CS 3210 Operating System Design
Fall 2003, MW 12-1

Exam 2 Study Guide

Topics Covered: (2 questions each)

- **Memory Management (ch 7)**
- **Process Address Space (ch 8)**
- **Signals (ch 10)**
- **Process Scheduling (ch 11)**
- **Virtual Filesystem (ch 12)**

Exam is closed-book. There will be 10 questions in 90 minutes with short answers (usually 2 or 3 well chosen sentences). I expect knowledge of general concepts, important ideas and some details. Questions are drawn mostly from the primary text. I highlight important info in class but may draw a detail question from something described in the text but mentioned in passing in class

I recommend using a top-down study strategy. Read the chapter, close the book and ask yourself, "What were the 3-5 most important ideas or concepts presented in this chapter?" A "brainstorming" approach is also useful. I do this in class as review sometimes. "Tell me everything you know about interrupts and exceptions." Just list everything you know and then organize among the major topics you listed above.

Memory Management

Process Address Space

- address space (AS): set of addressable memory locations
  - in theory 0-4GB on 32 bit machine
  - in Linux 0-3GB (kernel is mapped in high 1GB)
  - in reality 0-3GB – high 128MB (used for "high mem", vmalloc, etc.)
- AS mapping: valid (mapped) versus invalid (unmapped)
  - attempts to touch unmapped location: SEGV
  - mapping ultimately done via mmap() sys call
- AS lifecycle: fork -> exec -> mmap -> munmap -> brk -> shmat -> exit
- AS example events:
  - create a process (fork, exec)
  - grow the stack (automatic growth)
  - grow the heap (via malloc)
  - dynamic linking/loading
  - terminate a process
  - system v shared memory: shmat/shmdt
  - file mapping: mmap()
- file mapping
  - little known, popular with kernel hackers
  - make file look like memory
    - reads access file, writes flush through (eventually)
  - compare to /proc/kmem – memory that looks like a file (!)
  - "backing" file
  - optimizes file access (avoids a copy)
    - regular: file to kernel buffer, kernel to user buffer (2 copies)
    - file mapped: file to user space (1 copy)
  - executables are "file mapped"
    - implements loading!

    - "standard" read/write uses file mapping internally
- Note: page fault doesn't change AS mapping
    - page fault (faulting in a page) just allocates/loads page frame
    - mapping must already be defined
    - attempting to touch unmapped region -> exception (SEGV)
- Data structures
    - memory descriptor (current->mm)
        - shared (ref counted) by threads sharing AS
        - important fields: mmap, mm_rb, mm_cache, pgd
    - memory region descriptor (vm_area_struct; VMA)
        - contiguous mapped range with uniform permissions
        - vma list: linked list of these descriptors
        - red-black tree: balanced tree of VMAs for fast access
        - vm_ops: function table for loading, etc.
        - VMA access writes
            - Intel only has 2 bits for permissions!
        - utility functions for manipulating address range intervals
- /proc/*/maps – shows mapped regions
    - bug: contents shown twice
- be familiar with mmap() – man mmap
    - do_mmap
    - so called ANONYMOUS pages
- updating page tables, tlb when AS is modified
- page fault exception handler
    - complex case analysis – be familiar with cases
- copy-on-write: how it's implemented
    - VMA says RW, page table says RO
    - attempt to write triggers page fault
    - page fault handler notices "ah, this is copy on write situation"

Signals

- basic ideas
    - old "event delivery" mechanism
    - simplest form of IPC – just 1 bit (an event happened)
    - software analog of hardware interrupts
    - signals have names and numbers, default actions
        - some platform, architecture dependent (exceptions)
        - know a few example signals
    - processes can "send" signal to another process, group
    - kernel can send signals (e.g. in response to hw exception)
    - signals are sent (generated, raised) then delivered (handled, caught)
        - sending (kill) occurs in context of generating process
        - delivery in context of receiving process
        - signals are delivered on transition from kernel to user
            - pretty frequently: happens at least 100 times per second!
    - possible to "catch" signals: user-specified handlers
        - can't catch KILL or TERMINATE
    - "regular" signals
        - 1-31, "can't count"
    - "realtime" signals
        - 32-63, user-defined, queued ("can count")
        - bounded queue (~1000) to prevent denial of service
        - separate API
    - blocking, masking, pending
        - blocking – don't deliver if generated

- masking – block current signal during delivery
- pending – generated (sent) but not delivered
  - short interval even if not blocked
  - pending but blocked is possible (sigpending())
  - blocked but not pending possible (not generated yet)
- possible to specify signals to be masked during handler

signals and system calls
- blocking (slow) system calls terminated by signal delivery
  - returns EINTR
- possible to specify "auto restart" (BSD)
  - specified when registering signal handler (per signal)

unusual BSD feature – alternate signal stack

APIs
- 3 separate APIs, old API still available
- old "unreliable" signals (signal())
  - bug: not masked (reentrant), default action reinstalled
  - result: window of vulnerability
  - not possible to reliably catch all signals sent
- new POSIX "reliable" signals (sigaction())
  - no window of vulnerability but signals still "can't count"
  - old semantics available as parameters of sigaction()
- new POSIX "realtime" signals (rt_sigaction())
  - possible to include a little extra info
  - signal struct added to queue when generated
  - subtle semantics: interactions between regular, realtime

data structures
- signal sets: array of 2 ints as a bitmask (1 per signal)
- there is no signal 0 (used for testing delivery permissions)
- pending, blocked signal sets
- pending signal queue
- sigpending (convenience Boolean – any pending, non-blocked)
- sig struct – contains array of handlers, flags, masks
- utility routines on signal sets (ugly bit tricks!)

generating signals
- send/force_sig_info()
- check permissions
- some signals cancel out others
- ultimately, setting bit in target process task struct
  - actually, regular signals are queued (once) in Linux
  - seems to be a code simplicity technique
  - bit is still set if queue is full
- for realtime, setup info packet, enqueue

delivering signals
- much more complex than generating
- default actions pretty easy
- executing handlers must be done in user space
  - place a special stack frame on user-mode stack
- must execute all pending, non-blocked
- so must return to kernel after finishing handler
  - special system call: sigreturn()
  - returning to kernel executes code on user-mode stack!
- so called "trampoline code": kernel to user to kernel to user …
- be familiar with sample code on page 333
- be able to explain flow of control in Figure 10-2
- be familiar with signal stackframe layout in Figure 10-3

- kill() system call
    - possible to send to process group, all processes, specific process
- sigsuspend() – mask, wait for delivery atomically
    - sigpending() followed by sleep() not the same!
    - understand why …

Process Scheduling

- basic ideas
    - scheduling (policy), context-switch (mechanism)
    - well-studied, lots of (complex) theory
    - we want a simple, general purpose scheduler that does well most of the time
    - time-slicing, quanta – each process runs for a given interval
        - on quantum expiry process is runnable (not-blocked) but not selected
        - gives others a chance to run (fairness)
    - preemption (technically) – a higher priority process becoming runnable
        - Linux is preemptive for user processes
        - kernel (2.4) is not (2.6 has limited kernel preemption)
    - priority – anti-fairness mechanism ☺
        - UNIX – “niceness” (nice command)
            - regular users – only decrease priority (increasing niceness)
            - root – either increase or decrease
    - process classification
        - interactive, batch, real-time
        - cpu-bound, io-bound
    - scheduling heuristics
        - boost io-bound (like interactive) to make progress, increase responsiveness
        - decrease cpu-bound over time
    - syscalls
        - nice, get/setpriority
        - sched_get/setscheduler (policy)
        - sched_yield
- POSIX scheduling classes (policies)
    - SCHED_FIFO (run to completion – no time-slicing!)
    - SCHED_RR (time-slicing but high priority)
    - SCHED_OTHER (normal scheduler)
- real-time processes
    - SCHED-FIFO and SCHED_RR are so called “real-time” processes
    - always beat regular processes
    - real-time processes have priorities (to distinguish among them)
    - under UNIX only root can make a process real-time (sched_setscheduler())
    - a bit dangerous – can starve other processes, even kernel daemons
    - support for “soft” realtime (best effort)
    - “hard” realtime (guaranteed deadlines) is very difficult
- quantum length
    - too short – wastes cpu on overhead
    - too long – reduces responsiveness
    - linux currently starts at about 6 ms
    - quantum length related to priority (higher priority, longer quantum)
- scheduling “epoch”
    - don’t confuse with timekeeping epoch (jan 1, 1970)
    - epoch occurs when all runnable processes have 0 quantum

        - quantum is refreshed for ALL processes (including blocked processes!)
            - gives io-bound processes a little boost
            - boost is bounded by 2*max_priority
    - task_struct variables related to scheduling
        - nice – niceness (static priority)
        - counter – quantum (dynamic priority)
        - rt_priority – realtime priority
        - policy – SCHED_RR, SCHED_FIFO, or SCHED_OTHER
        - some processor info
        - need_resched
    - actual priority used by scheduler to select (“goodness”) is nice + counter
        - so overall priority decreases a bit during quantum
    - scheduler code
        - Uniprocessor and SMP versions
        - handle special cases
        - iterate through runnable
        - compute “goodness” of each runnable
        - select process with highest goodness to run
            - advantage for same thread as current (no flushing)
            - big advantage for same cpu (processor “affinity”)
        - may select currently running process to run next
        - on preemption, tries to reschedule preempted on another cpu
    - switchto macro
        - context switch
        - three parameters: prev, next, prev
        - before switch there is prev (A), next (B)
        - code after switchto is resumption of B (later)
        - need to know C, process running before B was rescheduled
        - prev is part of restored stack context so it contains A (normally)
        - switchto uses a trick to maintain C in a register and reset prev properly
        - whew! that’s complicated…

Virtual Filesystem

- vfs is abstraction layer to allow multiple distinct filesystem types to coexist
- “common” model (api) is basically standard UNIX semantics
    - e.g. NT has richer access modes not representable
- each new filesystem registers itself, provides standard functions
- vfs api calls do a little work and then dispatch fs-specific function
- many vfs-related system calls
- task_struct files: fs, files
- big four data structures
    - open file object
    - dentry
    - inode
    - superblock
- filesystem type descriptor is also important
- make sure you understand role and interrelationships of big four structures
- some objects have disk representations, other do not
- ones with disk reps have two slightly different formats: disk, memory
- in-memory version functions as a cache of disk version
- need dirty bits to track changes

each struct has a wildcard field (u, private_data) for fs-specific data
vfs layer functions
- sometimes don't need to call a fs-specific function
- sometimes have default implementations
- sometimes just dispatch to fs-specific version

inode – metadata + data pointers (so called "block map")
- inode # is unique (per-device) name of a file

open file object – no disk rep, file pointer, fops, points to dentry
dentry – directory entry abstraction, one line of a directory listing
- basically name + inode

superblock – fs descriptor
- contains list of dirty inodes, times, blocksize, fs options, etc.

unix file links
- hard – just another name for same file
  - basically a dentry
  - can't cross filesystem or link to directory (to avoid cycles)
- soft, symbolic – a small file with another pathname
  - can cross filesystems, cycles are possible

dentry cache
inode cache