Operating Systems Principles
UCLA-CS111-W18

Quentin Truong
Taught by Professor Reiher

Winter 2018

Contents

1 L3: Arpaci-Dusseau Chapter 5: Interlude: Process API . . . 2
1.1 The fork() System Call . . . 2
1.2 The wait() System Call . . . 2
1.3 The exec() System Call . . . 2
1.4 Why? Motivating The API . . . 2
1.5 Other Parts Of The API . . . 2

2 L3: Arpaci-Dusseau Chapter 6: Mechanism: Limited Direct Execution . . . 2
2.1 Basic Technique: Limited Direct Execution . . . 2
2.2 Problem 1: Restricted Operations . . . 3
2.3 Problem 2: Switching Between Processes . . . 3
2.4 Worried About Concurrency? . . . 4
2.5 Summary . . . 4

3 L3: Linking and Libraries: Object Modules, Linkage Editing, Libraries . . . 4
3.1 Introduction . . . 4
3.2 The Software Generation Tool Chain . . . 4
3.3 Object Modules . . . 4
3.4 Libraries . . . 5
3.5 Linkage Editing . . . 5
3.6 Load Modules . . . 5
3.7 Static vs. Shared Libraries . . . 5
3.8 Dynamically Loaded Libraries . . . 5

4 L3: Linkage Conventions: Stack Frames and Linkage Conventions . . . 5
4.1 Introduction . . . 5
4.2 The Stack Model of Programming Languages . . . 5
4.3 Subroutine Linkage Conventions . . . 6
4.4 Traps and Interrupts . . . 6

5 L4: Arpaci-Dusseau Chapter 7: Scheduling: Introduction . . . 7
5.1 Workload Assumptions . . . 7
5.2 Scheduling Metrics . . . 7
5.3 First In, First Out (FIFO) . . . 7
5.4 Shortest Job First (SJF) . . . 7
5.5 Shortest Time-to-Completion First (STCF) . . . 7
5.6 A New Metric: Response Time . . . 7
5.7 Round Robin . . . 7
5.8 Incorporating I/O . . . 8

5.9 No More Oracle/Summary . . . 8

6 L4: Arpaci-Dusseau Chapter 8: Scheduling: The Multi-Level Feedback Queue 8
6.1 MLFQ: Basic Rules . . . 8
6.2 Attempt 1: How To Change Priority . . . 8
6.3 Attempt 2: The Priority Boost . . . 8
6.4 Attempt 3: Better Accounting . . . 8
6.5 Tuning MLFQ And Other Issues . . . 8
6.6 MLFQ: Summary . . . 9

7 L4: Real Time Scheduling 9
7.1 What are Real-Time Systems . . . 9
7.2 Real-Time Scheduling Algorithms . . . 9
7.3 Real-Time and Linux . . . 9

8 L5: Arpaci-Dusseau Chapter 12: A Dialogue on Memory Virtualization 10
8.1 Overview . . . 10

9 L5: Arpaci-Dusseau Chapter 13: The Abstraction: Address Spaces 10
9.1 Early Systems . . . 10
9.2 Multiprogramming and Time Sharing . . . 10
9.3 The Address Space . . . 10
9.4 Goals . . . 10

10 L5: Arpaci-Dusseau Chapter 14: Interlude: Memory API 10
10.1 Types of Memory . . . 10
10.2 The malloc()/free() Call . . . 10
10.3 Common Errors . . . 11
10.4 Underlying OS Support . . . 11

11 L5: Arpaci-Dusseau Chapter 17: Free-Space Management 11
11.1 Assumptions . . . 11
11.2 Low-level Mechanisms . . . 11
11.3 Basic Strategies . . . 11
11.4 Other Approaches . . . 12

12 L5: Garbage Collection and Defragmentation 12
12.1 Garbage Collection . . . 12
12.2 Defragmentation . . . 12

13 L6: Arpaci-Dusseau Chapter 18: Paging: Introduction 13
13.1 A Simple Example And Overview . . . 13
13.2 Where Are Page Tables Stored? . . . 13
13.3 Whats Actually In The Page Table? . . . 13
13.4 Paging: Also Too Slow . . . 13

14 L6: Arpaci-Dusseau Chapter 19: Paging: Faster Translations (TLBs) 13
14.1 TLB Basic Algorithm . . . 13
14.2 Example: Accessing An Array . . . 14
14.3 Who Handles The TLB Miss? . . . 14
14.4 TLB Contents: Whats In There? . . . 14
14.5 TLB Issue: Context Switches . . . 14
14.6 Issue: Replacement Policy . . . 14

15 L6: Arpaci-Dusseau Chapter 21: Beyond Physical Memory: Mechanisms 14
15.1 Swap Space . . . 14
15.2 The Present Bit . . . 15
15.3 The Page Fault . . . 15
15.4 What If Memory Is Full? . . . 15
15.5 Page Fault Control Flow . . . 15
15.6 When Replacements Really Occur . . . 15

16 L6: Arpaci-Dusseau Chapter 22: Beyond Physical Memory: Policies 15
16.1 Cache Management . . . 15
16.2 The Optimal Replacement Policy . . . 16
16.3 Replacement Policies . . . 16
16.4 Implementing LRU . . . 16
16.5 Considering Dirty Pages . . . 16
16.6 Other VM Policies . . . 16
16.7 Thrashing . . . 16

17 L6: Working Sets 17
17.1 LRU is not enough . . . 17
17.2 The concept of a Working Set . . . 17
17.3 Implementing Working Set replacement . . . 17
17.4 Dynamic Equilibrium to the rescue . . . 17

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

19 L7: Arpaci-Dusseau Chapter 26: Concurrency: An Introduction 18
19.1 Introduction . . . 18
19.2 Why Use Threads? . . . 18
19.3 An Example: Thread Creation . . . 18
19.4 The Heart Of The Problem: Uncontrolled Scheduling . . . 18
19.5 The Wish For Atomicity . . . 18

20 L7: Arpaci-Dusseau Chapter 27: Interlude: Thread API 19
20.1 Threads . . . 19

21 L7: User-Mode Thread Implementation 19
21.1 Introduction . . . 19
21.2 User/Kernel . . . 19
21.3 User/Kernel . . . 19

22 L7: Inter-Process Communication 19
22.1 Introduction . . . 19
22.2 Simple Uni-Directional Byte Streams . . . 20
22.3 Named Pipes and Mailboxes . . . 20
22.4 General Network Connections . . . 20
22.5 Shared Memory . . . 20
22.6 Network Connections and Out-of-Band Signals . . . 20

23 L7: Named pipes, Send, Recv, Mmap 21
23.1 Named Pipes . . . 21
23.2 Send, Recv, Mmap . . . 21

24 L8: Arpaci-Dussseau Chapter 28: Locks **22**
24.1 Locks: The Basic Idea . . . 22
24.2 Pthread Locks . . . 22
24.3 Evaluating Locks . . . 22
24.4 Controlling Interrupts . . . 22
24.5 A Failed Attempt: Just Using Loads/Stores . . . 22
24.6 Building Working Spin Locks with Test-And-Set . . . 22
24.7 Compare-And-Swap . . . 23
24.8 Load-Linked and Store-Conditional . . . 23
24.9 Fetch-And-Add . . . 23
24.10A Simple Approach: Just Yield, Baby . . . 23
24.11Using Queues: Sleeping Instead Of Spinning . . . 23
24.12Two-Phase Locks . . . 23

25 L8: Arpaci-Dusseau Chapter 30.1: Condition Variables **23**
25.1 Definition and Routines . . . 23

26 L9: Arpaci-Dusseau Chapter 29: Lock-based Concurrent Data Structures **24**
26.1 Concurrent Counters . . . 24
26.2 Concurrent Linked Lists/Queues/Hash tables . . . 24

27 L9: Arpaci-Dusseau Chapter 30.2+: Condition Variables **24**
27.1 The Producer/Consumer (Bounded Buffer) Problem . . . 24
27.2 Covering Conditions . . . 24

28 L9: Arpaci-Dusseau Chapter 31: Semaphores **25**
28.1 Semaphores: A Definition . . . 25
28.2 Binary Semaphores (Locks) . . . 25
28.3 Semaphores For Ordering . . . 25
28.4 The Producer/Consumer (Bounded Buffer) Problem . . . 25
28.5 Reader-Writer Locks . . . 25
28.6 Dining Philosophers . . . 25

29 L9: Flock and Lockf **25**
29.1 Man . . . 25

30 L10: Arpaci-Dusseau Chapter 32: Common Concurrency Problems **27**
30.1 Non-Deadlock Bugs . . . 27
30.2 Deadlock Bugs . . . 27

31 L10: Deadlock avoidance **27**
31.1 Introduction . . . 27
31.2 Reservations . . . 28
31.3 Over-booking . . . 28

32 L10: Health Monitoring and Recovery **28**
32.1 Introduction + Health Monitoring . . . 28
32.2 Managed Recovery . . . 28
32.3 False Reports and Other Restarts . . . 28

33 L10: Java Synchronized Methods **29**
33.1 Synchronized Methods . . . 29

34 L10: Java Intrinsic Locks + Synchronization **29**
34.1 Intrinsic Locks and Synchronization 29
34.2 Synchronized Statements 29

35 L10: Monitors **29**
35.1 Monitors 29

36 L10: Measuring Operating Systems Performance **29**
36.1 Metrics 29
36.2 Complexity and the Role of Statistics in Measurement 30
36.3 Workloads 30
36.4 Common Mistakes in Performance Measurements 30

37 L11: Load and Stress Testing **31**
37.1 Introduction 31
37.2 Load Testing 31
37.3 Stress Testing 31

38 L12: Arpaci-Dusseau Chapter 33-33.6: Event-based Concurrency (Advanced) **32**
38.1 The Basic Idea: An Event Loop 32
38.2 A Problem: Blocking System Calls 32
38.3 A Solution: Asynchronous I/O 32

39 L12: Arpaci-Dusseau, Chapter 35: A Dialogue on Persistence **32**
39.1 Dialogue 32

40 L12: Device Drivers: Classes and Services **32**
40.1 f 32

41 L12: A Dialogue on Persistence **32**
41.1 f 32

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

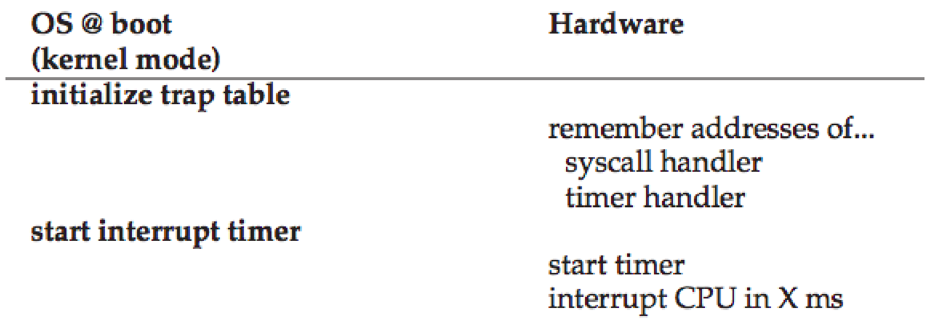
- Crux: How to efficiently virtualize CPU with control
- Limited Direct Execution
  - OS will create entry for process list, allocate memory for program, load program into memory, setup stack with argc/v, clear registers, execute call to main()
  - Program will run main(), execute return
  - OS will 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

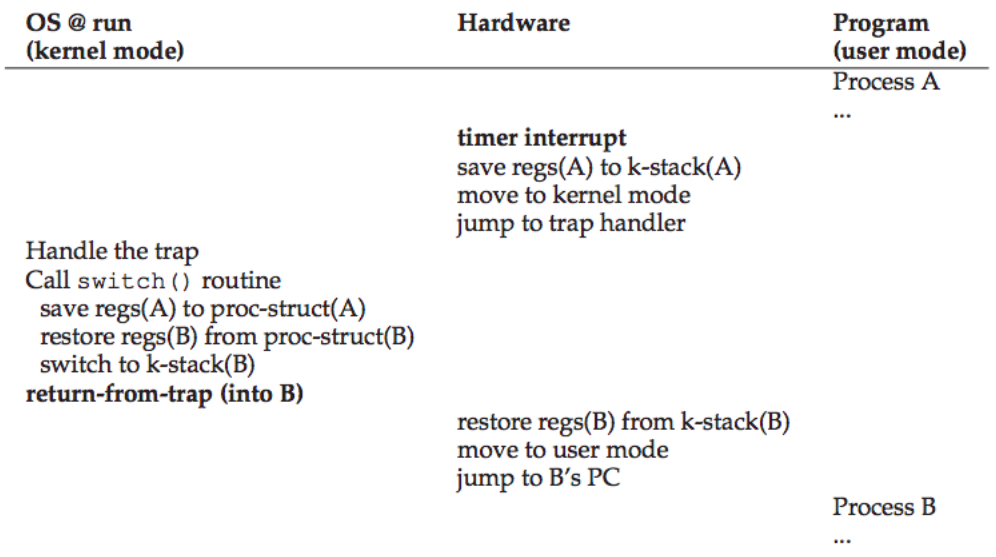
2.3 Problem 2: Switching Between Processes

– 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 f

–

41 L12: A Dialogue on Persistence

41.1 f

–