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 L5: 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: Arpaci-Dusseau Chapter 2

17.1 A

- –