CS 140 Final Exam Review
Administrative Details

Wednesday, December 9th, 3:30 - 6:30pm.
Like midterm: open notes, closed textbook, closed electronic devices

Total exam grade:
max(midterm > 0 ? final : 0, (midterm + final) / 2)

Exam will cover
  Pre-midterm topics
  Post-midterm topics
  Understanding of Pintos projects
Pre-Midterm Topics at a Glance
Pre-Midterm Review

Processes and Threads
Concurrency
Dispatching and Scheduling
Virtual Memory
Synchronization
Memory Allocation
Linking
Processes and Threads

• Kernel goal: Allow (safe) multitasking
  – Memory protection – keep each process independent, maintain kernel integrity
  – Resource sharing – Memory and CPU time
  – Keep programming model simple

• Tools:
  – Processes, light vs. heavyweight threads tradeoff
  – Fork/Wait system calls
  – Pipes for communication
Pre-Midterm Review

• Concurrency
  – Sequential Consistency, x86 consistency model
  – Atomic instructions, memory fences
  – Monitor style of locking (mutex + condition variables), Semaphores
  – Implementing locks – test_and_set, disabling interrupts

• Scheduling
  – Policy tradeoffs, average case vs pathological case
  – Priority scheduling, processor affinity
Pre-Midterm Review

• Virtual memory
  – Issues: Protection, Transparency, Resource exhaustion
  – Implementations: Load-time linking, Base+Bounds, Segmentation, Paging
  – Paging data structures and x86 paging, TLB
  – OS considerations: Eviction policies, working sets, thrashing

• Synchronization
  – Amdahl’s law – limits of possible speedup
  – C11 Atomics, MSI Cache Coherence
  – Wait-free synchronization and Transactional Memory
  – Deadlocks
PA3: Page Table Management

• Can you spot the issue here:

```c
struct spt_entry *spt = spt_lookup(uaddr);
if (spt == null) return false;
bool dirty = pagedir_is_dirty(spt);
if (dirty && spt->file) {
    write_to_fs(spt);
}
pagedir_clear_page(spt->uaddr);
// etc
```
Pre-Midterm Review

• Dynamic Storage Management
  – Heap Allocation
    • Approaches and tradeoffs: Best fit, First fit, LIFO
  – Garbage Collection
  – Reference Counting
Linking

• Overview: reference functions scattered across different source files and libraries

• Dynamic libraries

• Executable files
Linking Overview

• Simple 2-pass linker
• Pass 1
  – Coalesce sections (data, code, etc)
    • Must keep relative offsets from compiler intact
  – Arrange in memory
  – Read symbol tables to create a global table (functions/variables)
    • Name mangling: create unique names for overloaded functions
  – Compute virtual address of sections: start+offset
• Pass 2
  – Patch references using global symbol table (i.e. printf)
  – Emit result
Process Layout

- Kernel
- Stack
- Heap
- Data
- Code
Executable file

• How to get from a compiled program to a running one?
• Executable file: tells OS how/where load code and data

Header:

Object code: Instructions
ELF calls this .text

Exported Symbols: ELF calls this .sym

Relocations: external refs, ELF .text.rel

• Divided into segments
Loading

• Loader reads executable and places segments into memory
  – Optimizations: zero-initialized data, demand loading (PA3!), Copy-on-write sharing

• Who decides what goes where?
Loading, cont’d

• Who decides what goes where?
• Global code/data
  – Generated by compiler/linker, loaded into memory by loader
• Read-only data: Mapped by loader
• Stack: Mapped by kernel, expanded by program and kernel as needed (PA3)
• Heap: Handled by runtime allocator
Shared Libraries

• Dynamic Linking
  – Reduces size of binaries
  – No need to recompile to upgrade libraries

• Procedure Linkage Table/Global Offset Table
  – Loader uses indirection to speed up loading: no need to patch every library call
  – Lazy loading: dlfixup patches call on first use

• Security
  – Mark code as read-only, data as non-executable
  – Address space randomization: use different address each time so attacker doesn’t know values in advance
Post-Midterm Review
IO and Disks
IO Busses

• Data interconnect between different devices
• Comes in many varieties; use bridges or bus controllers to convert between them
Device Communication

• Port IO
  – Uses in/out instructions (x86) to read bytes from a given port number, or insw for bulk transfer
  – Older mechanism

• Memory-Mapped IO
  – Map certain sections of memory to control a device or transfer data

• Interrupts – signals sent to CPU (i.e. on completion of a block_read)
  – Polling vs interrupts tradeoffs

• Direct Memory Access
  – Allows devices to directly modify sections of memory
Magnetic Disks

• Time measures
  – Seek time: Move actuator arm over required track
  – Rotational latency: Wait for desired sector to pass under the head. (0.5 average disk rotation)
  – Transfer time: Read/Write data as it passes under the head

• Latency = Seek + Rotational (+ Transfer)

• Disk Scheduling: policies and tradeoffs
  – Elevator, SSTF, FCFS

• Modern day:
  – APIs hide track structure.
  – Inner tracks have fewer sectors than outer tracks
  – Bad sectors are automatically remapped.
Flash Memory

• No moving parts – different set of tradeoffs
  – No more seek time!

• Limited write durability

• Comes in different varieties
  – NAND – fast erase/write
  – NOR – faster read, slower erases
  – SLC vs MLC
File systems
Filesystem Goals

- Persistence
  - Even in the face of crashes
- Associate names with bytes
- Associate names with each other (directories)
- Efficient space utilization
- Performance
- Usability?
  - Acts one of principal human-computer interfaces
  - Translate human-readable names into data
Files

- Need to keep track of location of file’s contents (map byte offset to sector)
- Metadata structure usually called an inode
- Inodes must also be stored on disk
- Allocation strategies: tradeoffs
  - Contiguous, linked list, DOS FAT, indexed files
  - Multi-level indexed files (PA4!)
- Inode storage options: fixed-size array vs scattered
Buffer Cache

• Use some main memory to store recently accessed disk blocks
• Frequently referenced blocks in the buffer cache
  – Indirection blocks
• Solves the problem of slow access to large files.
File System Crash Recovery

• Problem:
  – Lost information
  – Crash during update to metadata can leave filesystem in unrecoverable state.
    • Adding blocks to file
    • Creating links to a file

• Approaches
  – fsck
  – Ordered Writes (Synchronous and dependency tracked)
  – Write ahead logging
File System Crash Recovery

• Unix fsck – looks for internal consistency
  – Block in file and also in free list
    • Remove from free list
  – Ref_count for a file descriptor doesn’t match the number of links in dir
    • Change to actual number of links
  – Block in 2 different files
    • Duplicate and give one to each. (Security?)
  – File descriptor has ref_cnt == 0, but its not references anywhere
    • /lost_found
File System Crash Recovery

• Unix fsck Limitations
  – Solves problem of inconsistency but not that of loss of information

• Soft Updates and Dependencies
  – Keep track of dependencies
  – Never write dependender before dependee
  – Soft updates: write blocks in any order, keep track of dependencies, roll back changes you can’t commit yet
  – Fsck split into foreground/background
    • Foreground checks consistency
    • Background reclaims linked space
Other FS Ideas

• Write-head log
  – Write operation to log, then disk
  – Replay the log after crash (may redo ops)
  – Extra overhead

• XFS
  – Breaks disk into Allocation Groups
  – Each AG has own metadata structures
  – B+trees – ordered keys for storing keys/values

• LFS – only keep a log!

• WAFL – write whole snapshots to disk
Networking
Networks

- Desired abstraction: reliable process-to-process communication
  - Reality: multi-hop journey across multiple switches
  - Layered view:
Networks, cont’d

• Use of protocols is key – provide well-defined service
  – TCP – reliable pipe communication
  – UDP – unreliable, best-effort transmission

• Packet encapsulation:
TCP

• Issue: Implementing reliable protocol on top of unreliable layers

• OS concerns:
  – Tracking unacknowledged packets
  – Wake process receiving data
  – Ack received packets quickly

• OS Interface: sockets
  – Similar to pipe abstraction
Consensus
Consensus

Consensus has many applications in robust, high availability systems.

— Banking!

Consensus Goal: get multiple agents to agree on the same value!

1. All must start with the same input value.
2. They must communicate using a consensus protocol.
3. Once decided, agents output chosen value.
Consensus, cont.

Properties of a Consensus Protocol
1. Safety
   — There must be agreement and validity among the output
2. Liveness
   — All non-faulty agents output a value
3. Fault Tolerance
   — Can recover from agent failure
   — Handle agent crashes
   — Handle compromised agents
Consensus, cont.

Theorem (FLP Impossibility Result): No deterministic protocol can guarantee all three in an asynchronous system!

What to do?
1. Randomize
2. No guaranteed termination necessary
3. Assume synchrony
Protection and Security
Access Control

• Matrix view: entry for every combination of user and object
  – Issue: how to store?
    • By rows – Capabilities – each process stores objects it can access
    • By columns – kernel stores list of users who can access
  – Unix: each user has userid, groupids
    • For each file: store file’s owner and group
    • Store three sets of permissions: owner, group, ‘other’
    • Uses rwx bits
    • Root user has all permissions

• Non-file Permissions
  – Bind to sockets, mount/unmount file systems, change owner of a file
  – Unix: restrict to root
Unix Security, cont’d

• Login process: runs as root, authenticates a user
  – Checks hash of password against stored hash, sets user and group id, uses execve syscall to launch shell

• Real and effective user IDs
  – Occasionally need root privilege: change password
  – Each process has a real and effective ID
  – Killing processes: effective or real user IDs must match
Unix Security Holes

• Example attack vectors
  – TOCTOU bugs (similar to race condition)
    • Check for symlink, then run command (attacks may change in the interim period)
    • Solution: lock resources, or use transactions
  – ptrace bug: escalate privileges to debug ssh
  – Lesson: be very cautious
Virtual Machines
Virtual Machines

- **Architecture overview**
  - Single OS: layer between hardware and user programs

- **VM** – must provide platform for virtualizing hardware (VMM)
  - Must "look like" hardware
VM Approaches

• Completely simulate hardware:
  – Bochs: fetch each instruction, decode it, simulate effects on machine state
  – Issue: slow!
  – Idea: most instructions are not affected by processor privilege level – why not just feed to CPU?

• Trap and emulate approach:
  – Most instructions “just work”
  – Privileged instructions trap into monitor, simulate instruction
  – Use indirection to provide illusion of underlying hardware
VM Memory Mapping

• Uses Shadow page table
  – Directly maps Guest VA to Host PA
  – Maintained by VMM
  – When need to track access to a page, invalidate in shadow page table: trap into VMM on page fault
Other VM Concerns

• Memory Management
  — How to know what pages of the guest OS to evict?
    Random eviction, have guest OS return memory to VMM
  — Share pages across VMs
  — Idle Memory Tax
Questions?

• Good luck on the final!