Operating Systems Principles
UCLA-CS111-W18

Quentin Truong
Taught by Professor Reiher

Winter 2018

Contents

1 L3: Arpaci-Dusseau Chapter 5: Interlude: Process API

1.1 The fork() System Call

- Crux: How to create and control processes
- fork()
  - Creates new process; returns child's PID to parent; returns 0 to child;
  - Each has own PC, registers, address space
- Nondeterministic Behavior
  - Scheduler will decide which process to run
  - May lead to problems in multi-threaded programs

1.2 The wait() System Call

- wait()
  - Parent calls wait() to wait for child to finish execution

1.3 The exec() System Call

- exec()
  - Loads code, overwrites code segment, and reinitializes memory space
  - Takes exceutable name and arguments
  - Does not create a new process; transform current process

1.4 Why? Motivating The API

- Separation
  - Separating fork() and exec() allows code to alter the environment of the about-to-run program
- Example
  - Shell forks a process, execs the program, and waits until finished
  - The separation allows for things such as output to be redirected (closes stdout and opens file)

1.5 Other Parts Of The API

- kill()
  - System call sends signal to process to sleep, die, etc

2 L3: Arpaci-Dusseau Chapter 6: Mechanism: Limited Direct Execution

2.1 Basic Technique: Limited Direct Execution

- Crux: How to efficiently virtualize CPU with control
- Limited Direct Execution
  - OS will create entry for process list, allocate memory for program, load program into memory, setup stack with argc/v, clear registers, execute call to main()
  - Program will run main(), execute return
  - OS will free memory, remove from process list
- LDE good bc fast, but
  - Problem of keeping control
  - Problem of time sharing still

2.2 Problem 1: Restricted Operations

- User mode vs. Kernel mode
  - Restricted mode which needs to ask kernel to perform system calls
  - Calls like open() are actually procedure calls with trap to enter kernel and raise privilege
  - Return-from-trap is used to enter user mode from kernel and drop privilege
  - Push counters, flags, registers onto per-process kernel stack when trapping
- Trap table is used to control what code is executed when trapping
  - Trap handler used by hardware to cause interrupts
  - Telling hardware where trap table is is privileged
  - Trap handler actually uses system-call number, rather than specifying an address (another layer of protection)
- Two phases of LDE
  - At boot, kernel initializes trap table and remembers where it is

OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember addresses of...
	syscall handler
	timer handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

2.3 Problem 2: Switching Between Processes

- How can OS regain control?
  - Because process is running, so OS is not running
- Cooperative Approach
  - System calls include explicit yield system call, transfering control back to OS
- Noncooperative Approach
  - Reboot, Timer Interrupt
- Saving and Restoring Context
  - Scheduler will choose when to switch processes

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A
		...
	timer interrupt	
	save regs(A) to k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) to proc-struct(A)		
restore regs(B) from proc-struct(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) from k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

2.4 Worried About Concurrency?

- Interrupt during interrupt?
  - Many complex things to do
  - Could disable interrupts (but this might lose interrupts), or locking schemes, etc

2.5 Summary

- Reboot
  - Good technique because restores system to well-tested state
  - OS will 'baby-proof' by only allowing processes to run in restricted mode and with interrupt handlers

3 L3: Linking and Libraries: Object Modules, Linkage Editing, Libraries

3.1 Introduction

- Process as fundamental; as executing instance of program
  - Program as one or more files (these are not the executables though)
  - Source must be translated

3.2 The Software Generation Tool Chain

- Source module
  - Editable text in some language like C
- Relocatable object module
  - Sets of compiled instructions; incomplete programs
- Library
  - Collection of object modules
- Load module
  - Complete programs ready to be loaded into memory
- Compiler
  - Parse source modules; usually generates assembly, may generate pseudo-machine
- Assembler
  - Object module with mostly machine code
  - Memory addresses of functions, variables may not be filled in
- Linkage Editor
  - Find all required object modules and resolve all references
- Program Loader
  - Examines load module, creates virtual space, reads instructions, initializes data values
  - Find and map additional shared libraries

3.3 Object Modules

- Code in multiple files
  - Because more understandable if splitting functionality
  - Many functions are reused, so use external libraries
- Relocatable object modules are program fragments
  - Incomplete because make references to code in other modules
  - Even the references to other code are only relative
- ELF format
  - Header section with types, sizes, and location of other sections
  - Code and data section to be loaded contiguously
  - Symbol table of external symbols
  - Relocation entries describing location of field, width/type of field, symbol table entry

3.4 Libraries

- Reusable, standard functions in libraries
  - Libraries not always orthogonal and independent
- Build program by combining object modules and resolving external references

3.5 Linkage Editing

- Resolution
  - Search libraries to find object modules to resolve external references
- Loading
  - Lay text and data in single virtual address space
- Relocation
  - Ensure references correctly reflect chosen address

3.6 Load Modules

- Load module requires no relocation and is complete
- When loading new module
  - Determine required text and data sizes and locations, allocate segments, read contents, create a stack segment with pointer
- Load module has symbol table to help determine where exceptions occurred

3.7 Static vs. Shared Libraries

- Static Linking
  - Many copies, so inefficient; also, permenant copy, so don't receive updates
- Shared Libraries
  - Implementations vary, but one way
    - Reserve address for libraries, linkage edit, map with redirection table, etc, more mapping
  - Efficient, but doesn't work for static data because one copy
  - But can be slow to load many libraries, and must know library name at loadtime

3.8 Dynamically Loaded Libraries

- DLL loaded once needed
  - Choose and load library, binds, use library, unload
  - Resource efficient because can unload
- Implicitly Loaded Dynamically Loadable Libraries
  - Another implementation of DLL with different pros/cons

4 L3: Linkage Conventions: Stack Frames and Linkage Conventions

4.1 Introduction

- What is the state of computation and how can it be saved?
- What is the mechanism of requesting and receiving services?

4.2 The Stack Model of Programming Languages

- Procedure-local variables
  - Stored on a LIFO stack
  - New call frames pushed onto stack when procedure called; old frames popped when procedure reutrns
  - Long-lived resources on heap, not stack

4.3 Subroutine Linkage Conventions

- X86 Subroutine Linkage
  - Pass parameters to be called by routine
  - Save return address and transfer control to entry
  - Save content of non-volatile registers
  - Allocate space for local variables
- X86 Return Process
  - Return value to where routine expects it
  - Pop local storage
  - Restore registers
  - Subroutine transfer control to return address
- Responsibilities split between caller and callee
- Saving and restoring state of procedure is mostly a matter of stack frame and registers

4.4 Traps and Interrupts

- Procedure call vs Trap/Interrupt
  - Procedure requested by running software and expects result; linkage conventions under software control
  - After trap/interrupt, should restore state
- How
  - Number associated with every interrupt/exception, maps to PS/PC
  - Push new program counter and program status (from interrupt/trap vector table) onto CPU stack
  - Resume execution at new PC
  - First level handler
    - Save general registers on stack
    - Choose second level handler based on info from interrupt/trap
  - Second level handler (procedure call)
    - Deal with interrupt/exception
    - Return to first level handler
      - Restore saved registers and return-from-interrupt/trap
  - CPU realoads PC/PS and resumes execution
- Stacking/unstacking interrupt/trap is 100x+ slower than procedure call

5 L4: Arpaci-Dusseau Chapter 7: Scheduling: Introduction

5.1 Workload Assumptions

- Workload as the processes running in the system
- Fully-operational scheduling discipline
  - Assume each job runs for same amount of time, arrives at same time, once started will run to completion, only uses CPU, run-time length is known

5.2 Scheduling Metrics

- Scheduling metric is something we can measure is useful for scheduling
  - $Turnaround_{time} : Time_{completion} - Time_{arrival}$
- Performance and Fairness often at odds with each other
  - Fairness measured by Jain's Fairness Index

5.3 First In, First Out (FIFO)

- Properties of FIFO
  - Simple and easy to implement while working well based on assumptions
- Convoy Effect
  - FIFO fails if few high-resource consumers are ahead of low-resource consumers

5.4 Shortest Job First (SJF)

- SJF is optimal given the assumptions
  - But fails if relaxes arrival-time assumption
  - A long process may start, then a short process comes in

5.5 Shortest Time-to-Completion First (STCF)

- Preemptive schedulers will context switch to run another process
  - Non-preemptive schedulers run jobs to completion before considering another
  - SJF is nonpreemptive
- Shortest time-to-completion (STCF) also known as Preemptive shortest job first (PSJF)
  - Anytime a new job arrives, determine which job has shortest time remaining, and runs that one

5.6 A New Metric: Response Time

- $T_{response} : T_{firstrun} - T_{arrival}$
- STCF is especially bad for optimizing response time

5.7 Round Robin

- RR (time-slicing) runs job for a time slice (scheduling quantum) before switching to next
  - Length of time slice is essential; if long, then long $T_{response}$; if short, context switching dominates
  - Must choose a length of time which will amortize the cost well
  - Also must consider cost of flushing CPU caches, TLBs, branch predictors, chip hardware
- Performs extremely poorly wrt turnaround time
  - Most fair policies (evenly distribute) are like this

5.8 Incorporating I/O

- Overlap leads to higher utilization and better performance
  - Use for IO, messages, etc
- Overlap CPU when one process requires IO
  - While IO for process A, run process B on CPU (because A is blocked)

5.9 No More Oracle/Summary

- Assumption of known run-time length is highly invalid
- Shortest job remaining optimizes turnaround time
- Alternating between jobs optimizes response time
- Looking ahead
  - Multi-level feedback: Using past events to predict future

6 L4: Arpaci-Dusseau Chapter 8: Scheduling: The Multi-Level Feedback Queue

6.1 MLFQ: Basic Rules

- MFLQ has a number of distinct queues with different priority levels
- If priority(A) $<$ priority(B), A runs
- If priority(A) == priority(B), A and B run in RR
- Vary priority based on observed behavior

6.2 Attempt 1: How To Change Priority

- When job enters, has highest priority
- If job uses entire time slice, priority is reduced
- If job gives up CPU early, priority remains the same
- Assume jobs are short so that it will either complete or move down in priority
- Starvation
  - If there are too many interactive (IO) jobs, then longer processes with low priority will never run
- Gaming the scheduler
  - Could write program to use less than entire timeslice, to always keep highest priority
- Changing Behavior
  - Program may become interactive after computations, so needs higher priority

6.3 Attempt 2: The Priority Boost

- Boost all processes to top priority after a certain time length
- Difficult to know correct value for these voo-doo constant parameters (refer to Ousterhouts Law)

6.4 Attempt 3: Better Accounting

- Account CPU time (Anti-gaming method)
  - Once job uses up time allotment on given level, priority is reduced

6.5 Tuning MLFQ And Other Issues

- Difficult to find correct parameters
  - High-priority queue contains interactive processes and run for short timeslices (20ms)
  - Low-priority queue contains long-running processes and so run for longer timeslices (up to a few hundred ms)
  - Many queues, like 60

    - Priorities boosted every second or so
  - Other schedulers use mathematical formulas to calculate priority (decay-usage)
  - Even may offer advice to scheduler using Linux's nice program

6.6 MLFQ: Summary

  - Multiple levels of queues with feedback to determine priority
  - Rules
    - If priority(A) > priority(B), A runs
    - If priority(A) = priority(B), A and B run in RR
    - When a job enters the system, has highest priority
    - When a job uses entire time allotment at a given level, its priority is reduced
    - After some time period S, move all the jobs in the system to the topmost queue

7 L4: Real Time Scheduling

7.1 What are Real-Time Systems

  - Priority scheduling is best effort
    - Sometimes need more than just best effort (space shuttle reentry, data, assembly line, media players)
  - Traditonal vs Real-time systems
    - Turn-around time, fairness, response time for traditional
    - Timeliness may be ms/day of accumulated tardiness
    - Predictability is deviation in delivered timeliness
    - Feasibility is whether possible to meet requirements
    - Hard real-time is a requirement to run specifiy tasks at specified intervals
    - Soft real-time requires good response time, at the cost of degraded performance or recoverable failure
  - Real-time systems
    - May know length of jobs/priorities, and starvation of certain jobs may be acceptable

7.2 Real-Time Scheduling Algorithms

  - Static scheduling
    - May be possible to define fixed schedule if know all tasks to run and expected completion time
  - Dynamic Scheduling for changing workloads
    - Questions of how to choose next task and how to deal with overload
  - If high enough frequency of work, may just work for sufficiently-light loaded systems

7.3 Real-Time and Linux

  - Linux was not designed as embedded or real-time system
    - Supports a real-time scheduler sched_setscheduler, but still does not have same level of response-times
  - Windows believes in general throughput not deadlines, and is bad for critical real-time operations

8 L5: Arpaci-Dusseau Chapter 12: A Dialogue on Memory Virtualization

8.1 Overview

- Every address generated by a user program is a virtual address
  - Large contiguous address space is easier to work with than small crowded space
  - Isolation and protetion are also important in preventing processes each other's memory

9 L5: Arpaci-Dusseau Chapter 13: The Abstraction: Address Spaces

9.1 Early Systems

- OS as set of routines (a library)
- Program in physical memory used rest of space

9.2 Multiprogramming and Time Sharing

- Multiprogramming
  - Multiple processes ready to run at a given time with OS switching between them
  - Increases utilization of CPU; increased efficiency of CPU is very relevant bc so expensive
- Timesharing and interactivity
  - Long program-debug cycles bad for programmers
  - Giving all programs full access to memory is not safe

9.3 The Address Space

- Address space is easy to use abstraction of physical memory
  - Contains code, stack, heap
  - Every program thinks it had very large address space, even though it doesn't

9.4 Goals

- Transparency
  - Cannot tell that memory is virtual
- Efficiency
  - OS should make virtualization efficient wrt time and space, relying on hardware for this
- Protection
  - Isolate process memory from each other

10 L5: Arpaci-Dusseau Chapter 14: Interlude: Memory API

10.1 Types of Memory

- Stack
  - Automatic memory is managed implicitly by compiler
- Heap
  - Long lived memory where allocations and deallocations handled by programmer

10.2 The malloc()/free() Call

- double *d = (double *) malloc(sizeof(double));
- free(d); // prevents memory leaks

10.3 Common Errors

- Modern languages have automatic memory-management or a garbage collector because people don't free
- Seg fault if you forget to allocate
- Buffer overflow if not enough allocated space
- Dangling pointer if you free memory before finished using it
- Double freeing memory is undefined
- Incorrect use of free (passing it things other than pointer from malloc) is dangerous
- Use Valground and Purify to find memory leaks

10.4 Underlying OS Support

- Break is the location at the end of the heap
  - System call brk is used to increase/decrease size of heap

11 L5: Arpaci-Dusseau Chapter 17: Free-Space Management

11.1 Assumptions

- Free list manages the heap; contains references to all the free chunks in the region
- External fragmentation
  - Have enough space, but not contiguous, so can't malloc
- Internal fragmentation
  - Gives memory larger than requested, which remains unused

11.2 Low-level Mechanisms

- Splitting and Coalescing
  - Split free chunk in two, returning first to the caller
  - Coalesces adjacent free memory together, forming a single larger free chunk
- Header of allocated memory
  - Contains size of region and magic number to speed up deallocation
- Embedding free list
  - Build free list inside the free space itslf
  - Nodes with size and next pointer
- Growing heap
  - Just give up and return NULL
  - Or call sbrk system call to OS to grow heap

11.3 Basic Strategies

- Best fit
  - Return smallest chunk that is equal or larger than the requested size
  - Requires linear search
- Worst fit
  - Find largest chunk, split it, return requested size
  - Requires linear search
- First fit
  - Returns first block big enough
  - Faster because no exhaustive search
- Next fit
  - Returns first block big enough starting from previous location
  - Spreads searches through free space more uniformly

11.4 Other Approaches

- Segregated Lists
  - Keep separated list to manage all objects of that size
  - Hard to determine much memory to dedicate to that list
- Slab allocator by Jeff Bonwick
  - Object caches for kernel objects
  - Each object cache are segregated free lists
  - Requests slabs of memory from general allocator, when running low
- Binary buddy Allocation
  - Big space of 2^N
  - Suffers from internal fragmentation but can recursively coalesce

12 L5: Garbage Collection and Defragmentation

12.1 Garbage Collection

- Allocated resources are freed through explicit/implicit action by client
  - close(2), free(3), delete operator, returning from a C/C++ subroutin, exit(2)
- If shared by multiple concurrent clients
  - Free only if reference count is zero (don't free if others are still using it, just decrement the reference count)
- Garbage Collection
  - Analyzes allocated resources to determine which are still in use
  - Data structures assoc with resource references are designed to be easily enumerated to enable the scan for accessible resources
  - Comes at a performance cost

12.2 Defragmentation

- Shards of free memory are not useful
  - Coalescing is only useful if adjacent memory free at same time
- Defragmentation
  - Changes which resources are still allocated
- Flash management
  - NAND Flash is a pseudo-Write-Once-Read-Many medium
  - Identify large (64MB) block with many 4KB blocks not in use
  - Move all in use blocks and update resource allocation map
  - Erase large block and add 4KB blocks to free list
- Disk Space Allocation
  - Choose region to create contiguous free space
  - For each file in that region, move it elsewhere
  - Coalesce all that free memory
  - Move set of files into that region
  - Repeat until all files and free space is contiguous
- Internal fragmentation is like rust, it never sleeps
  - Defragmentation used to be run periodically, now is run continuously
- Conclusions
  - If using garbage collection, must make all resources discoverable, how to trigger scans, prevent race conditions with application
  - Must not disrupt running applications when using defragmentation

13 L6: Arpaci-Dusseau Chapter 18: Paging: Introduction

13.1 A Simple Example And Overview

- Paging
  - Divide process address space into fixed-sized units
  - View memory as fixed-sized page frames
- Free list
  - OS may hold list of free pages
- Page table is a per process data structure
  - Stores address translations for virtual pages so we know where it is in physical memory
- Virtual address [VPN, OFFSET]
  - Virtual page number (VPN) indexes page table to find physical frame/page number (PFN/PPN)
  - Translate VPN to PPN then load from memory
  - Offset determines which byte within page

13.2 Where Are Page Tables Stored?

- Page table entry
  - Holds physical translation
  - If roughly 4 bytes per PTE, page tables would be big
  - Problem bc page table per process
- Stored somewhere in memory

13.3 Whats Actually In The Page Table?

- Linear Page Table
  - Index array by VPN to look up PTE and to find physical frame number (PFN)
  - Valid bit indicates if memory is valid (traps if invalid)
  - Proction bit indicates whether page can be read/written/executed (trap if bad access)
  - Present bit indicates whether page is in memory or disk (if it has been swapped out)
  - Dirty bit indicates if page has been modified since brought into memory
  - Reference/aceess bit indicates if page has been accessed (to determine which pages are popular; used for page replacement)

13.4 Paging: Also Too Slow

- Must translate virtual address
  - VPN = (Virtual address & VPN_{MASK}) >> SHIFT
  - PTEaddr = Page table base address + VPN * sizeof(PTE)
  - Offset = Virtual address & $OFFSET_{MASK}$
  - PhysAddr = (PFN << SHIFT) — Offset

14 L6: Arpaci-Dusseau Chapter 19: Paging: Faster Translations (TLBs)

14.1 TLB Basic Algorithm

- TLB
  - Bc chopped address space into many fixed-sized units, paging requires a lot of memory to map addresses
  - This mapping memory is also stored in physical memory, which would require an additional memory lookup to read
  - Instead, use a TLB, which is an address translation cache, to hold popular virtual-to-physical translations
- TLB Hit/miss
  - If virtual page number (VPN) from virtual address (VA) is inside the TLB (translation lookaside buffer), then have TLB hit and may extract the page frame number (PFN)

- If VPN from VA is not inside TLB, then have TLB miss and must access page table (in memory) to find translation, update TLB, then restart lookup into TLB

14.2 Example: Accessing An Array

- Start with a miss, then multiple hits
  - Rely on spatial locality for first pass
  - Rely on temporal locality for second pass
- Caching is fundamental
  - Temporal and spatial locality are necessary
  - Can't make caches large because physics says large cache is slow

14.3 Who Handles The TLB Miss?

- Hardware
  - Use page table base register to walk page table and find PTE
- Software
  - Hardware raises exception, pauses instructions, privilege raises to kernel mode, jumps to trap handler
- Infinite TLB misses
  - If is a problem, keep TLB miss handlers in physical memory (unmapped) so it will always be a hit
- RISC vs CISC (Aside)
  - Complex has more and more powerful instructions
  - Reduced has fewer and simpler primitives

14.4 TLB Contents: Whats In There?

- Fully associative means a given translation can be anywhere in the TLB
- VPN — PFN — other bits
  - Other bits include valid bit, protection bits (regarding w/r/x), address space identifier, dirty bit, etc

14.5 TLB Issue: Context Switches

- Fully associative means a given translation can be anywhere in the TLB
- VPN — PFN — other bits
  - Other bits include valid bit, protection bits (regarding w/r/x), address space identifier, dirty bit, etc
  - Could flush TLB on context switch, or could use address space identifier

14.6 Issue: Replacement Policy

- LRU Replacement Policy
  - Least recently used, but usually can't actually do this, so vaguely do LRU

15 L6: Arpaci-Dusseau Chapter 21: Beyond Physical Memory: Mechanisms

15.1 Swap Space

- Swap Space
  - Use hard disk drive as storage
  - Reserved space on disk for moving pages back and forth

15.2 The Present Bit

- Extract VPN from VA, check for TLB hit and produce PA if possible
  - Otherwise, receive TLB miss and go to memory through page table base register to find PTE
- Present bit
  - Set to one if page is in physical memory
  - Otherwise, is not in physical memory and is a page fault
  - OS invoked to service page fault, so page-fault handler runs

15.3 The Page Fault

- OS page-fault handler
  - Hardware does not do it because hardware does not know enough about swap space, I/O, etc
  - OS looks in PTE to find address and request it from disk
  - Process is blocked during this, so run another process

15.4 What If Memory Is Full?

- Page-replacement Policy
  - Page in from swap space; Page out from memory
  - Replace if memory is full
  - 10k-100k times slower if poor page-replacement policy

15.5 Page Fault Control Flow

- If TLB miss
  - If invalid, OS trap handle terminates process
  - If not present, run page fault handler
    - Find physical frame for soon-to-be-faulted-in page
    - Run replacement alg if necessary
    - I/O request page from swap space
    - Retry for TLB miss, then retry for TLB hit
  - If present and valid, grab PFN from PTE and retry

15.6 When Replacements Really Occur

- Swap daemon
  - If fewer pages than the low watermark, then background thread evicts pages
  - Continues evicting pages until the high watermark
  - Then goes back to sleep and waits
- Clustering
  - Clustering/grouping these pages to swap partition increases efficiency because it reduces disk seek and rotational overheads
- Background work
  - Do work in background (buffered disk writes, etc) because it is more efficient and makes better use of idle time

16 L6: Arpaci-Dusseau Chapter 22: Beyond Physical Memory: Policies

16.1 Cache Management

- Minimize cache misses because a single miss will make it very slow
  - Average memory access time (AMAT) $= T_M + (P_{miss} * T_D)$

16.2 The Optimal Replacement Policy

- Farthest in the future
  - Is optimal
  - Use this as a reference point, something to compare our algorithms against
- Types of misses
  - Cold-start miss is compulsory because cache is empty
  - Capacity miss is because cache ran out of space
  - Conflict miss is because of hardware limits on where items can be placed in a hardware cache (not a problem for OS page cache)

16.3 Replacement Policies

- FIFO
  - Performs quite terribly, but is simple to implement
  - Belady's Anomaly: FIFO performs even worse on larger cache than on smaller cache
- Random
  - Can work
- Least-Frequently-Used (LFU)/Least-Recently-Used (LRU)
  - Rely on locality and do what their names say
- Most-Frequently-Used (MFU)/Most-Recently-Used (MRU)
  - Exist and do not work well
- Workload examples
  - FIFO doesn't do well, random can do well, LRU does fairly well

16.4 Implementing LRU

- True LRU is expensive
  - Finding truly least-recently-used page is prohibitively time-consuming
- Approximate LRU using Clock algorithm
  - Whenever page is referenced, use bit is set
  - Clock hand points to some page, if bit is set, unsets it and checks next
  - If bit is unset, replaces it

16.5 Considering Dirty Pages

- If page is dirty (set dirty bit), then must be written back to disk if we want to evict it
  - Prefer to evict clean pages

16.6 Other VM Policies

- Demand Paging
  - Bring page into memory only 'on demand'
  - Opposite of prefetching memory
- Clustering/Grouping of writes
  - Write many things at same time because of how disk drive works

16.7 Thrashing

- If memory is just oversubscribed
  - Then will constantly page and thrash
- Admission control
  - Decide to not run some processes, so that we may do well on the remaining processes
- Out-of-memory killer
  - Will choose a memory-intensive process and kill it

17 L6: Working Sets

17.1 LRU is not enough

- Global LRU
  - Most-recently used page is from current process and will not run for a while
  - Least-recently used page is from old process about to run

17.2 The concept of a Working Set

- Is the set of pages for a given process
  - Increasing the number of pages makes little difference in performance, but decreasing makes a difference
- Different computations require different sizes, getting the number correct will minimize page faults and maximize throughput

17.3 Implementing Working Set replacement

- More information recorded about pages
  - Associated with owning process
  - Accumulated CPU time
  - Last referenced time
  - Target age parameter
- Age decisions are made on the basis of accumulated CPU time
  - Page ages if owner runs without them
  - Pages younger than a target age are preferrably not replaced
  - Give pages older than target age away

17.4 Dynamic Equilibrium to the rescue

- Page stealing algorithm
  - Every process is continuously losing and stealing pages
  - Processes that reference more pages more often will accumulate larger working sets while others will find their set reduced
  - Working sets adjust automatically

18 L7: Arpaci-Dusseau Chapter 25: A Dialogue on Concurrency

18.1 Dialogue

- Multi-threaded applications
  - Threads access memory; we don't want multiple threads to access memory at same time
  - OS supports primitives such as locks and condition variables

19 L7: Arpaci-Dusseau Chapter 26: Concurrency: An Introduction

19.1 Introduction

- Context switch
  - Save state (program counter, registers) to thread control block
  - Address space stays the same, so page table does not need to be switched
- Multiple stacks in address space if multiple threads
- Thread-local storage
  - Stack of that thread

19.2 Why Use Threads?

- Used threads to exploit parallelism
  - If single processor, then not relevant
  - Otherwise, parallelize and used thread per CPU
- Use threads to do someting when blocked program
  - Instead of waiting for IO, just switch to another thread and do things
- Choose process for logically separate tasks with little sharing of data structures

19.3 An Example: Thread Creation

- Use pthreads
- Will run in different order according to scheduler
- May not be deterministic

19.4 The Heart Of The Problem: Uncontrolled Scheduling

- Race condition
  - Execution depends on timing execution of code (indeterminate)
- Critical section
  - Multiple threads executing code resulting in race condition
- Mutual exclusion
  - If one thread executing inside critical section, others will be prevented

19.5 The Wish For Atomicity

- Atomic (all or nothing)
  - Don't just have atomic instructions for all because too many instrucions
- Synchronization primitives
  - General set of instructions to control multi-threaded programs

20 L7: Arpaci-Dusseau Chapter 27: Interlude: Thread API

20.1 Threads

- Use pthreat_create to create new thread
- Use pthread_join to wait for thread to complete
- Use pthread_mutex_lock and pthread_mutex_unlock to provide mutual exclusion to critical sections via locks
  - Need to properly initialize and check that lock/unlock actually succeed
- Condition variables
  - Enables thread to wait until particular condition occurs
  - Needs lock and condition
  - Sleeps until other thread signals
- Spinlock
  - Wait in loop until lock available, consuming CPU cycles

21 L7: User-Mode Thread Implementation

21.1 Introduction

- Threads are independent schedulable unit of execution
  - Runs within address space of process
  - Has access to system resources from process
  - Has own registers and stack

21.2 User/Kernel

- User-level thread done without OS
  - Allocates memory, dispathes thread, sleeps, exits, free memory
  - If system call blocks, entire process blocks
  - Cannot exploit multi-processors
- Kernel implemented threads
  - Exploits multi-processors and switches between threads when one blocks

21.3 User/Kernel

- Non-preemptive scheduling
  - User-mode threads are more efficient than kernel for contex-switches
- Preemptive scheduling
  - Allowing OS to schedule is better than setting alarms and signals

22 L7: Inter-Process Communication

22.1 Introduction

- coordination of operations with other processes
  - synchronization (e.g. mutexes and condition variables)
  - the exchange of signals (e.g. kill(2))
  - control operations (e.g. fork(2), wait(2), ptrace(2))
- the exchange of data between processes:
  - uni-directional/bi-directional

22.2 Simple Uni-Directional Byte Streams

- Pipes
  - Opened by parent and inherited from child
  - Each program in pipeline is unaware of what others do, byte streams are unstructured, etc
  - If reader exhausts data in pipe, reader does not get EOF (is blocked instead)
  - Flow control: Available buffering capacity of pipe may be limited, so writer may be blocked for reader to catch up
  - Writing to pipe without open read fd is illegal (gets signal exception)
  - When both read/write fd are closed, pipe file is deleted
- Only data privacy mechanisms are on initial/output file
  - Generally no auth/encryption while passing

22.3 Named Pipes and Mailboxes

- Named-pipe fifo
  - Persistent pipe whos reader/writers can open by name (rather than inheriting)
  - Writes may be interspersed
  - Readers/writers can't authenticate identity
- Mailboxes
  - Data is not bytestream, each write is stored as message
  - Each write has authenticated ID
  - Unprocessed msgs remain in mailbox

22.4 General Network Connections

- Higher level communication/service models
  - Remote procedure calls - distributed request/response APIs
  - RESTful service models - layered on HTTP GETS/PUTS
  - Publish/Subscribe services - content based info flow
- Complexity
  - Interoperability with software running different OS and ISA
  - Security issues, changing addresses, failing connections

22.5 Shared Memory

- High performance for Inter-Process Communication
  - Efficiency wrt low cost per byte
  - Throughput wrt bytes per second
  - Latency wrt minimum delay
- Ultra high performance
  - Shared memory by creating a file for communication
  - Process maps file into virtual address space
  - Is available immediately upon writing
  - Very fast but can only be used on same memory bus
  - Has no authentication and a single bug can kill both

22.6 Network Connections and Out-of-Band Signals

- Preempting queued operations
  - Have a reserved out-of-band channel so signal can preempt others if urgent
  - Adds overhead but allows important messages to skip FIFO line (network connection is FIFO)

23 L7: Named pipes, Send, Recv, Mmap

23.1 Named Pipes

- Named pipes exist as device special file
- Can be accessed by processes of different ancestries
- When I/O done, pipe remains
- Normally, if FIFO opened for reading, process will block until another process opens it for writing
- If write to pipe without reader, will get SIGPIPE

23.2 Send, Recv, Mmap

- Send
  - Send a message on a socket
- Recv
  - Receive a message from a socket
- Mmap
  - Map or unmap files or devices into memory
  - Creates a new mapping in the virtual address space of the calling process

24 L8: Arpaci-Dussseau Chapter 28: Locks

24.1 Locks: The Basic Idea

- Use of lock
  - Put around critical sections so that it is performed atomically
- Lock variable
  - If no other thread holds lock, thread acquires lock and enters critical section
  - If another thread holds lock, then will not return

24.2 Pthread Locks

- Mutex is the POSIX library lock
  - Provides mutual exclusion (exclude other threads from entering until first thread has completed)
- Use multiple locks (as opposed to one big lock for any critical section)
  - Fine-grained vs coarse-grained approach

24.3 Evaluating Locks

- Mutual exclusion, fairness, performance
  - Needs to prevent multiple threads from entering critical section
  - Needs to not let contending threads starve
  - Need time overheads to not be high

24.4 Controlling Interrupts

- Disable interrupts during critical section to provide mutual exclusion
  - For single-processor system, makes code atomic
- Cons
  - Requires user to call privileged operation
  - Greedy user could lock for entire process
  - Buggy user could break computer
  - Does not work for multiprocessors because multiple threads can still enter critical section
  - Interrupts may be lost
- OS is allowed to use this as mutual-exclusion primitive for updating data structures

24.5 A Failed Attempt: Just Using Loads/Stores

- Flag
  - Doesn't work because the checking/setting of flag is not atomic
  - Also spin-waiting (persistently checking value of flag) is incredibly inefficient

24.6 Building Working Spin Locks with Test-And-Set

- Test-and-set instruction (atomic exchange)
  - Puts new value into old value; returns old value (atomically)
  - Is sufficient to build a spinlock
- Spinlock
  - To work on a single processor, requires preemptive scheduler (otherwise, thread would never relinquish CPU)
  - Is a correct lock
  - No fairness guarantees
  - Terrible performance if single processor
  - If N threads contending, N-1 time slices may be wasted while spinning on single processor
  - Okay performance if multiple processors

24.7 Compare-And-Swap

- Compare-and-swap (compare-and-exchange)
  - Test if value of ptr is equal to value at expected, if so, update with new value, otherwise, do nothing
  - Can build spinlock with this

24.8 Load-Linked and Store-Conditional

- Load-linked
  - Fetch value from memory and put in register
- Store-conditional
  - If success, updates and returns 1; otherwise, no update and returns 0
  - Only one thread is able to acquire lock if using these (because store-conditional will fail)

24.9 Fetch-And-Add

- Fetch-and-add
  - Increment value and return old value
- Ticket lock
  - If thread wants lock, do fetch-and-add and wait
  - Global lock-¿turn determines who's turn
  - All threads make progress

24.10 A Simple Approach: Just Yield, Baby

- Yield
  - System call yield to allow processes to deschedule self
  - Better than spinlock, but still costly
  - Does not address starvation issue

24.11 Using Queues: Sleeping Instead Of Spinning

- Sleep and wake
  - Test-and-set with explicit queue of lock waiters
  - Avoids starvation
  - May sleep forever in wakeup/waiting race if release of lock occurs after park()
  - So have setpark() to indicate a thread is about to park; if interrupted and another thread unparks, thread will return rather than sleep
- Solaris uses park/unpark
- Linux uses futex

24.12 Two-Phase Locks

- Spins during first cycle, then on second cycle will sleep
- Hybrid approach is effective

25 L8: Arpaci-Dusseau Chapter 30.1: Condition Variables

25.1 Definition and Routines

- Condition variable
  - Explicit queue for threads if condition is not met
  - When condition is correct, wakes

26 L9: Arpaci-Dusseau Chapter 29: Lock-based Concurrent Data Structures

26.1 Concurrent Counters

- Simple but not Scalable Counter
  - Use pthread_mutex_lock/unlock to create a threadsafe, concurrent data structure
  - Poor performance if multiple threads (compared to one thread)
- Sloppy Counter for Scalable Counting
  - Numerous local physical counters (one per CPU core) as well as single global counter
  - Periodically transfer local values to global counter; reset local counter
  - How often we transfer is determined by S, the sloppy threshold

26.2 Concurrent Linked Lists/Queues/Hash tables

- Concurrent Linked List
  - Global lock or hand-over-hand lock
  - Hand-over-hand locking tends to be too slow
- Concurrent Queue
  - Lock for head and tail
  - Dummy node to separate head and tail operations
- Concurrent Hash table
  - Lock per hash bucket has good performance
- More concurrency may not be faster if there are many locks

27 L9: Arpaci-Dusseau Chapter 30.2+: Condition Variables

27.1 The Producer/Consumer (Bounded Buffer) Problem

- Bounded Buffer
  - Producer puts data into it, consumer gets data from it
  - Is a shared resource which requires synchronization
- Mesa Semantics
  - No guarantee that the state will still be as desired when a woken thread runs
  - Need to check condition before run, so while loop
- Hoare Semantics
  - Stronger guarantee that woken thread will run immediately once woken
- Two condition variables
  - Producers wait on empty and signal filled
  - Consumers wait on filled and signal empty
- Spurious wakeups
  - If multiple threads are woken up
  - Use while loop to recheck condition a thread is waiting on
- Correct solution
  - Producer sleeps only if all buffers are filled
  - Consumer sleeps only if all buffers are empty

27.2 Covering Conditions

- Memory allocator problem
  - If multiple threads sleep because need different amounts of memory (100, 10)
  - Other thread frees memory (50), how does it know which thread to wake?
- Memory allocator solution
  - Covering Condition will broadcast a signal to wake up all threads

28 L9: Arpaci-Dusseau Chapter 31: Semaphores

28.1 Semaphores: A Definition

- Semaphore
  - Sem_wait() or P()
    - Decrement counter and wait if negative
  - Sem_post() or V()
    - Increment counter and wake up a thread if one is sleeping

28.2 Binary Semaphores (Locks)

- Example of holding the lock
  - Thread 0 calls sem_wait() (decrements counter from 1 to 0), takes lock, and enters
  - Thread 1 calls sem_wait() (decrements counter from 0 to -1) and waits
  - Thread 0 runs and calls call_post() (increment counter from -1 to 0) and wakes thread 1
  - Thread 1 runs and calls call_post() (increment counter from 0 to 1)
- Binary semaphore
  - Call a semaphore used only as a lock (held or not held) a binary semaphore

28.3 Semaphores For Ordering

- Semaphores for child to run before parent
  - Initiate counter to 0, parent calls sem_wait(), child calls sem_post(), parent finishes
  - Initiate counter to 0, child calls sem_post(), parent finishes

28.4 The Producer/Consumer (Bounded Buffer) Problem

- Multiple producers/consumers
  - Need mutex because filling buffer and incrementing index of buffer is a critical section
  - Mutex should be inside of the wait()/post() to avoid deadlock

28.5 Reader-Writer Locks

- Reader-writer lock
  - Multiple reads can occur at same time if no write
  - Write has to wait if there are reads going on
- Overhead might make it not worthwhile wrt performance

28.6 Dining Philosophers

- Problem
  - There are 5 philosophers at around table with 5 forks
  - Each philosopher wants to think (does not need any fork) and eat (needs both left and right fork)
- Solution
  - Each philosopher, except one, grabs right fork before left fork
  - Last philosopher grabs left fork before right fork
- If all philosophers grabbed forks in same order, a deadlock may occur

29 L9: Flock and Lockf

29.1 Man

- flock
  - Apply or remove an advisory lock on the open file

- A single file may not simultaneously have both shared and exclusive locks

– lockf

- Apply, test or remove a POSIX lock on a section of an open file

30 L10: Arpaci-Dusseau Chapter 32: Common Concurrency Problems

30.1 Non-Deadlock Bugs

- Atomicity-Violation Bugs
  - Ex: First thread performs check for not null, then dereferences; Second thread sets value to null
  - The desired serializability among multiple memory accesses is violated
  - Can use locks to fix this
- Order-Violation Bugs
  - Ex: First thread inits and uses; second thread just uses (assumes it was init)
  - Can use condition variables to enforce order

30.2 Deadlock Bugs

- Hard to find deadlock bugs
  - Encapsulation + modularity does not mesh with locking
- Conditions for Deadlock
  - Mutual Exclusion - threads claim exclusive control of resources they require
  - Hold-and-wait - threads hold resources while waiting for additional resources
  - No preemption - resources cannot be forcibly removed from threads holding them
  - Circular waits - circular chain of threads each holding a resource required by the next thread in the chain
  - All these conditions must occur together for deadlock to occur
- Prevention
  - Circular wait
    - provide total/partial ordering such that no cyclical waiting may occur
    - Is difficult to find order, and is only a convention/suggestion (not enforced
  - Hold-and-wait
    - Prevention lock prevents deadlock from context switch during lock acquisition
  - No Preemption
    - Use trylock to grab the lock or return error if it doesn't work
    - Livelock could occur (two threads repeatedly failing to acquire both locks)
    - Difficult because would have to release everything acquired
  - Mutual Exclusion
    - Use lock-free and wait-free data structures (built using powerful hardware instructions)
  - knowledge of which threads need which locks
    - Schedule threads on multiple CPU's such that threads which need the same lock are never run together
    - Rarely used approach
  - Detect and Recover
    - Tom West's Law - "Not everything worth doing is worth doing well"
    - Just let the computer freeze if it only happens once/yr on consumer PC
- Use a different concurrency model
  - MapReduce (from Google)

31 L10: Deadlock avoidance

31.1 Introduction

- Deadlock arises from exhaustion of critical resource
  - This time, the problem is not a resource dependency graph
  - Problem is that some processes will free up memory once complete, but require additional memory to complete
  - Solution is to refuse to grant requests which would put the system in a dangerously resource-depleted state

31.2 Reservations

- Declining allocation mid-operation is hard
  - Solution is to make processes reserve resources beforehand
  - Extends to creating new files, processes, and sockets

31.3 Over-booking

- Few users request their maximum resource allocation
  - So is relatively safe to over-book resources
  - Airlines and networks both do this
  - OS does not do this with memory because running out of memory and having to kill a process is extremely bad
- Dealing with Rejection
  - OK to reject requests, because the error is clean and we can move on

32 L10: Health Monitoring and Recovery

32.1 Introduction + Health Monitoring

- Formal deadlock detection is difficult to perform + inadequate for most problems
- Health Monitoring
  - Internal monitoring agent has transaction log
    - This agent may fail
  - Clients submit failure report to central monitoring service
    - But maybe other requests failed
  - Server sends heart beat to central monitoring service
    - This does not tell if server is serving requests
  - External health monitoring service periodically sends tests to servers
    - But maybe other requests failed
- Use a combination of methods

32.2 Managed Recovery

- Highly available services
  - Must be designed to be killed and restarted at any time
- Restart types
  - Warm-start - restore from last saved state
  - Cold-start - ignore any saved state and restart from scratch
  - Reset and reboot - reboot system and cold-start
  - Restart single process
  - Restart groups of processes
  - Restart software on a node
  - Restart groups or nodes or entire system

32.3 False Reports and Other Restarts

- Tradeoff between cancelling all requests for a restart vs. prolonging outage
  - Misdiagnosing problem may be even worse
  - Preferable for server to detect its own problems and restart components
- Non-disruptive rolling upgrades
  - If system can operate without some of its nodes, then restart nodes one-at-a-time while upgrading
  - New software must be upwards compatible; must be able to roll-back to previous release
- Prophylactic reboots

- Periodically reboot system because it seems to fix problems
- Systems become slower as time progresses, so just restart it regularly

33 L10: Java Synchronized Methods

33.1 Synchronized Methods

– Add to declaration of function
- Two invocations of synchronized methods on same object will not interleave
- Synchronized method has happens-before relationship with any subsequent invocation of synchronized method

– Constructors cannot be synchronized (because it does not make sense)

34 L10: Java Intrinsic Locks + Synchronization

34.1 Intrinsic Locks and Synchronization

– Internal entity known as intrinsic lock or monitor lock
- If thread owns lock, no other thread can acquire
- Once thread releases, happens-before relationship established between this and subsequent acquisitions of the lock
- When thread invokes synchronized method, automatically tries to acquire lock
  – Releases lock on return (even if exception)

34.2 Synchronized Statements

– Create synchronized code by specifying object that provides the intrinsic lock
- Fine-grained concurrency

– Reentrant synchronization
- Synchronized code may call a method which also contained synchronized code, where both sets of code use the same lock
- Synchronized blocks in java are reentrant; therefore, a thread may enter other synchronized blocks on same ock

35 L10: Monitors

35.1 Monitors

– Synchronization construct
- Has both mutex and condition variables
- Thread-safe class that uses mutual exclusion to safely allow access to a method/variable by more than one thread
- By using multiple condition variables, can allow threads to wait on certain condition

36 L10: Measuring Operating Systems Performance

36.1 Metrics

– Metrics should be numerical and relevant to your goals
– Need to be practically measurable and measurable in ways which don't affect the measured value

36.2 Complexity and the Role of Statistics in Measurement

- Statistics
  - Need to measure many times usually should report median, mean, range, standard deviation
- Think first, measure second
  - There are many options in software, should not measure possibility, because combinatorial explosion
  - Measure things that are necessary and measure enough to be confident
  - Some things are hard to account for, like hardware caching, so metrics will vary across multiple runs

36.3 Workloads

- Workloads
  - Traces, live workloads, standard benchmarks, simulated workloads
  - Traces are good because realistic and but not easy to reconfigure system; also privacy is a concern
  - Live workloads are good because realistic and can capture a range of behavior, but not easy to control behavior
  - Standard benchmarks are good because easy integrate, easy comparison, are scalable and bugfree, but may be limited in testing
  - Simulated workloads are good for ease of varying models and flexibility in scaling, but only as useful as the quality of tests

36.4 Common Mistakes in Performance Measurements

- Measure things realistic and pertinent to real-world behavior
- Measure latency with regard to utilization (fully loaded system)
- Report variability of measurements with distribution type (bimodal, modal, uniform, etc)
- Do not ignore special cases
  - Do not ignore startup effects (caching is significant)
  - These sort of things may or may not be relevant to what you are trying to measure
- Do not ignore effect of measurement program on measurements
- Don't toss any data + label each case well
- Use the numbers to attain wisdom; remember the goal

37 L11: Load and Stress Testing

37.1 Introduction

- Functional Validation
  - Establish conditions, perform operations, confirm assertions
- Load and Stress Testing
  - Run a set of tests many times, may not even care about return values, just care that it runs

37.2 Load Testing

- Performance Metrics
  - Response time, throughput, CPU time/utilization, disk I/O utilization, network packet utilization
- Load Generators
  - Generate test traffic at calibrated rate on the whole system
  - Test load for request rate, mix, and particular request patterns
- Performance assessment
  - Deliver at specified rate and measure response time
  - Deliver requests at increasing rate until max throughput
  - Deliver requests and use this as background for performance of other system services
  - Deliver requests at a rate and use this as a test
- Accelerated Aging
  - Simulate high rate of traffic to accelerate aging

37.3 Stress Testing

- Use randomly generated complex usage scenarios to increase likelihood of unlikely events
- Generate large number of conflicting requests
- Introduce wide range of errors
- Introduce wide changes in load
- Perform once-a-year type situations hundreds of times per minute
- Few products survive, but necessary for mission-critical programs

38 L12: Arpaci-Dusseau Chapter 33-33.6: Event-based Concurrency (Advanced)

38.1 The Basic Idea: An Event Loop

- Event-based concurrency
  - Wait for event to occur, then when it occurs, check what event it is and do its work
  - Addresses issue that managing thread-based concurrency is difficult because of locks and deadlocks
  - Addresses issue of lack of control over scheduling
- Event loop
  - Process events with event handler
  - Deciding which event to handle next is equivalent to scheduling
- select() or poll()
  - Allow server to know if new packet has arrived and when it's OK to reply
  - Timeout argument allows you to decide how long to block
- Asynchronous vs Synchronous
  - Non-blocking vs blocking

38.2 A Problem: Blocking System Calls

- Simpler because no locks
  - Has no locks because single threaded
- No blocking calls are allowed
  - Because only one thread, so if event loop blocks, entire system blocks

38.3 A Solution: Asynchronous I/O

- AIO Control Block
  - Enables asynchronous IO
- Async IO
  - Issue call, return immediately if successful
  - Periodically poll to check if IO complete
  - Or use interrupts (Unix signals)

39 L12: Arpaci-Dusseau, Chapter 35: A Dialogue on Persistence

39.1 Dialogue

- Persistence is relevant
  - Hard to make data persist despite power outages, disk failures, computer crashes

40 L12: Device Drivers: Classes and Services

40.1 Introduction

- Device Drivers
  - Generalizing Abstractions
    - Few general classes/models/behaviors/interfaces to be implemented by all drivers for a given class
  - Simplifying Abstractions
    - Implementation of standard class interface which opaquely encapsulates details
- Object oriented code reuse in device drivers
  - Similar high-level behavior despite underlying device
  - Minimize cost of developing drivers
  - Ensure benefits of improvements accrue to old drivers, not just new ones

- Derive device driver subclass from major class
- Define subclass interfaces
- Create per-device implementations

40.2 Major Driver Classes

– Block Devices
  - Random access, addressable in fixed-sized blocks
  - Request method to enqueue async DMA requests
  - Used within OS to access disks
– Character Devices
  - Sequential or byte-addressable
  - Used directly by applications bc potential for DMA between device and user-space buffer
– Even in oldest UNIX, device drivers were divided into distinct classes (but not mutually exclusive)

40.3 Driver sub-classes

– Device Driver Interface
  - Subclass specific interfaces
  - Identical behavior over many devices
  - Much important functionality implemented at higher level
  - Device must conform to interface or will not function w higher level software

40.4 Services for Device Drivers

– Nearly impossible to fully implement driver by yourself
  - Use OS Services like dynamic memory allocation, IO, bus management, condition variables, mutual exclusion, interrupts, DMA, scatter/gather, configuration/registry services, etc
– Driver-Kernel Interface
  - Collection of services exposed by OS available for device driver use
  - Requirement to maintain stable DKI constrains OS; but if did modify DKI in a noncompatible way, then many drivers would fail
– Conclusion
  - Many drivers demonstrate evolution from basic super-class (character devices) into sub-classes
  - Each subclass inherits lots of higher level frameworks which do most work

41 L12: Arpaci-Dusseau, Chapter 36: I/O Devices

41.1 System Architecture

– CPU attached to main memory via memory IO bus
– Some devices connected via general IO bus
– Slowest devices connected using peripheral IO bus

41.2 A Canonical Device

– Hardware interface is presented to rest of system
– Internal structure is implementation/device specific
– Firmware is software within hardware to implement functionality

41.3 The Canonical Protocol

- Device Interface
  - Status register tells current status
  - Command register tells device to do command
  - Data register used to pass data to/from device
- Protocol in 4 steps
  - Device polling, ready for command
  - OS sends data to data register
  - OS writes command to device to start doing work
  - OS waits for device to finish by polling in a loop
- Programmed IO
  - When CPU involved w data movement

41.4 Lowering CPU Overhead With Interrupts

- Interrupt service routine / Interrupt handler
  - CPU will jump to handler if hardware interrupt
- Interrupts allow for overlapping routines, instead of constantly just polling, can do real work
  - There are times where interrupts will be slower, because the task is so short that the first poll will already find the device finished
  - Hybrid approach between polling and interrupts may be the best
  - Chance for livelock if using interrupts
  - Coalescing is when you wait for a bit after receiving interrupt, so that multiple interrupts can be coalesced into a single interrupt delivery

41.5 More Efficient Data Movement With DMA

- DMA
  - OS tells DMA engine where data is, how much to copy, and where to send it
  - Then OS is done w transfer and can do other work
  - DMA engine raises interrupt once transfer is complete

41.6 Methods Of Device Interaction

- Explicit IO Instructions
  - Caller specifies register with data in it and port with device name
  - Execute privileged instruction
- Memory Mapped IO
  - Hardware makes device registers available as memory locations
  - Load/store to device instead of main memory

41.7 Fitting Into The OS: The Device Driver

- Device Driver
  - Encapsulates specifics of device interactions
  - Devices with special capabilities often lose this due to the generic interfaces
  - Roughly 70 percent of Linux OS code is from device drivers

42 L12: Arpaci-Dusseau, Chapter 37: Hard Disk Drives

42.1 The Interface

- Address space
  - Is from 0 to n-1 for n sectors

- Torn write
  - Many manufacturers only guarantee 512-byte write as atomic, so larger maybe torn if power loss
- Assume sequential access is faster than random access

42.2 Basic Geometry

- Platter is circular hard surface where data is stored by inducing magnetic charges
- Disk may have multiple platters, each with two surfaces
- Platters bound around spindle, spinning around 10000 rotations per minute
- Concentric circle of sectors is a track; hundreds of tracks fit into width of human hair
- Disk head does read/write; disk arm positions head over desired track

42.3 A Simple Disk Drive

- Rotational Delay
  - Time for platter to complete half of a full revolution (1/2 R)
- Seek time
  - Acceleration, Coasting, Deacceleration, Settling phases
  - Then transfer
- Track Skew
  - Skew tracks relative to each other so that sequential reads still get good performance even when switching tracks
- Multi-zoned
  - Outer tracks have more sectors than inner tracks because geometry
- Cache/track buffer
  - Just a cache, maybe all sectors on the track so that disk can quickly respond to sequential requests
- Write back vs Write Through
  - Write back is to memory (faster), write through is to disk (more correct)

42.4 I/O Time: Doing The Math

- IO Time = Seek + Rotation + Transfer
  - R_{IO} = Size of transfer / IO Time
- Average seek is 1/3 the full distance because math
- Performance differs between drives, especially if you optimize for diff things

42.5 Disk Scheduling

- Shortest Seek Time First (SSTF/SSF)
  - Pick requests on nearest track to complete first
  - Has problem that OS does not know geometry
  - Nearest Block First (NBF)
    - Pick nearest block address first
  - Also has problem of starvation
- Elevator (SCAN/CSCAN)
  - Move back and forth across disk, servicing requests in order across tracks
  - FSCAN freezes queue when servicing, so that faraway requests don't starve
  - CSCAN only sweeps from outer to inner, to add fairness, bc otherwise the middle gets serviced twice
- Shortest Positioning Time First (SPTF/SAFT)
  - Things just depend on speed, just get whatever is fastest to move to
- Drive does SPTF scheduling
- IO Merging
  - Merge requests that are near each other, so that you can make fewer overall requests
- Work conserving

    - Issue request to drive immediately
- Anticipatory disk scheduling / non-work conserving
    - Wait a little bit, and see if you can get a better request, thus increasing overall efficiency

43 L12: Arpaci-Dusseau, Chapter 38: Redundant Arrays of Inexpensive Disks

43.1 Interface And RAID Internals

- RAID
    - Multiple disks to build faster, bigger, more reliable system
    - Transparent to OS, appears as a large disk, so easy to deploy
- One logical IO may translate to multiple physical IO, depending on type of RAID

43.2 How To Evaluate A RAID

- Fail-stop fault model
    - Disk either is working or has failed; also assume that failure is easy to detect
- Three axes
    - Capacity, Reliable, Performance
- Best chunk size depends on workload

43.3 RAID Level 0: Striping

- Striping
    - Spread sequential blocks across disks to parallelize requests for contiguous chunks of the array

43.4 RAID Level 1: Mirroring

- Mirroring
    - Two copies of all data; expensive single-failure solution
    - RAID Consistent Update Problem
        - If system crashes after write to one disk but not the second
        - So use write-ahead log in memory to record actions
    - Random reads are good, because can distribute reads across disks

43.5 RAID Level 4: Saving Space With Parity

- Parity
    - Compute XOR of all bits (or block) from a stripe, and put it in another disk
    - If lose any one disk, can recover
- Full-stripe write
    - Write all blocks on stripe and compute parity at same time
- Additive parity
    - Read all data blocks in stripe and compute parity
- Subtractive parity
    - Flip parity if new bit is different from old bit
- Throughput under small random writes is terrible

43.6 RAID Level 5: Rotating Parity

- Rotating parity
    - Address small-write problem by rotating parity block across drives (remove parity-disk bottleneck)

43.7 RAID Comparison: A Summary

- Striping
  - Best performance
- Mirroring on RAID 1
  - Simple, reliable, decent performance, but more expensive
  - Average write time a little higher than if writing to just one disk
- RAID 4/5
  - Gives capacity and reliability, but worse performance
- Other RAIDS exist
  - Deal with multiple disk failures, fault handling, and software RAIDS

44 L12: Dynamically Loadable Kernel Modules

44.1 Introduction

- Provide common framework
  - Fill in for problem-specific implementations later
- Device drivers as dynamically loadable modules
  - Good for device drivers because theres so many of them
  - After-market addable, delivered by someone other than the OS maker, hot-pluggable

44.2 Dynamically Loaded Module

- Choosing Which Module to Load
  - Today, self-identifying devices; before, had plugin registry or probe system
- Loading
  - If module self contained, then just allocate memory and load
  - If module needs other functions and unresolved external references, then run-time loader
- Run-time loadder
  - Usually will return vector with pointer to initialization function
  - Then call this func, allocate memory/IO, init data strucs
- Using DLM
  - OS shows device instance as special file
- Unloading
  - Unregister, shutdown devices, return allocated memory/IO

44.3 Criticality of Stable Interfaces/Hot-Pluggable Devices and Drivers/Summary

- Tension between new hardware/software features and retaining compatability
- Hot-plug busses can generate events
  - Hot-plug manager subscribes to hot-plug events, loads drivers, removes devices
- Stable interfaces are essential bc methods implemented and services provided need to be standardized
- Device drivers fall into the category of dynamically loaded modules
  - Select module to load, dependencies, init, shutdown, binding, defining/managing interfaces are some characteristics

45 L13: Arpaci-Dusseau Chapter 39: Interlude: Files and Directories

45.1 Files and Directories

- File
  - Linear array of bytes
  - Has lowlevel name (number) known as inode number
  - Type (like .txt) is just a convention, file does not need to actually be of that type
- Directory
  - Also has inode number
  - Specifically has list of pairs of userreadable names with corresponding inode number
  - Directory hierarchy starts with root directory

45.2 The File System Interface

- Creating Files
  - open() and creat() return file descriptors (integer) private per process
- Reading and Writing Files
  - Use strace to see system calls; read() and write() are used
- Reading And Writing, But Not Sequentially
  - If reading from random offsets, use lseek() (also a sys call) to change value of variable in kernel
- Writing Immediately with fsync()
  - Use fsync() to force all dirty data to disk
- Renaming Files
  - Use rename(), which is atomic
- Getting Information About Files
  - Find this information in the inode
- Removing files
  - Use unlink()
- Making Directories
  - Use mkdir()
- Reading Directories
  - Use opendir(), readdir(), closedir()
- Deleting Directories
  - Use rmdir()

45.3 Hard Links and Symbolic Links

- link() creates another name in directory you are linking to and refers it to the same inode number
  - Two human names refer to same file
- Creating a file
  - Is making inode structure, linking human-readable name to file, and putting file in directory
- Link count/reference count
  - Tracks how many different file names have been linked to particular inode
  - unlink() decrements count, breaks link, and only deletes if zero references to inode
- Symbolic/soft link
  - Hardlinks cant link to directories or files in other disk partitions
  - Symbolic link is actually a file itself (is just a soft link file)
  - Removing original file but keeping soft link will create dangling reference

45.4 Making and Mounting a File System

- Making a filesystem
  - Use mkfs to make empty filesystem with root directory onto disk partition
- Mount port
  - Use mount(), which puts directory tree at the specified point

46 L13: Arpaci-Dusseau Chapter 40: File System Implementation

46.1 The Way To Think

- Data structures store data and metadata
- Access methods maps methods like open() and read() onto structures

46.2 Overall Organization

- Divide disk into blocks
- Data Region holds user data
- Metadata holds info like where files are, file sizes, permissions
  - This data is put in inode and goes into inode table (array of inodes)
- Allocation structures
  - Track whether inodes and data blocks are allocated
  - Freelist to track free blocks
  - Or could use data bitmap for data region
  - And an inode bitmap for inode table
- 0th block is superblock
  - Holds info about this file system, like how many inodes and data blocks, where inode table is, etc
  - When mounting, OS will read this
- Superblock — inode bitmap — data bitmap — inodes — data region

46.3 File Organization: The Inode

- Index node (Inode)
  - Low-level number with inumber
  - To read, multiply number by sizeof(inode) + start address of inode table
  - Holds metadata and some sort of pointer to the data itself
- The Multi-Level Index
  - Indirect pointer points to block that contains direct pointers
  - Double indirect points to block that contains indirect pointers
  - Triple indirect points to block that contains double indirect pointers
- Extents instead of pointers
  - Disk pointer with length (in blocks)
  - Limiting because hard to find very big contiguous space, so use multiple extents
- Typical traits of file systems
  - Most files are small (2k)
  - Average file size is growing (200k)
  - Most bytes are stored in large files
  - File systems hold a lot (100k) files
  - File systems are about half full
  - Directories are small, with under 20 entries

46.4 Directory Organization

- Directory entry
  - Each has inum, reclen, strlen, and name
  - Deleting file can leave gap, so use reclen; also set inode num to zero if delete
  - Directory is entry in inode table; has pointers to data region (like other inodes) and directory entries are in this data region
  - Directory data holds name -¿ inode number mappings

46.5 Free Space Management

- Use bitmaps to track free blocks and free inodes
  - Could use free list, superblock has pointer to first free block, which then holds pointer to next free block
  - Mark bit in bitmap with 1 if allocated
  - Allocation structures only consulted when allocating, not when only reading
- Preallocation policy
  - Find set of contiguous blocks when allocating space (improves performance)

46.6 Access Paths: Reading and Writing

- Read
  - Recursively follow absolute path
    - Start from root (which we just know somehow, usually is inode num 2)
    - Search directories until we find the next piece, and repeat
    - Eventually find the correct inode num matching the name, allocates file descriptor in per process open file-table, and returns fd to user
- Write
  - May have to allocate a block
    - And update allocation structures (data and inode bitmap)

46.7 Caching and Buffering

- System memory DRAM to cache important blocks
  - Early systems had fixed-sized cache
  - Later, used dynamic partitioning approach; integrate virtual memory pages and file system pages into unified page cache
  - So first IO is costly, but later ones mostly just hit the cache
- Write buffering
  - Delay writes and batch updates (wait 5-30secs)
  - Also allows you to schedule writes efficiently
  - Many databases can't do this, and use fsync() to force, or direct IO, or the raw disk interface

47 L13: File Types and Attributes

47.1 Ordinary Files

- Blob of zeroes and ones, meaningful when interpreted by program which understands underlying data format
  - Textfile is bytestream delimited by newline and rendered as characters
- Data Types and Associated Applications
  - Require user to specifically invoke correct command to process the data
  - Consult registry that associates program with file type
  - Or just use suffix or magic number or some attribute
- File Structure and Organization
  - Databases evolved from indexed sequential files to relational databases to key-value stores

47.2 Other file types

- Directories are just files
  - Representation of namespaces, associations of names with blobs of data
  - Directories hold files from multiple users; keyvalue stores contains data of one user
- Pipe
  - Inter-Process Communications Ports
  - Use write() and read() on file descriptors
  - Can use open() on named pipes
- IO Devices
  - Many sequential access devices (keyboards and printers) are byte-stream
  - Random access devices (disks) use read()/write()/seek() model
  - Other devices simply have memory mapped to it and are accessed directly, rather than through byte-stream

47.3 File Attributes

- System Attributes
  - Type (regular, directory, pipe, device, symbolic link)
  - Ownership (identify owning user/group)
  - Protection (permitted access)
  - File create/update/access time
  - Size
- Extended Attributes
  - Encrypted, signed, checksummed, internationalized localisations
  - Other name=value attributes
  - Resource forks have more info
- Diversity of Semantics
  - POSIX operations are standard
  - All the other ones are really not standard

48 L13: Object Storage/Key-value database/FUSE

48.1 Object Storage

- Allows addressing and identification of individual objects by more than just filename and path
- Separates metadata from object to support additional functionalities
- Data is stored/managed as objects rather than files or blocks

48.2 Key-value database

- Associative arrays
  - Stored and retrieved using key
  - Opposite of relational database
  - May save memory relative to RDB because doesnt need to store optional parameters

48.3 FUSE

- Filesystem in Userspace
  - Software interface for Unix-like OS that lets non-privileged users create own filesystem in userspace through bridge to kernel interfaces
  - Good for virtual file systems; don't actually store data, just translating

49 L13: Introduction to DOS FAT Volume and File Structure

49.1 Introduction/Structural Overview

- BIOS
  - BASIC IO Subsystem meant to provide runtime support for BASIC interpreter
- DOS FAT system still used
- Structural Overview
  - Bootstrap
    - Code to be loaded into memory and executed when computer is turned on
  - Volume descriptors
    - Info describing the size, type, layout of the file system
  - File descriptors
    - Info describes file and where actual data is on disk
  - Free space descriptors
    - List of blocks unused that can be allocated to files
  - File name descriptors
    - Data structures regarding names with each file
- DOS FAT
  - First block is bootstrap
  - Then file allocation table; FAT is used as free list and to track which blocks have been allocated
  - Rest of volume is data clusters

49.2 Boot block BIOS Parameter Block and FDISK Table

- Boot Block
  - Begins with branch instruction
  - Followed by volume description (BIOS Parameter block)
  - Followed by real bootstrap code
  - Followed by optional disk partitioning table (FDISK table)
  - Followed by signature (error checking)
- BIOS Parameter Block
  - Describes device geometry like number of bytes per sector, sectors per track, tracks per cylinder, sectors per volume
  - Describes how file system is layed out on the volume, like number of sectors per cluster, number of reserved sectors, number of alternate file alloc tables, number of entries in root directory
- FDISK Table
  - Partition disk into logical sub-disks
  - Each entry describes a disk partition
  - Master Boot Record (MBR) is the first sector of the disk and includes the FDISK table and a bootstrap to find the active partition
    - Which then reads the partition boot record

49.3 File Descriptors (directories)

- Directory entry
  - Combine file description and file name into single file descriptor
  - Has things like name, file attributes like file/subdir, hidden, readonly, pointer to first logical block, file length

49.4 Links and Free Space (File Allocation Table)

- Allocate space to files in logical multi-block clusters
  - File descriptor entry contains pointer to first cluster of that file; entry for that next cluster tells us cluster number for next

- Entire FAT in memory, so successive block numbers can be looked up w/o need to do disk IO
- If next block pointer == -1, then is end of file
- If next block pointer == 0, then this cluster is free

– Garbage Collection
- Early DOS did not free blocks when file was deleted, just crossed out first byte and eventually run out of memory
- Eventually would initiate garbage collection, starting from root directory and marking nonvalid entries as free

49.5 Descendents of the DOS file system

– Longer file names
- Users' biggest complaints were the short filenames
- Used primary and auxiliary entry; primary is backwards compatible and auxiliary holds extra letters of name and stuff

– Alternate/back-up FAT
- FAT corruption is catastrophic; so copy FAT to pre-reserved location, so will only lose new files rather than all files

– ISO 9660 for CDs
- Written once and contiguously, so lots of DOS features are irrelevant
- Tree structured directory hierarchies
- Dir entries contain file name, type, location fo first block, number of bytes, time/date
- Variable length fields (learned from DOS)
- Variable length extended attributes section as well

49.6 Summary

– DOS is simple
- Doesn't have fancy linking stuff or multi user access control
- Doesn't take much space and just uses FAT isntead of free bock lists or file block pointers
- DOS is primitive and more or less works; but has lots of cludgy, upwards compatible upgrades

50 L14: Arpaci-Dusseau Chapter 41: Locality and The Fast File System

50.1 The Problem: Poor Performance

- Ken Thompson
  - UNIX Wizard writes first file system which has superblock, inodes, and data blocks
  - Solid step forward from record-based storage system and one-level hierarchies
- Performance
  - Really terrible because it treated disk like random-access memory, so lots of seektime and external + internal fragmentation
- Fast File System (FFS) is disk aware (Berkeley)
  - Took interface but changed implementation
  - Cylinder groups are groups of sets of tracks on different surfaces on hard drive
  - Holds superblock, inode itmap, data bitmap, inodes, and data blocks for each cylinder group
- Policies: How To Allocate Files and Directories
  - Keep related stuff together
  - Put directory data and inodes in cylinder groups with few allocated directories (balance directories across groups)
  - Allocate data blocks of file in same group as its inode
  - When measured, FFS does better than random, but still misses many types of locality

50.2 Large File Exception

- Large File
  - Don't put it in group because it'll fill the entire group, which is bad for future files
  - After direct pointers are used up, FFS puts other blocks in other block groups
- Amortization
  - Reduce overhead by doing more work per overhead paid
  - Basically, transfer a lot of data if you are going to physically seek

50.3 A Few Other Things About FFS

- Subblocks
  - Worried about internal fragmentation because 4KB blocks
  - So use subblock of 512 bytes, and allocate these blocks until the file grows to 4KB, then copy it over into a 4KB block and free the subblocks
  - Lots of IO and expensive
- Modified libc
  - Buffered writes prevent need for subblocks in most cases
- Parameterization
  - When reading sequentially, read block 1, then block 2, but by time issued read for block 2, it rotated away
  - So Sttager the layout scheme to avoid extra rotations
- Track Buffer
  - Read the entire track, and just buffer that shit
- Long file names and atomic rename()
  - FFS allowed for it which made it more user-friendly
- Symbolic Link
  - FFS allowed for you to create alias to any other foile or directory on system (more flexible than hard link, which cannot point to directories)

51 L14: Arpaci-Dusseau Chapter 42: Crash Consistency: FSCK and Journaling

51.1 Example

- File system data structures must persist despite presence of power loss or system crash
  - Crash-consistency problem where two on-disk structures, and only one gets updated before crash
- Problems of appending block to a file
  - Need to update inode, datablock, and bitmap, but can't update all three atomically
  - Variety of problems can occur from this

51.2 File System Checker

- fsck is a Unix tool for finding inconsistencies in metadata and repairing them
  - Checks if superblock looks reasonable (correct file system size), if corrupt, allow admin to use alternate copy
  - Scans inodes, indirect blocks, double indirects, etc; compares this against allocation bitmaps
  - Checks inodes for corruption
  - Verifies inode link counts by scanning directory tree and counting
  - Checks duplicate pointers where two inodes refer to same block
  - Checks for bad blocks (if it points out of valid range)
  - Checks integrity of directory, ensuring '.' and '..' are there, that each inode referred to in the dir entries is allocated, and that no directory is linked more than once in the hierarchy
- fsck is very slow and kind of irrational

51.3 Journaling (Write-Ahead Logging)

- Journaling is the file systems version of write-ahead logging
  - Write note in well-known place of what you are about to do
  - Have transaction identifier (TID)
  - Transaction begin (TxB) tells about update (final addresses, TID)
  - Transaction end (TxE) marks end of transaction
- Physical logging
  - Put exact physical contents of update in journal
- Logical logging
  - Put logical representation of update in journal, like "append data block Db to file X"
  - Maybe better performance
- Checkpointing
  - Once transaction is safely on disk, can overwrite old structures in file system
- Journal write and then checkpoint
  - Write transaction to log; wait for them to complete; write pending updates to final locations in file system
- Writing to journal
  - Don't write all five blocks (TxB, inode, bitmap, datablock, TxE) at once, because not safe, in case the Db (arbitrary data) is missed, we cannot know
  - So write the first four, then TxE once they have completed
  - So actually, it is journal write, journal commit, checkpoint
- Recovery
  - If crashes before transaction commit, then just skip update
  - If crash after commit, then replay transactions (redo logging) to ensure on-disk structures are consistent
- Batching Log Updates
  - Do not commit each update to disk one at a time
  - Buffer all updates into a global transaction
- Making The Log Finite

    - Large log causes longer recovery
    - If log is full, file system becomes useless
    - So use a circular log and free space once checkpointed
  - Journal superblock
    - Mark in journal superblock that the transaction has been checkpointed
    - Reduces recovery time and enables re-use of log in circular fashion
  - The above strategy is full data journaling
  - Metadata Journaling
    - Data write to final location, Journal metadata write to log, Journal commit to log and wait for it to complete, checkpoint metadata to final location, free transaction in journal superblock
  - Tricky Case: Bock Reuse
    - Stephen Tweedie's Nightmare for metadata journaling
    - Create a file in a directory
    - Delete directory
    - Create new file where the directory used to be
    - If crash, then recovery will overwrite with metadata from the old file/directory
    - Can solve this by never reusing block until checkpointed, or using a revoke record to never replay revoked data
  - Other approaches
    - Copy-on-write
      - Never overwrite files or directories, place new updates to previously unused locations on disk
    - Backpointer-based consistency
    - Many others

52 L14: Arpaci-Dusseau Chapter 43: Log-structured File Systems

52.1 Writing To Disk Sequentially

  - Motivation
    - System memories are growing, performance determined by writes
    - Large gap between random IO performance and sequential IO
    - Existing file systems perform poorly on common workloads, like many short seeks for many writes
    - File systems are not RAID aware, so logical write may cause many physical IOs
  - Log-structured File System
    - Buffers all updates in an in-memory segment, writing only when segment is full
    - Always writes to free locations
  - Write Buffering
    - Keeps track of updates in memory; writes them all to disk at once
    - Segment is a large chunk of updates
  - How Much To Buffer?
    - Want to amortize cost and get closer to achieving peak bandwidth
    - Use formula to decide how much to buffer based on how close to peak bandwidth

52.2 Problem: Finding Inodes

  - Old UNIX file system
    - Calculate exact disk address by multiplying inode number by size of inode and adding the start of the array
  - FFS
    - Splits up inode table into chunks and places a group of inodes within each cylinder group
    - So just need to know how much each chunk of inodes is, and start addresses
  - LFS
    - Inodes are scattered throughout disk, so more complex

52.3 Solution Through Indirection: The Inode Map

- Inode map takes inode number and produces disk address of most recent version
- LFS places chunks of inode map next to where its writing all other information
- Checkpoint region
  - Pointers to latest pieces of inode map, so find inode map by reading CR
- Reading A File From Disk: A Recap
  - Read checkpoint region
  - Use checkpoint regions pointers to find entire inode map
  - Then when given innode number of a file, simply look up inode-number to inode-disk-address mapping in imap and read most recent version of inode

52.4 What About Directories?

- Directory
  - Just collection of (name, inode number) mappings
  - Look in inode map to find location of directory's inode
  - Find location of directory data from dir inode
- Recursive update problem
  - Whenever inode updated, location on disk changes, which may have mandated change in direcotry which pointst ot this

52.5 A New Problem: Garbage Collection

- Periodically clean old versions of file data, inodes, structures
  - Compact M existing live segments into N new segments, and write N segments to new locations
  - Because we don't want to just have little blocks laying around
- Segment Summary Block
  - Has inode number (which file) and offset (block of file) for each data block
  - Compare info in segment summary block with info in inode to see if a block is alive
- When to clean?
  - Original approach was between hot and cold
  - Hot if frequently overwrriten
  - Cold if mostly stable
  - Clean the cold segments sooner and hot later

52.6 Crash Recovery and The Log

- LFS has two checkpoint regions
  - Detect system crash during CR update because of inconsistent timestamps
  - Read CR to recover
- Roll forward
  - Start with last checkpoint region, find end of log, use that to read through next segments to find valid updates

53 L14: Arpaci-Dusseau Chapter 45: Data Integrity and Protection

53.1 Disk Failure Modes

- Latent sector errors (LSE)
  - Head crash is when disk head touches surface
  - Cosmic rays can flip bits
  - Error correcting codes (ECC) are used by drive to detect errors/sometimes fix them
  - Silent Faults are when disk give no indication of problem, despite returning faulty data

    - Fail-partial (instead of fail-stop) is when disk works despite some blocks inaccessible/corrupt
  - Should try to detect both LSE and block corruption, because they both occur somewhat frequently

53.2 Handling Latent Sector Errors

- If mirrored RAID or RAID 4/5, just reconstruct
- Or add extra redundancy, like through RAID-DP's second parity disk, to help reconstruct during entire disk failure and LSE

53.3 Detecting Corruption: The Checksum

- Detection of corruption is difficult
- Checksum is a function peprformed on a chunk of data which produces something
- Common Checksums
  - XOR
  - Addition
  - Fletcher checksum; $s1+ = d_i; s2+ = s1 \forall d_i$
  - Cyclic redundancy check (CRC); treat region as a binary number, divide it by some constant value; use the remainder
- No perfect checksum
  - Because going from big to small
  - Just want to minimize collisions and be easy to compute
- Checksum layout
  - 520 bytes for data sector, 8 bytes for checksum
  - Maybe put all checksums next to sector, or put them at front
- Using checksums
  - Compute and compare against stored; if corrupted, use backup or give up
- Overhead
  - Space overheads are relatively small
  - Time overheads can be noticeable
  - Sometimes, extra IO overhead will occur

53.4 A New Problem: Misdirected Writes

- Misdirected write is when data is written to the wrong location
  - Add physical identifier (like block number and disk) and compare it when reading

53.5 One Last Problem: Lost Writes

- Write doesn't actually occur even though it was supposed to
  - Use write-verify or read-after-write
  - Immediately read daa after write and compare

53.6 Scrubbing

- Disk scrubbing
  - Periodically read through every block of the system and verify checksums

54 L15: Security for Operating Systems

54.1 Introduction

- Important because so widely used
  - Few OS choices and entirety of security is built upon secure OS
  - OS is big and complex, has control of all hardware, can do anything it wants, processes cannot protect self against OS
  -

54.2 Security Goals and Policies

- Confidentiality
  - If not supposed to be known by others, don't tell anyone it
- Integrity
  - If not supposed to be changed, don't let anyone change it
- Availability
  - If something is supposed to be available, don't let anyone prevent its use
- Non-repudiation
  - If someone told us something, cannot later deny that they did
- Security vs Fault tolerance
  - Don't want to totally isolate processes from each other, and processes share OS resources

54.3 Designing Secure Systems

- Economy of mechanism
  - Keep system small and simple because it'll have fewer bugs and be easier to understand
- Fail safe defaults
  - Set the default policy to be more secure, not less
- Complete mediation
  - Check if action meets security policies every time the action is taken
- Open design
  - Assume adversary knows design, and still achieve goals (they usually do know)
- Separation of privilege
  - Require separate parties/credentials to perform critical actions (two-factor auth)
- Least privilege
  - Give minimum privileges necessary (bc adversary maybe leverage friends error)
- Least common mechanism
  - Separate data structures for different users/processes (page table)
- Acceptability
  - Users must be willing to use it

54.4 The Basics of OS Security

- Controlling processes access to hardware
  - Hardware is shared bt all processes, so don't let one process interfere with another
- Weak links
  - Spend time securing weakest link (typically, people)
- System calls
  - OS will use PID and access control mechanisms to determine if process is authorized

55 L15: Authentication

55.1 Introduction

- Context
  - Who is asking is relevant
- Pricipal is security-meaningful entity, like human or some softwares
  - Process or entity performing request on behalf of principal is called the agent
- Credential is any data created/managed by OS to track access decisions for future reference
- Permission privileges based on identity are very important
  - Need way to attach identity to process

55.2 Attaching Identities to Processes and Authentication

- Inherit identity usually
  - OS's user login will assign new identity
  - OS must be able to qury user identity info
- Authentication
  - Based on what you know, what you have, what you are
- Authentication by what you know
  - Typically password which nobody knows/can guess except the true user
  - Hash password using cryptographic hash
  - Attackers often use dictionary attack, just list of specialized, common strings
  - Password vaults are encrypted files on your computer that store passwords and have their own password
  - Salt is a big random number concatenated to password, then you hash this and store both the result and random number
  - Passwords are kind of outdated; best used as part of multi-factor auth
  - Don't indicate whether username or password is incorrect
- Authentication by what you have
  - Like ID or ticket
  - Compares username/password and don't tell which is wrong
  - Or hard to learn information (string on screen, changing numbers, cellphone text)
- Authentication by what you are
  - Image of user needs to not require perfect match
  - Close enough, for auth
  - False negative if incorrectly decided not to authenticate
  - False positive if incorrectly decided to authenticate
  - Crossover error rate is where the two curves cross
  - Biometrics are okay sometimes if you have the hardware and want convenience over higher security
  - But biometrics over network is dangerous
  - Biometric of typing pattern is also feasible
- Authentication by where they are
  - Like person at DMV

55.3 Authenticating Non-Humans

- Identity or password
  - Also can change identity if admin
  - Or if part of a group

56 L15: Access Control

56.1 Important Aspects of the Access Control Problem

- Figure out if request fits within security policy
  - If so, perform operation; otherwise, don't
- Authorization
  - Determining of particular subject is allowed to perform something
- Reference Monitor
  - Code that decides when access control decisions will be made
- Access Control Lists
  - Just list to know who to let in
  - 9 bits, RWX for owner, group, everyone else
  - Easy to figure out who has access
  - Revocation of privilege is easy
  - Easy to find the information because its with the file metadata
  - Hard to determine entire set of resources a principal (process or user) can access
  - Also, need common identity across all machines in distributed environment
- Capabilities (Hydra OS)
  - Like a key, just give it to whoever needs
  - Need to be able to revoke keys, and need to hand to correct person
  - Capability is just a bunch of bits, long and complex
  - So people could just generate the capabilities, should be uncopyable
  - Deal with this by not letting process actually access the capability, OS does it all and encrypts the shit if distributed
  - Having giant list would be expensive (because so many files)
  - Easy to give subsets of capabilities to child processes
- Namespaces
  - Names across distributed environments is hard
  - Could have central approval, or could just do it with namespaces, where participants get to choose anything from some portion
- Use control lists, and capabilities under the covers
  - When call open(), ACL is not examined on subsequent writes
  - Linux creates data structure like a capability and attach it to the process's PCB that is checked on subsequent read/writes
  - When file is closes, capability is deleted, and so needs to go through ACL to open again
  - Do similar things for hardware devices, IPC channels because they are treated like files
  - This solution avoids costs of checking ACL on every operation (capability easy to check)
  - Managing capabilies is avoided because only setup if ACL successful
  - Scaling not an issue because few files opened, typically

56.2 Mandatory and Discretionary Access Control

- Mandatory Access Control
  - Can override desires of owner of the information
- Discretionary Access Control
  - Almost anyone/no one is given access to a given resource at discretion of owning user

56.3 Practicalities of Access Control Mechanisms

- Default access permissions
  - Can be changed, but users rarely do this
- Android
  - Many small independent apps, which should not have the same privileges as the user himself

    - So use permission labels as mandatory capabilities to exercise the principle of least privilege
    - Not perfect though, because many users don't know what they are granting and will grant it anyways because they want the app to work
- Role-Based Access Control
    - Create roles, and assign privileges to that role; let users assume and change roles;
    - Users choose from a restricted set of roles
- Type Enforcement
    - Finer granularity than just read/write, can be permitted to add a particular type of record, etc
    - Associates detailed access rules to particular objects using security context
- Privilege escalation
    - Allows caeful extension of privileges by allowing program to run with privileges beyond thos eof user who invokes them
    - SetUID lest you run with permissions of another user
- Linux sudo
    - Allow designated users to run certain programs under another identity

57 L15: Cryptography

57.1 Introduction

- If others gain access to critical data outside control of OS
    - Don't want them to be able to use it, so alter it
    - Use cryptography to alter data in controlled ways
    - Is expensive and must be done correctly to be useful
- Cryptography
    - Cipher is algorithm used on data
    - Key is second piece of info to augment
    - Ciphertext is altered data
    - Computationally infeasible to to extract P without K, encryption/decryption algs are probably known by everyone
    - Cryptography relies entirely on secrecy of key
- Can tell if someone tampered with data by including and comparing hash of plaintext
- Symmetric Cryptography
    - Use same key to encrypt and decrypt

57.2 Public Key Cryptography

- Public and private key
    - Public key known to everyone and used to decrypt
    - Private key is secret and used to encrypt
    - Can use decryption key to encrypt, then use encryption key to decrypt
        - Doesn't work for authentiation anymore though, because people know the key used to encrypt
    - Authentication if encrypt with private key
    - Secret communication if encrypt with public key
    - Hundreds of times more expensive than symmetric
- Authentication and secret
    - Two pairs of public and private keys
    - Alice encrypts msg with her private key, then encrypts msg with Bob's public key
    - Only Bob can decrypt it with his private key, then confirm it is Alice using her public key (only Alice could have created that msg)
- Key distribution
    - Originally, everyone only knows their own public key

57.3 Cryptographic Hashes

- Cryptographic hash ensures integrity
  - Hash reduces size
  - Hash a piece of data, encrypt only hash, send encrypted hash and data to partner
  - If opponent tampers, when decrypt hash and repeat hash on data, will detect mismatch
  - Want to make it infeasible for opponent to find any other plaintext which hashes to the same thing
  - Encrypt hash because anyone can just replace msg and hash, but only one person knows proper encrypted version
- Cryptographic hashes for other uses
  - Used for storing salted passwords
  - Proof-of-work for spam prevention and blockchain

57.4 Cracking Cryptography

- Cracking
  - Usually, crack it by obtaining keys or through flaws in your management of cryptography
  - Improper distribution of secret keys, choosing keys from reduced set of choices, etc
- Heartbleed
  - Obtain keys and decrypt entire session
- Key Selection
  - Must select keys randomly, so that opponent can't just figure out how you created the keys
  - Perfect forward secrecy is when figuring out one key won't help opponent figure out other keys
- Gathering entropy
  - Read low-order bits from hardware processes
- PK uses long keys (2K or 4K bits)
  - Can use mathematical properties of system to derive private key from public key, so use very long keys to make this difficult

57.5 Cryptography and Operating Systems

- Must trust OS, or else no chance
- Hash passwords, encrypt data, and at-rest data encryption
- At-Rest Data Encryption
  - Never store decrypted data on disk
  - Data in memory is decrypted and is encrypted when written to disk
  - Few percent extra latency from overhead for IO intensive operations
  - Only provides protection if someone physically takes the storage device
    - Person will read useless bits
- Homomorphic cryptography
  - Allows you to perform operations on encrypted form of data without decrypting it
  - Not used bc high performance/storage costs
- Password vault
  - Encrypt all passwords and have a single password to decrypt them

57.6 Cryptographic Capabilities

- Encrypted data structure for capabilities
  - Could give this data structure to user, who would present it
  - Symmetric cryptographic capabilities make sense when all machines creating/checking are trusted and key distrib is easy
  - Assymetric if secrecy is required
  - Could hold info like expiration times, identity, etc

58 L16: Distributed Systems Goals and Challenges

58.1 Goals: Why Build Distributed Systems

- Internet
  - Collaborate with others
- Client/Server Usage Model
  - More functionality and more efficient if remote/centralized resources rather than all connected to computer
- Reliability and Availability
  - Distribute service over multiple independent servers, so no single point of failure
- Scalability
  - Don't replace computer every time
  - Build a system which can be incrementally expanded (scale-out)
- Flexibility
  - May decide we need to run different parts on different computers
  - Easy to change deployment model if components interact through network protocols

58.2 Challenges: Why are Distributed Systems Hard to Build

- New and More Modes of Failure
  - Whole system may die if one part of system fails
  - One node can crash while others continue running
  - Occasional network messages may be delayed/lost
  - Switch failure may interrupt communication between some nodes
  - In distributed system, hard to tell if a component has failed
    - May only detect this because no longer receiving messages
  - Need a robust system if we want to keep running despite failure of individual components
    - Error checking, breaking locks, restoring resources
- Complexity of Distributed State
  - Distinct nodes in distributed system operate independently, so hard to ensure correct serialization
  - Different nodes may consider a single resource to be in different states
- Complexity of Management
  - Many different configurations
  - Nodes may not be up when updates are sent
- Much Higher Loads
  - Build distributed systems to handle more loads, which lead to bottlenecks, messages, overhead, race conditions, etc
- Heterogeneity
  - May have different instruction set architecture, OS, version of software/protocols, networks, file systems, etc
- Emergent phenomena
  - Emergent behaviors that are not present in constituent components
- Peter Deutsch's "Seven Fallacies" of Distributed Computing
  - Network is reliable
  - Latency is zero
  - Bandwidth is infinite
  - Network is secure
  - Topology does not change (routes between clients/servers)
  - There is one admin
  - Transport cost is zero
  - Network is homogenous (extra 8th)

59 L16: Arpaci-Dusseau: Chapter 48: Distributed Systems

59.1 Communication Basics

- Failure as a challenge
  - Many things fail, hard to build perfect things, so use distributed system
- Performance, security, and communication are all important
- Communication Basics
  - Fundamentally unreliable
  - Lots of reasons for packet loss/corruption, like bits getting flipped, electricla problems, lack of buffering (packets must be dropped)

59.2 Unreliable Communication Layers

- UDP/IP
  - Uses sockets API to create communication endpoint and sends UDP datagrams, fixed-sized messages
  - Does not ensure packets reach destination, but includes checksum

59.3 Reliable Communication Layers

- Acknowledgement (Ack)
  - Receiver sends short acknowledgement msg back to sender
  - Timeout meachnism if sender doesn't receive in time
  - If timeout, will retry
  - Acknowledgement msg could get lost, so need to guarantee each message received exactly once
- Sequence Counter
  - Sender and Receiver both have starting value
  - Only send value to application if message number matches counter
  - If counter is higher than message number, then already received it, so acknowledge but don't send to application
- TCP/IP
  - Reliable communication layer
  - Has many other mechanisms to handle congestion, multiple outstanding requests, etc

59.4 Communication Abstractions

- Timeout Value
  - Exponential back-off, increase timeout value exponentially to avoid overlaoding the system with re-sends
  - Timeout value should ideally be just long enough to detect packet loss
- Distributed shared memory (DSM)
  - Enable processes on different machines to share large, virtual address space
  - Nobody uses anymore beacuse some accesses to memory are really slow

59.5 Remote Procedure Call (RPC)

- Stub Generator
  - Stub generator takes interface and generates client stub
  - Client stub code
    - Creates msg buffer, packs info, sends msg, waits for reply, unpacks/unmarshal/deserialize return code/args, return to caller
  - Server code
    - Unpack/unmarshal/deserialize msg, call actual function, package results, send reply
- Issue of more complex data structures
  - Deal with pointers and stuff by passing well-known types or annotating data structure
- Thread pool

    - Finite set of threads created
    - When msg arrives, dispatch to one of the worker threads to do work
    - Main thread can continue receiving
- Run-time Library
    - How to locate remote service - naming problem
    - Simple approach is to build off of hostnames and port numbers
    - Building on TCP may be inefficient, but UDP may be Unreliable
    - Build on UDP for performance, RPC then implements reliability using timeout/retry, acknowledgements, and sequencing
- If remote call takes a long time to complete, needs to not appear as a failure
    - Could acknowledge, then reply later
    - Client could also ask if still working, and get a reply
- Large arguments
    - Server-side fragmentation (breaking large packets into smaller ones)
    - Receiver-side reassembly (small to larger logical whole)
- Byte ordering
    - Big endian vs little endian problem, solved by welld-defined endianness in msg format
    - Big endian is most significant to least significant, like Arabic (little is opposite)
- End-to-end Argument
    - Corruption could occur before even sent, or before written to memory by receiver, etc
    - So typically compare checksums
- Synchronous or asynchronous
    - Client may have to wait a long time, so some RPC packages enable you to invoke request and return immediately to do other work
- gRPC and Apache Thrift
    - Two RPC packages

60 L16: REST-ful Interfaces

60.1 Communication Basics

- Representational State Transfer by Roy Fielding
    - Web services provide interoperability between computer system through the internet
    - Allow requests to access/manipulate textual representations through uniorm, predefined, stateless operations
    - Client-server architecture enables separation of concerns, improving scalability
    - Stateless operations mean that no client context is stored on server between requests; every requests has full information
    - Cacheability must be defined by responses to prevent reuse of stale data
    - Layered system improves scalability by enabling load balancing and providing shared caches

61 L16: Lease-Based Serialization

61.1 Challenges of Distributed Locking

- Zero latency, reliable delivery, stable topology, consistent management
    - Obtaining lock through messages may take 1mil x longer
    - Operations not guaranteed to complete, because requests/responses may be lost
    - When node crashes, locks may be forgotten/lost
    - There is no WAN-scale (wide area network) atomic instructions or interrupt disables
    - Set of resources and order are not known, so cannot use ordering to prevent deadlock

61.2 Addressing these Challenges

- Replace leases with locks
  - Lease grants owner exclusive access to resource until owner releases or lease duration expires
  - Locks work on honor system; leases are enforced, because request will be rejected
- Recovering from failure of lease-holder
  - If failed midway through an update, then object state is inconsistent
  - Choice of lease period needs tuning (renew many times per period vs long time to recover)
- Don't send network msg for every entry into critical section
  - Leases should be used for longer-lived resources (DHCP allocated IP addresses) because amortization

61.3 Evaluating Leases

- Mutual Exclusion
  - At least as good as locks; also good bc potential for enforcement (unlike locks)
- Fairness
  - Depends on policies of remote lock manager
- Performance
  - RPC to get lease is very expensive, but okay if we keep for many operations
- Progress
  - Automatic preemption creates deadlock immunity
  - Lessees have to wait for lease period to expire if lease-holder dies
- Robustness
  - More robust than locks
- Problems
  - Very complex to recover from lock-server failures
  - Hard to determine time for automatic lease expiration
- Opportunistic Locks
  - Can ask for a long-term lease, enabling future locking to be purely local operation
  - Lock manager revokes if necessary

62 L16: Lease-Based Serialization

62.1 Communication Basics

- Consensus requires agreement among multiple processes for some data value
  - Some processes may fail or be unreliable, so needs fault tolerance
  - Satisfy termination, validity, integrity, agreement to be fault tolerant
- Termination
  - Every correct process decides some value
- Validity
  - If all processes propose v, correct processes decide v
- Integrity
  - Corect process decides at most one value
- Agreement
  - Every correct process must agree on same value

63 L16: Distributed System Security

63.1 Introduction

- OS can only control itse own machine's resources
  - Other machines in distributed system might not be secure, but still need to trust their credentials and capabilities

- Adversaries can alter our communication across network

63.2 The Role of Authentication

- Cannot really know
  - Cannot know that partner in system is using correct security policies
  - Can only detect when they don't and then take compensating action
- Authenticate to know who you are truly speaking to over network
  - Rely on cryptohashed salted password or private/public key to authenticate
  - Use PK to authenticate website to user (bc only need to give one public key to everyone)
  - Use passwords to authenticate user to website (bc easier; alternative is to distribute user public keys to website)

63.3 Public Key Authentication for Distributed Systems

- Certificate Authority
  - Company will confirm that a particular company has a particular public key
  - Hashes the company name, public key, and some info
  - Then encrypt this hash with CA's private key, producing digital signature
  - Then combine this info with CA's info into a certificate
  - Give this certificate to company for money
- If user needs company's public key
  - User will examine certificate, regenerate the hash, and compare
  - If no problems, then continue
  - Good bc user keeps company untrusted until certificate is valid
- Certificate authority's public key
  - Comes with software like browsers/etc
- Anyone can create certificate
  - Can make your own
- Replay attacks
  - If public key decrypts someone's private msg, then it indeed came from that person, but risk for replay attack
  - Problem where someone captures msg, then sends it at a later time
  - Fix this by having unique information in each message

63.4 Password Authentication for Distributed Systems

- Good for users authenticating to server
  - Not good for servers authenticating to user
- User provide user ID and password
  - Encrypt both before sending over network
- Use authentication to establish initial authenticity
  - Then use something else for subsequent interactions

63.5 SSL/TLS

- SSL is before TLS
  - Secure Socket Layer came before Transport Layer Security
  - TLS is better, don't use SSL
  - People refer to SSL and TLS as the same thing
- SSL
  - Move encrypted data through ordinary socket
  - Setup new key to encrypt bulk of data for each connection, even if same partner
  - Decide on set of ciphers, techniques, hashes, etc

    - Client usually obtains certificate with server's public key, then use public key in certificate to verify authenticity
    - This way of sending certificate is just as insecure as others
    - Once have certificate, can proceed with Diffie-Hellman key exchange
        - Server signs messages with private key, client does not do the same though
        - Alice and Bob agree upon a large prime number (n) and a primitive mod n (g)
        - Both Alice And Bob generate random numbers x and y
        - Then compute $X = g^x mod n$ and $Y = g^y mod n$
        - Then they compute $k = Y^x mod n = X^y mod n$, which will be equal
        - Guarantee securely shrae a key with someone, but not sure who
- SSL/TLS is a protocol
    - Software implementations may have bugs

63.6 Other Authentication Approaches

- Web cookies
    - Verify identity once and rely on it for future for convenience
    - Various security issues, like user storing cookie, etc
- Challenge/Response
    - Hardware token or data secret must be setup and distributed before challenge is issued
    - Server sends challenge to client, client is only one who can compute response based on some secret knowledge
    - Probably will crypto hash the challenge and response
- Authentication Server
    - Everyone trusts server, rely on server to authenticate identity
    - Kerberos is an example of an online version
    - Certificate authorities are like an offline example of this

63.7 Higher Level Tools

- HTTPS
    - Cryptographically protected version of HTTP
    - Connects SSL/TLS to existing definition of HTTP
    - SSL takes care of establishing secure connection, authenticating using certificate, establishing new symmetric encryption
    - Certificates are gathered and managed by browser
    - Designed to secure web browsing at a low development cost
- SSH
    - Secure Shell was originally intended to be used as a secure remote shell, but now extended into general tool for secure interactions between computers
    - Can be used for X window sessions, TCP ports, etc
    - Users must still be authenticated, shared encryption keys established, integrity checked
    - Relies on public key cryptography and certificates
- Man-in-the-middle
    - Two parties think they are communicating directly, but a malicious third party is in the middle, sees everything, can inject
    - When setting up SSH with remote machine you haven't logged into before, you get a warning asking to trust the public key
    - Everyone just says yes, so maybe this warning not that good, but still there
- SSH not build on SSL, separate implementation

63.8 Summary

- Distributed systems are critical to modern computing, but difficult to secure
  - Need to ensure that insecure networks don't introduce new security problems
  - Messages sent are encrypted and authenticated, protecting privacy and integrity
- SSL/TLS, public keys, and certificates help provide these services
- Passwords help authenticate remote human users
- Symmetric cryptography used to transport most data, because cheaper than assymetric cryptography
- Symmetric keys not shared by system participants
  - Exchange symmetric key as first step
- Key secrecy is eessential
- Diffie Hellman key exchange is used, but still requires authentication that only intended participants know the key
- Do not build your own cryptographic solution

64 L16: Leases

64.1 Communication Basics

–