly reusable by the two threads.

A memory reference is **200x** slower than an L1 cache hit. So while the CPU blocks for memory access, we can perform a micro-context switch and run the other thread on the CPU for a while. This usually provides a **1.2 - 1.8x** improvement in performance.

Q5. What are the difficulties of scaling up a system to a large number of CPUs?

Scaling up to large numbers of CPUs is hard because eventually memory bandwidth becomes a bottleneck for the system (memory is still shared between all of the CPUs, but they do have their own L1/L2 caches). Memory buses cannot provide parallel accesses to more than a couple of CPU cores, so eventually many CPUs will become blocked while waiting for memory since the memory BUS is servicing another CPU. This will reduce parallelism greatly, and basically defeat the concept of multiple CPUs in the first place.

This can be improved with a **Non-Uniform Memory Architecture** (or NUMA for short). NUMA separates CPUs into independent components (nodes) and gives its each node its own (high-speed) local memory. The local memories are interconnected via a (slower) memory network.

Now, each CPU has its own local memory as well so a CPU will not need to wait in line to access local memory. However, this local memory is obviously much smaller than the computer's RAM, so we will need to occasionally access the memory network to get memory that lives in other areas. Overall, we get **nearly linear** scaling in the case where each CPU uses almost entirely its own local memory.

In addition to servicing read/write requests, we must also consider NUMA cache coherency. For example, what if one NUMA's cache stores data that was updated in the local memory of another NUMA node? These are hard problems (and therefore interesting).

Clustering Concepts

Q1. Discuss in the concept of membership within a cluster, and how we can check who the members of a cluster are.

Membership is

Q2. What is “split-brain” and what are two different mechanisms to prevent/detect when this has happened?

Split-brain is when a network failure divides a cluster into multiple sub-clusters which can not communicate with each other. Each sub-cluster continues operating independently and may make incompatible decisions. Two mechanisms to prevent/detect split-brain are quorum and voting devices.

Quorum is when you make a rule saying that there can’t be a cluster with less than (N + 2) / 1 members, where N is the total number of nodes in the client. This means that there cannot be two separate clusters operating independently, because there can’t be two sub-clusters each with (N + 1) / 2 nodes.

Voting devices can be pieces of hardware that must be present for a cluster to function, and are owned by one node. All nodes in the cluster can use that piece of hardware as a voting device to show that they are active and able to communicate with other nodes.

Q3. How can we handle rogue nodes?

By using **fencing.**

Horizontal Scaled Systems

Q1. Compare and contrast horizontally scalable and vertically scalable systems.

A horizontally scalable system is one whose capacity and throughput are increased by adding additional nodes. A vertically scaled system, on the other hand, is scaled by replacing smaller nodes with larger and more powerful ones. It's the equivalent of adding more disks to your data center versus replacing each disk with one that has higher capacity.

A horizontally scaled system is based on protocols developed for client-server distributed storage and distributed computing systems. BUT, unlike their more complex cousins, their evolution has been guided by the principles of maximum independence (loose coupling) and parallelism.

Q2. Discuss the differences between loosely coupled and tightly coupled systems.

Overall, **coupling** refers to the degree of independence between software modules.

Loosely coupled systems are characterized by a parallel group of independent computers serving similar but independent requests with minimal coordination, cooperation, and resource sharing required. Some examples include web servers, Google search farm, and Hadoop.

A tightly coupled system, on the other hand, has a high amount of interdependencies between system components (e.g. computers). They may share the same resources and locks.

Q3. Discuss in great detail the concept of a horizontally scalable system, and how they improve capacity, throughput, and availability.

Scalability and robustness are improved by eliminating the need for synchronization and communication between parallel servers. This is because they don't need to share resources.

If the system is also implemented via **stateless protocols** we can achieve things like load balancing, easy recovery after crashing, and the use of idempotent requests.

Less configuration of additional servers in a horizontally scaled system also makes the job easier for devops people, since we just need to tell the server where to start looking for requests and don't need to deal with giving it resources, locks, etc.

Flexibility and performance are improved by enabling the (non-disruptive) addition or removal of servers at any time. This is why most cloud services must be designed to be

horizontally scalable (so that you can add nodes as your app scales, and decrease nodes as load decreases without any disruption or perceived decrease in performance).

Reliability and availability is drastically improved by the *shared nothing architecture*, which prevents the failure of any single component from affecting multiple systems. To take this a level further than components, a good horizontally scalable system is installed in distinct *availability zones*, giving it the ability to failover to distant copies and servers (this is sometimes called *geographic disaster recovery*).

Q4. Compare and contrast the active/active vs active/standby and their relative performance.

They are both approaches to high availability.
The **active/active** protocol is a design where every available node (server) in the horizontally scaled system is actively servicing requests. When one server fails or is running especially slowly, its load is distributed across the rest of the servers for it to handle. This is what **MapReduce** does. This has high performance, but the performance may be (drastically) degraded if failure occurs often.

The **active/standby** protocol is a design where not all of the available nodes are servicing requests, they're just waiting for a working node to fail. When a working node fails, the standby node can be hot swapped in for the failed one. This means that throughput/capacity may be less since we're not using every node available, but performance will not suffer as much in the case of failure.

Of course, if you're Google or Amazon, you can have both of these.

Q5. How can a highly stateful distributed key value store be implemented in a horizontally scalable fashion?

Highly stateful systems such as distributed databases can be designed to be horizontally scalable using the idea of shards. This idea is that the namespace of requests is divided, and each partition is assigned to some small amount of servers. When we need to scale, instead of assigning more servers to the partitions (which would break the concept of shards in the first place), we would break the partitions into even smaller partitions (therefore breaking the namespace up more) and assign servers to those.

For example, we can first partition all key-value accesses with one server being assigned to service all requests beginning with A - M, and the other with N - Z. When we need to scale we can break it into A - G, G - R, and R - Z.

Q6. What are two methods to perform distributed computations on data, and discuss the mechanisms, advantages, and disadvantages of each?

Bringing the data to the computation, or bringing the computation to the data are two options. In both cases, the client (ie, us) knows the computation we would like to perform. There are two cases: we send the computation to where the data is (somewhere other than our machine), or we bring the data to our local machine to perform the computation.

Bringing the data to the computation involves locking the data structure, copying it into your local memory, updating the pointer to reflect its new location, freeing the old copy, and performing all operations on this new copy. An advantage of this is that we can now run our computation locally (and future ones) whenever we want, once we have the data it is no longer up to other nodes to run our computation at their leisure. A disadvantage is that we have to deal with locking, which is basically always a mess waiting to happen.

Moving the computation to the data involves looking up the node that owns the data/resource we would like to manipulate. We then send a network message to that node, requesting it to perform our operations. We can then repeatedly poll or just wait for an interrupt to know that our request has completed (or errored/timed out). An advantage of this is that we don't have to deal with locking the data or copying it to local memory (which can be prohibitively expensive if you're working with MapReduce style datasets), but a disadvantage is that this computation may not happen quickly if the node that owns this resource is quite busy.

Note: Both of these are powerful general mechanisms which apply in several cases where there are nodes working in the same system, such as in the NUMA architecture or in distributed systems where servers are located across the country.

Words to Pull Out of Your Ass

- Hybrid approach
- Cache
- Buffer
- Faults/Catch
- Signals/Interrupts
- Every 30 seconds
- Falacy
- Sequential I/O > Random I/O
- Tradeoff
- Heartbeat
- Amortize

- scalability/reliability/availability/flexibility
- Consistency
- Atomicity
- distributed