CS 140 Final Exam Review
Administrative Details

• Thursday
  March 19, 12:15 – 3:15pm
  Room TBD (B01?)

• 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
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
Post-Midterm Review
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
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
Process Layout

- Kernel
- Stack
- Heap
- Data
- Code
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
IO and Disks
IO Busses

• Data interconnect between different devices
• Comes in many varieties; use bridges or bus controllers to convert between them
• Single (ISA) vs. Multi-master (PCI)
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
IDE and SATA

• IDE: older parallel interface
  – Controller typically accessed through port IO
  – Data transfer using device DMA or port IO
  – Can be used with or without interrupts

• SATA: Serial ATA
  – More modern
  – Port IO and DMA read/writes, plus new features
  – NCQ – allow drive to re-order commands
Magnetic Disks

• Geometry
  – Platters
  – Actuator arms and Heads

• Data Organization
  – Tracks
  – Sectors
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 $= \text{Seek} + \text{Rotational (}+\text{ Transfer)}$
• 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

• Filesystem vs VM:
  – Both provide location transparency and mapping
  – More extreme range of sizes
  – Files more likely to be dense and sequentially accessed
    • Random access still very possible (DBs, paging)

• File abstraction: named sequence of bytes

• Operations: create, delete, read, write

• Performance dominated by # of disk accesses, not data transfer
Files, Cont’d

• 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.
Other Topics

• Free space management
  – Linked list of free blocks
  – Bitmap
    • Tricks of locality, pretending that the disk has 10% less capacity

• Block sizes
  – 512B (one sector)
  – 4.3BSD : Multiple block sizes.
    • 4Kbytes large blocks
    • fragment concept – reduce fragmentation

• Fast File System [Review lecture slides]
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 it’s not references anywhere
    • /lost_found
File System Crash Recovery

• Unix fsck Limitations
  – Solves problem of inconsistency but not that of loss of information
  – Security Issues
  – Takes a long time

• Soft Updates and Dependencies
  – Keep track of dependencies
  – Never write depender 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
Sockets

• New system calls:

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket – make socket</td>
<td>socket – make socket</td>
</tr>
<tr>
<td>bind – assign address</td>
<td>bind* – assign address</td>
</tr>
<tr>
<td>listen – listen for clients</td>
<td>connect – connect to listening socket</td>
</tr>
<tr>
<td>accept – accept connection</td>
<td></td>
</tr>
</tbody>
</table>

• Implementation
  – mbufs/sk_buffs: fixed-size structures for storing packet data
  – Network Interface Cards: provide ifnet data structure with function pointers for enqueueing and sending packets
  – OS must maintain routing table, reliably transmit packets
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
Security Cont’d

• MAC vs DAC
  – Place limits on propagating permissions
  – Shows need for systematic approach to problem
• Structured approach to security
  – Combination of label and security label – forms lattice that allows us to reason about relationships
• Covert channels
  – Non-standard ways of propagating information
  – Hard to protect against
  – Extreme solution: no sharing
• LOMAC: minimally intrusive MAC, focus on integrity
• Flask: Separate policy from enforcement – distinct server for allowing/denying requests
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

- **Devices**
  - Make in/out instructions trap into VMM
  - Cause Memory-mapped I/O to page fault
  - Simulate actual I/O action

- **Binary translation**: need to run guest OS on VM with decent performance – re-write privileged instructions to modify VMM state

- **Hardware support**: CPU can run in guest mode (different from binary user/kernel)
  - Can specify which ops should trap
  - No need for binary translation
Questions?

• Good luck on the final!