UPE Tutoring:

CS 111 Final Review

Sign-in: http://bit.ly/2UwhSDH

Slides link available upon sign-in

Concurrency

- Goal of synchronization: protect access to resources so that data always appears to be in consistent state
- To do this we need **critical sections**, code segments where only one thread may be running at a time
- Critical sections ensure two forms of **atomicity**
  - Before/after atomicity - operations do not step on one another, no mingling of intermediate steps
  - All/nothing atomicity - from the perspective of other threads, an operation (no matter how complex) appears to be either done or not done, <u>never half finished</u>

Concurrency: Locks

- Critical sections can be enforced using **locks**
  - i.e. pthread_mutex_lock() in C
- Entrance to critical section only after thread acquires lock
- Other threads cannot enter until lock released

Concurrency: Lock Implementation

- 4 evaluation measures of locks
  - Correctness
  - Progress - potential for deadlock?
  - Fairness - can some thread never acquire lock?
  - Performance - is CPU overhead of using the lock minimized?

Concurrency: Lock Implementation

- Disable interrupts
  - Usually used in interrupt handlers themselves to enforce **re-entrancy**
  - Correctness: works unless on multi-core architecture, impossible in user-mode code
  - Progress: can deadlock if some resource takes forever (lock never released)
  - Fairness: long disables lead to potential monopolization of CPU, short ones OK
  - Performance: overhead for the disabling instruction itself is small, but long lock-could degrade system-wide performance

Concurrency: Lock Implementation

- A simple variable
  - I.e. 1 for locked, 0 for unlocked
- But how to ensure setting of variable itself is atomic?

Lock Implementation: Hardware instructions

- Test and set: one way of performing atomic variable updates
- <u>Atomically</u> do 3 steps:
  - Record old value
  - Set new value
  - Return old value
- We call testAndSet(1) ← try to lock the lock to 1
- If it returns 0 → old value was 0 → we have the lock and it's set to 1
- If it returns 1 → old value was 1 → value unchanged: we don't have the lock

Lock Implementation: Hardware instructions

- Compare and swap
- <u>Atomically</u>:
  - If value is what's expected, set to new value
  - Return old value
- We call compareAndSwap(0, 1) ← try to lock the lock to 1, expecting it to be 0
- If it returns 0 → old value was 0 → we have the lock and it's set to 1
- If it returns 1 → old value was 1 → value unchanged: we don't have the lock

Lock Implementation

- The aforementioned hardware tools allow us to implement different kinds of locks
- Spinlock
  - while (locked) { try lock }
  - Simple, desirable if contention low, so overhead of sleeping/waking from blocking locks is avoided
  - Waste of CPU clock cycles “spinning” waiting for lock to be freed
- Spin & yield
  - Spin only few times, then yield (context switch)
  - Potential context switch overhead

Lock Implementation

- Condition variable
  - When a thread must wait for an event to occur before proceeding
  - Instead of spinning, use condition variable
  - **Blocks** (sleeps) until variable changes, resulting in a signal being sent to OS

Deadlocks

- When two things are waiting for each other to finished before they start
- Caused by hold-and-wait on common resources:
  - Locks/mutexes/semaphores
  - Memory
- Bad dependency management
- 4 Conditions:
  - Mutual Exclusion
  - Incremental Allocation
  - No Pre-emption
  - Circular waiting

Deadlock

```
process A(){                                        process B(){
    acquireX();   ← Step1                  Step2 →      acquireY();
    acquireY():   ← Step3                               acquireX():

    //do stuff                                          //do stuff

    releaseY();                                         releaseX();
    releaseX();                                         releaseY();
}                                                   }
```

Problem

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

Problem

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

Answer: *Without hold-and-wait, the deadlock couldn't occur since the lock would be released eventually (no holding!) and other processes would not be stuck forever waiting for the lock. One method is to release other held locks if trying to acquire the next lock fails. That way, other processes won't have to wait for the first lock.*

I/O

- Devices and Drivers
  - Operate as interrupt driven
  - Connected to a bus
- Disk I/O
  - Reading and writing data is a big time consumer
  - Direct Memory Access: Allows busses and devices to directly move memory around without having to go through the cpu, though this takes up bus time
  - If device processes want reads/writes, they make a request and block

I/O

- Disk reads can be slow
  - Disk seek and rotation overheads are unwanted
  - Good things:
    - Caching and buffering are friends :)
    - Larger transfers
    - Queued requests
    - Combining nearby requests
    - Write-back (sometimes)
  - Double buffering

Problem

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

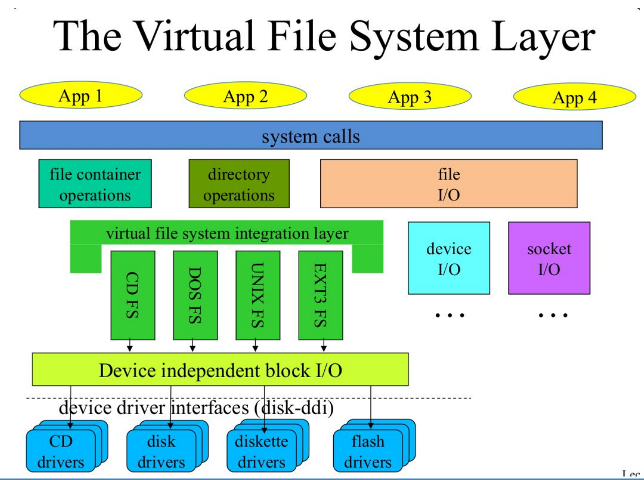
Problem

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

Answer: *There are many right answers. One solution - caches and buffering. Helps reduce read/write disk operations. Makes OS more complicated since it has to manage when to actually interact with the disk.*

File System Abstractions

- File System API: interface between **user and kernel** (syscall)
  - File Container Operations: manipulate file as objects (ignore contents of file)
    - `stat(2), chmod(2), ...`
  - Directory Operations: organization of file system hierarchy
    - `mkdir(2), getcwd(2), ...`
  - File I/O Operations: contents of the file
    - `open(2), close(2), read(2), write(2), lseek(2), ...`

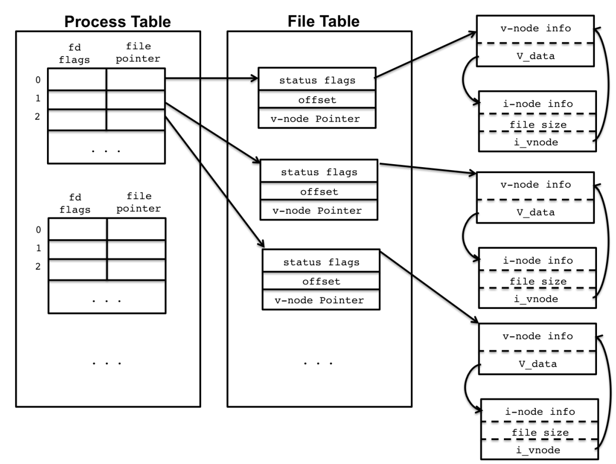


File System Layer

- **Virtual File System Layer**: uniform interface for different **file system implementations** (inside kernel)
  - E.g. DOS, ext4, NFS, procfs, tmpfs, ...
  - Each file system implemented by a kernel module
  - All file system modules implement the same basic operations
- **Block I/O Layer**: uniform interface for **Block I/O** (interact with hardware)
  - Make all disks look the same
    - Standard operations on block device
    - Map logical block numbers to device addresses
    - Encapsulate all the particulars of device support
  - Advantage
    - Unified buffer cache for disk data
    - Automatic buffer management: allocation, deallocation, automatic write-back

File System Layer Design

- Why Virtual File System
  - Multiple file systems
    - Different storage devices, different services, different purposes
    - On seasnet: `$ mount  #` shows currently mounted file systems
- Page Cache: In memory cache representing structures on disk
  - Caches file control structures in memory
  - File access exhibits reference locality
  - Common cache reduce number of disk accesses
  - Shared among different processes
    - Some structures are private to processes: file offsets (cursor), ...
- Fun with linux disk cache: https://www.linuxatemyram.com/play.html



DOS FAT File System

- Divide disk into "clusters"
- FAT (File Allocation Table): contains the number of next cluster in file
  - Each entry corresponds to a cluster
- Characteristics
  - To access (seek) **n**th "cluster" of a file: O(**n**)
  - Size of FAT entries limits the disk size
  - No support for sparse files
    - Requires all "clusters" to be present
    - ```
      $ dd if=/dev/zero of=sparsefile bs=1 count=0 seek=128G
      $ ls -hl sparsefile
      $ du -h sparsefile
      ```
      ```

DOS FAT File System

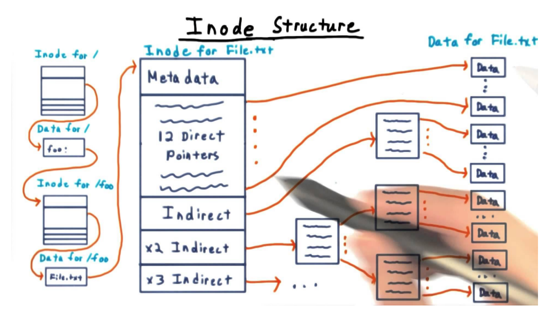
block 0_{512}	boot block	
block 1_{512}	BIOS parameter block (BPB)	Cluster size and FAT length are specified in the BPB
block 2_{512}	File Allocation Table (FAT)	Data clusters begin immediately after the end of the FAT
	cluster #1 (root directory)	Root directory begins in the first data cluster
	cluster #2 …	

Unix File System

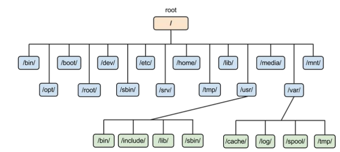
Block 0	Boot block	
Block 1	Super block	Block size and number of I-nodes are specified in super block
Block 2	I-nodes	I-node #1 (traditionally) describes the root directory
	Available blocks	Data blocks begin immediately after the end of the I-nodes.

Unix File System

- Inode: **fixed size metadata** that stores attributes and disk block locations of the object
  - `$ ls -il file  # shows inode number of file`
  - Block pointers: direct pointers, indirect pointer
    - First ~10 blocks can be found without extra I/O
    - 1-3 extra I/O per thousand pages
    - Largest possible file
      - Each direct pointer: 1 block
      - Each indirect pointer: 1024 blocks (assuming each block can hold 1024 block pointers)
      - Each doubly/triply indirect pointer, …
    - Support sparse file: with empty block pointers (similar to NULL pointer)



File System Hierarchy



- Directory: s special kind of file
  - Contains multiple directory entries: name, pointer to the inode
  - One entry is special file `..` : parent directory
- Navigate File System
  - Absolute path
    - `$ namei /etc/systemd/system/default.target`
  - Relative path
    - `$ namei ../../`
- Why can't we open a file by inode?
  - Bypass Unix file permission check
    - `654321  /dir   ----------` (directory not readable and not accessible)
    - `123456  /dir/f -rwx------`

Hard Link vs. Symbolic Link

- Hard link: file exists under multiple names
  - Appear exactly the same as a normal file
  - In **inode**, there is a field storing the link count
    - When a reference to this file is removed, link count reduce by 1
    - When link count reduce to 0, the space underlying the file is freed
- Symbolic link: an **alias** to another file (just contains a pathname)
  - A special kind of file: `$ stat symlink  # shows symlink is a symbolic link`
  - Not a reference to the inode
    - Does not prevent deletion (does not affect *inode link count* of linked file)

File System Reliability

- RAIDs (Redundant Array of Inexpensive Disks)
  - Level 0 (Stripping)
    - Spread consecutive blocks across different disks for high performance
  - Level 1 (Mirroring)
    - Keep more than 1 copy of each block on different disks
  - Level 4 (Saving Space with Parity)
    - Keep a disk that stores parity of blocks on different disks
    - Good if only 1 disk fails - Fails if multiple disks fail
  - Level 5 (Rotating Parity)
    - Similar to Level 4, but includes parity block for each row on different disks

File System Journaling

- Keeps track of changes not yet committed to the File System
  - If system crashes, look at log to see the state of the disk vs the latest committed transaction for recovery
- Data Journaling:
  - Journal Write: Write the contents of journal to log and wait for completion
  - Journal Commit: Write transaction commit block to the log and wait for completion
  - Write contents of update to final location on disk
- Metadata Journaling:
  - Store metadata in log about writes
  - First write data blocks, then metadata blocks
  - After completion, write the commit block
- Log can become really long, and thus expensive to store: Circular Log

Virtual Memory

What?

- Abstract way of referring to physical memory location using virtual memory
- Generally a hardware method (software doesn't know anything about what the real memory location is)

Why?

- Memory Abstraction
- Safety
- Running more than just one process

How?

- Address Translation (how hardware converts from virtual to physical memory
- Need to juggle translation for multiple programs
- MMU(Memory Management Unit)

Virtual Memory

The basic way: Base and Bounds

- Hardware keeps track of the Base and Bounds for each process
- This helps translate each processes' virtual memory to the actual physical memory
- Process thinks it is at address 0x00000000
  - In reality it is 0x00000000 + Base
  - Max value is 0x00000000 + Bounds
- The physical base location is picked from the *free list*
- There is a Memory Management Unit that does all of this per address

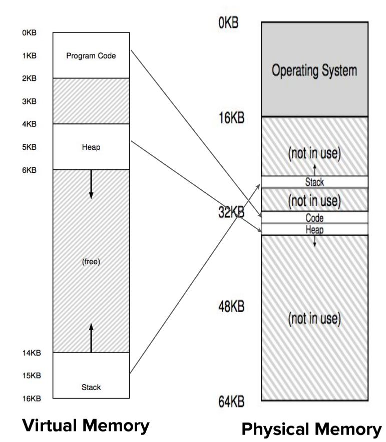
Segmentation

- Problem of fragmentation
  - Sometimes segments get placed in weird spots and even though you have enough memory, a new process might not fit in the "gaps"
- Lets use segments!
- Segmentation breaks up process memory into regularized chunks
- Essentially base/bounds for every segment of memory
- Since stack and heap grow in opposite directions, you also need another bit to specify direction of each segment

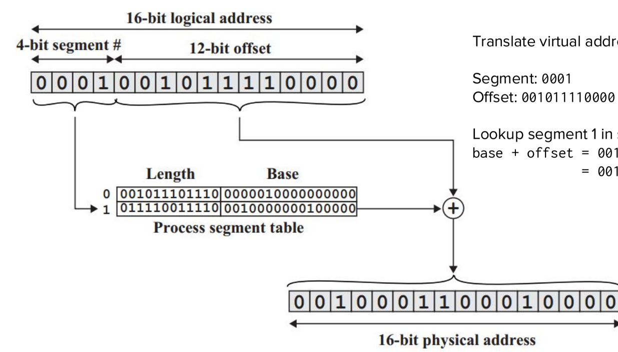
Segmentation

Some Process

Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



Address Translation with Segmentation

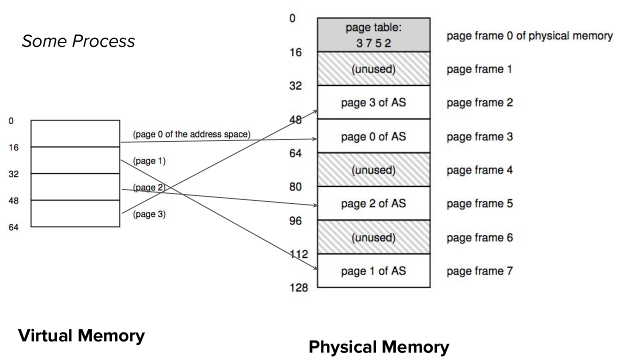


Translate virtual address `0001001011110000` to physical address

Segment: `0001`
Offset: `001011110000`

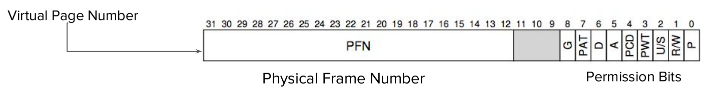
Lookup segment 1 in segment table, physical address is
`base + offset = 0010000000100000 + 001011110000`
`= 0010001100010000`

Paging

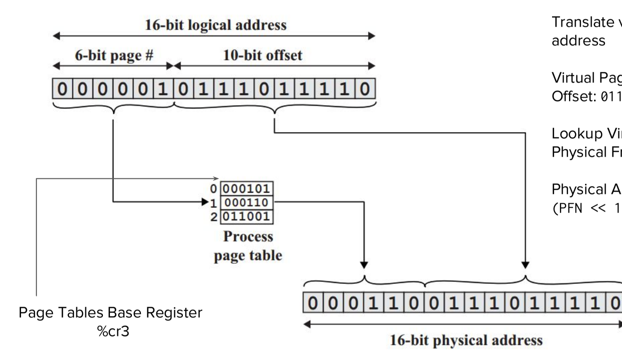


Paging

- Divide physical memory and virtual address space into units of single **fixed size**
  - On seasnet, page size is 4K (see it for yourself! `getconf PAGE_SIZE`)
  - Typically called page frame
- Treat the virtual address space in the same way
- Store data in each ***page*** in virtual address space to ***page frame*** in physical address
- How to translate from page to page frame
  - Page Table: per process data structure that stores address translations of virtual pages to physical page frames
  - Page Table Entry



Address Translation with Paging



Translate virtual address `000001 0111011110` to physical address

Virtual Page Number: `000001`
Offset: `0111011110`

Lookup Virtual Page Number (VPN) in page table, get Physical Frame Number (PFN) `000110`

Physical Address
`(PFN << 10) | offset = 000110 0111011110`

Paging v.s. Segmentation

- Segmentation
  - Specify arbitrarily sized ranges of the address space
  - Address space ranges can be used for a particular purpose, such as code segment or stack
- Paging
  - Divide allocated memory space into smaller pieces of the same size
  - Allow the virtual memory management system to load, relocate, and otherwise manage the space more flexibly
- Segmentation and paging can exist in the same system
  - Segmentation specifies address that are valid/legal
  - Paging allows OS to map small sections of virtual address ranges in physical memory

Translation Lookaside Buffer (TLB)

- TLB caches recently used pages
- TLB miss (accesses to virtual addresses not listed in the TLB) trigger a page table lookup
  - The cache entry is then added to TLB for future access
  - Huge performance penalty
- TLB gets flushed whenever a context switch happens
  - Why? Different processes have different address space

Page Fault

- When a process tries to access a page of virtual address space that is **not mapped** onto a page frame of physical memory
  - E.g. access an address that is invalid (valid bit in Page Table Entry)
- Disambiguate with *segmentation fault*: when a process tries to access an **invalid or illegal** memory address
  - E.g. writing to an address that is read only (R/W bit in Page Table Entry)
- What could happen to a process experiencing page fault?
  - Nothing: e.g. on demand paging, transparent to process
  - Receive signal `SIGSEGV`: e.g. access unmapped memory

On Demand Paging

- Why on demand paging?
  - Kernel doesn't have to load all pages into physical memory
  - Load pages when a process actually uses it
  - Improve memory locality
- How to implement it?
  - Mark the virtual pages not present in physical memory (**present bit** in Page Table Entry)

On Demand Paging, cont'd

- How to load page on demand?
  - Hardware finds out that the page is not present in physical memory, generates interrupt
  - Traps to kernel, and control is transferred to **page fault handler**
  - Checks permission and determines which pages are needed
  - Allocates physical page frames
    - Load from disk (file or swap)
  - Update Page Table Entries with allocated physical page frames and setting appropriate permission bits
  - Resume execution

Process Memory Layout, revisited

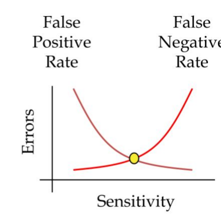
```
$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:07 131133                     /usr/bin/cat
0060b000-0060c000 r--p 0000b000 08:07 131133                     /usr/bin/cat
0060c000-0060d000 rw-p 0000c000 08:07 131133                     /usr/bin/cat
01577000-01598000 rw-p 00000000 00:00 0                          [heap]
7f62049a6000-7f620aecf000 r--p 00000000 08:07 661772             /usr/lib/locale/locale-archive
7f620aecf000-7f620b087000 r-xp 00000000 08:07 261432             /usr/lib64/libc-2.17.so
7f620b087000-7f620b287000 ---p 001b8000 08:07 261432             /usr/lib64/libc-2.17.so
7f620b287000-7f620b28b000 r--p 001b8000 08:07 261432             /usr/lib64/libc-2.17.so
7f620b28b000-7f620b28d000 rw-p 001bc000 08:07 261432             /usr/lib64/libc-2.17.so
7f620b28d000-7f620b292000 rw-p 00000000 00:00 0
7f620b292000-7f620b2b3000 r-xp 00000000 08:07 261434             /usr/lib64/ld-2.17.so
7f620b487000-7f620b48a000 rw-p 00000000 00:00 0
7f620b4b2000-7f620b4b3000 rw-p 00000000 00:00 0
7f620b4b3000-7f620b4b4000 r--p 00021000 08:07 261434             /usr/lib64/ld-2.17.so
7f620b4b4000-7f620b4b5000 rw-p 00022000 08:07 261434             /usr/lib64/ld-2.17.so
7f620b4b5000-7f620b4b6000 rw-p 00000000 00:00 0
7ffda2887000-7ffda28a8000 rw-p 00000000 00:00 0                  [stack]
7ffda29ad000-7ffda29af000 r-xp 00000000 00:00 0                  [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
```


Security - Basics

- Three Main Topics
  - Authentication - is this who he claims to be?
  - Authorisation - do we trust this person / allow him to do this action?
  - Cryptography - how can we maintain confidentiality and integrity
- Three Goals
  - Confidentiality - other people should not be able to see this
  - Integrity - you get the message I send you
  - Availability - This resource is available as specified by its interface, no one blocks it.
- The OS
  - If the OS is compromised/you can't trust it... you're "got"

Security - Authentication

- What you know
  - Passwords and some Challenge Response
  - Salt -> Hash
    - Why? Dictionary Attacks
- What you have
  - Phones, dongles Challenge Response
- Who you are
  - Biometric Auth.
    - False positive, False negative
    - Tweaking? Crossover Error Rate Point.
- Multi-Factor, use advantages of multiple methodologies.



Security - Authorisation

- Subject, Object, Access.
- Reference monitor - handles the algorithms to figure out who can access what when.
- Access Control List
  - Each obj. Has a list of subjects who can access.
  - Stored by files (ex: linux RWX bits for UGO)
- Capabilities
  - Each subj. Has a list of objects he can access. (ex: Android Permission Label)
  - Usually stored in OS (think Process Control Block)
- Each has Pros and Cons, and make up for where the other lacks.
- Other: Role Based Action Control

Security - Cryptography

- Altering bits in a controlled way which improves the security of a system
- Symmetric - use a key both parties have to decrypt and encrypt
  - If the key gets out… you're "got". (ex. Heartbleed controversy)
  - DES vs AES
- Asymmetric - Public and private keys (ex. RSA, Elliptic Curve)
  - I can sign with your public and you are the only one who can decrypt with private
  - I can sign with my private and you can decrypt knowing the message is from me
    - Windows Update Example
- Hashing Functions
  - Passwords, as before
  - Message Integrity
  - "Proof of work"

Remote Data: Goals and Benchmarks

Basic Goal: Client accesses data held in a remote location in the same manner that they access local data.

Benchmarks

- Transparency: Remote data access being indistinguishable from local data access
- Performance: Speed and scalability
- Cost: Capital cost and operational cost
- Capacity: Space available to the clients
- Availability: Clients can query for files at any time.

Client/Server

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

Remote file transfer

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

Remote Disk Access

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

Remote File Access

- The remote server is seen as a local file system
- Pros
  - Transparency
  - Functional encapsulation
  - Supports multi-client file sharing
  - Good performance
- Cons
  - Requires implementation in the OS
  - Introduces complexities to client/server
- Compared to Distributed File Systems
  - Remote file access is more simple
  - Distributed file system has higher performance and scalability

Security for Remote File Systems

- Issues
  - Privacy and integrity of data on the network
    - Solution: Encrypt the data sent over the network
  - Authentication of remote users
    - Solutions:
      - Anonymous Access: No authentication required for reads
      - P2P: Leave the authentication up to the clients
      - Server Authentication: Server authenticates clients
      - Domain Authentication: Trusted third party handles authentication

Reliability and Availability

- Reliability
  - Implemented via redundancy (Mirroring, Parity, Erasure Coding)
  - Important to automatically recover after failure
- Availability and Fail-Over
  - Fail-Over: Being able to detect failed servers and transferring requests to a working server
  - Failure Detection can be done either at client end or server end
  - Network protocols that do not save state make fail-over easy

Remote File System Performance

- Disk Bandwidth Implications
  - Single server, multiple clients = limited throughput
  - Striping files across multiple servers increases scalable output
- Network delay/packet-loss decreases file system throughput
- Cost of Reads/Writes
  - Use caching to improve reads and writes at the cost of consistency
- Cost of Mirroring
  - Mirroring can be done by the server or by the client
- Direct Data Path: Client directly connected to the storage servers, improves throughput, latency, and scalability

Improving remote file systems

- Improving availability
  - Improving failure detection, fail-over speed, and recovery speed
- Improving speed
  - Minimize messages sent over the network (expensive)
- Improving scalability
  - Avoid single control points
  - Separate data plane and control plane (Direct Data Path)
  - Avoid consensus-based protocols

Good luck!

Sign-in	http://bit.ly/2UwhSDH
Slides	http://bit.ly/2CdJCWM
Midterm Slides	http://bit.ly/2u09Kjc

Questions? Need more help?

- Come up and ask us! We'll try our best.
- UPE offers daily computer science tutoring:
  - Location: ACM/UPE Clubhouse (Boelter 2763)
  - Schedule: https://upe.seas.ucla.edu/tutoring/
- You can also post on the Facebook event page.