Midterm Review

COMS W4118
Prof. Kaustubh R. Joshi
krj@cs.columbia.edu

http://www.cs.columbia.edu/~krj/os

References: Operating Systems Concepts (9e), Linux Kernel Development, previous W4118s

Midterm Logistics

- Next week during class time
- 60 minute exam: please be on time
- Closed book, closed notes, closed electronics
  - Allowed to bring one sheet of letter paper with handwritten notes on both sides
  - Can use old-school calculator
- Format
  - 4 questions
  - 200 points
  - Extra credit questions

What’s in

- Question types
  - Multiple choice questions
  - Short answers (2-3 sentences)
  - Numerical problems
  - Carry out tasks based on things you learned in class
  - Code related problems (write or analyze pseudocode)
- What’s in
  - All material up to and including scheduling
  - **Lecture slides** (or if I said it in class)
  - Concepts 9e, Ch. 1-7
  - Linux Kernel Development: Ch. 2-10
  - Based on general concepts

What's Out

- No memorization needed, but will expect you to know:
  - High level mechanisms (monitors)
  - What functions do in general (e.g., lock/unlock)
  - What certain data-structures are used for in general (e.g., task_struct)
- I expect you understand the concepts at a base level
  - Won't ask you to explain them
  - Test your understanding of concepts through applied questions
- No need to memorize
  - Synchronization algorithms (but be prepared to explain)
  - Specific Linux function semantics
  - Data Linux structure elements
  - Which OS implements which facility in what way
- No long descriptive answers
- No need to write working code (pseudo-code possible)

Syllabus

- General concepts and Linux/Android specifics
  - Basics
  - OS Architecture
  - Events
  - Processes
  - Threads
  - Synchronization
  - Synchronization Errors
  - Scheduling

Basic Architecture Concepts

- What hardware provides
  - – Stored program computer
  - – Instruction types
  - – Memory model (multicore, SMP, cache)
  - – I/O, memory-mapped I/O
  - – Interrupts, DMA
  - – Basic structures: stacks, heaps

OS Functions

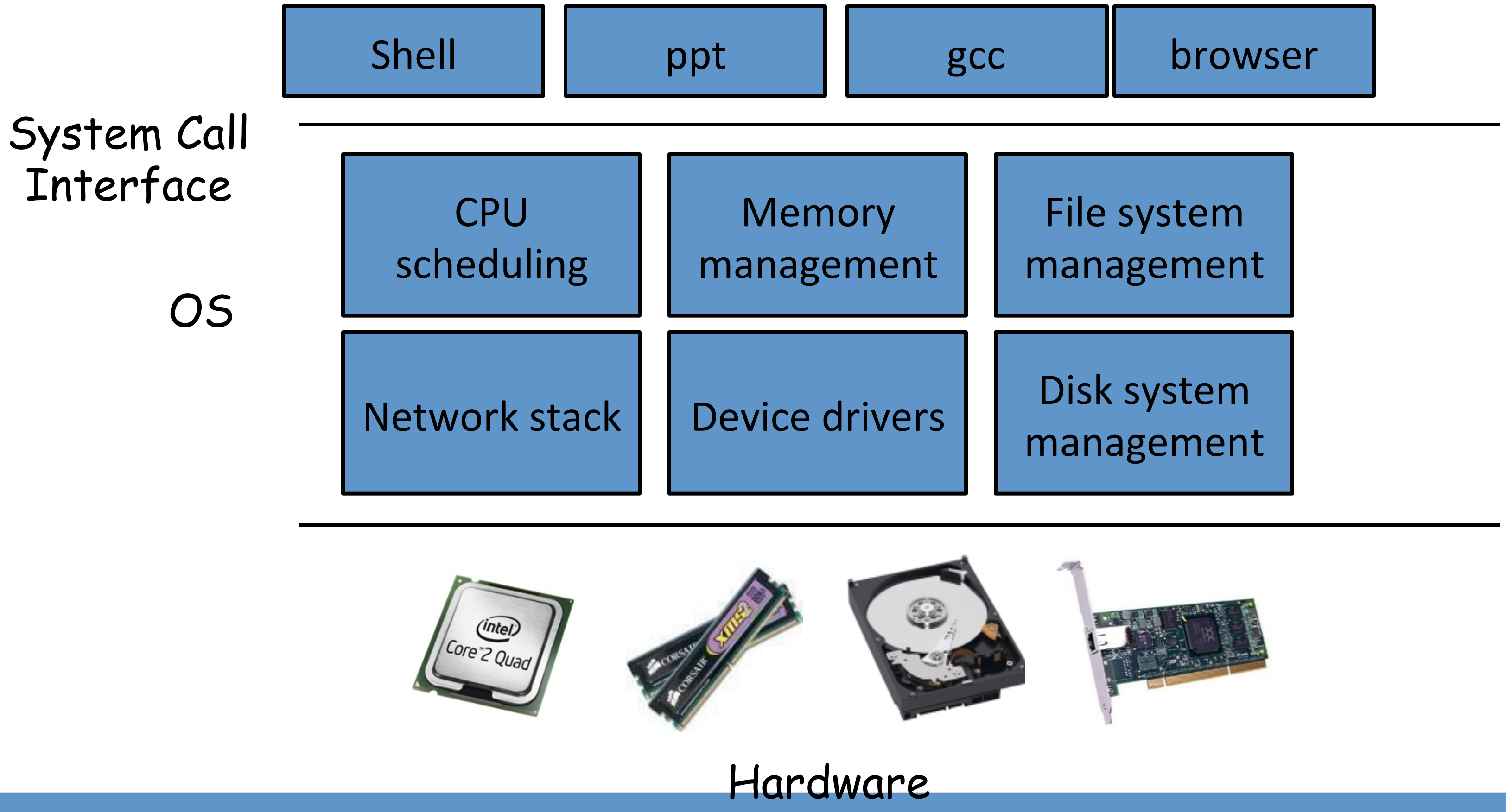
- What does an OS do?
  - Support multiprogramming
  - Resource allocation
  - Isolation
  - Abstraction
  - Shared facilities and libraries
- Pieces of an OS
  - Kernel
  - Scheduler
  - Memory management
  - File systems
  - Device drivers

OS History

- Early OSes
  - Monitors
  - Batch Processing
  - Spooling
  - Multiprogramming
  - Timesharing
- Types of modern OSes
  - Mainframes, Clusters, Servers, Desktops, Mobile, Embedded
  - Trends

OS = resource manager/coordinator

- Computer has resources, OS must manage.
  - Resource = CPU, Memory, disk, device, bandwidth, …



Shell | ppt | gcc | browser

System Call Interface

OS

CPU scheduling | Memory management | File system management

Network stack | Device drivers | Disk system management

Hardware

OS = resource manager/coordinator (cont.)

- Why good?
  - Sharing/Multiplexing: more than 1 app/user to use resource
  - Protection: protect apps from each other, OS from app
    - Who gets what when
  - Performance: efficient/fair access to resources
- Why hard? Mechanisms vs. policies
  - Mechanism: how to do things
  - Policy: what will be done
  - Ideal: general mechanisms, flexible policies
    - Difficult to design right

OS Abstractions

- Processes
- Address spaces
- Files
- Directories
- IPC: pipes, Shared mem, sockets
- Threads

OS = hardware abstraction layer

- "standard library" "OS as virtual machine"
  - E.g. printf("hello world"), shows up on screen
  - App issue system calls to use OS abstractions
- Why good?
  - Ease of use: higher level, easier to program
  - Reusability: provide common functionality for reuse
    - E.g. each app doesn't have to write a graphics driver
  - Portability / Uniformity: stable, consistent interface, different OS/ver/hw look same
    - E.g. scsi/ide/flash disks
- Why hard?
  - What are the right abstractions ?

OS Architecture

- Privileged mode
  - Why needed
  - How implemented
  - Address spaces and privileges

- OS Structures
  - Monolithic kernels
  - Microkernels
  - Kernel modules
  - Virtual machines

Processes

- Process: an execution stream in the context of a particular process state or “virtual CPU”
    - Seperately scheduled, isolated, protected
    - Per-process kernel stack
    - Process creation, copying, destruction
    - Process relationships: parent, child, special processes
- Process state
    - Registers, memory, I/O
    - Run states: ready, running, blocked, zombie, dead, new
- Process management
    - Process control block (Linux: task_struct)
    - Process list
    - Wait queues

Address Space

- Address Space (AS): all memory a process can address
  - Linear array of bytes: [0, N), N roughly 2^32, 2^64
  - Physical layout vs. address space layout
- Address space = protection domain
  - OS isolates address spaces
  - One process can't even see another's address space
  - Same pointer in different AS point to different memory
  - Can change mapping dynamically
- Address spaces and context switching
  - Need to change address space
  - Expensive operation
  - Impact on cache

Process Dispatch

- What is process dispatch?
- When does dispatch occur?
  - Cooperative vs. preemptive multitasking
- How does dispatch occur?
  - Context switch: change CPU state from one process to other
  - Registers, address space, files, stacks, I/O
  - Role of PCB (task_struct) in storing state
  - Role of kernel stack

System calls

- User processes cannot perform privileged operations themselves
- Must request OS to do so on their behalf by issuing system calls
  - System calls vs. API calls
  - How syscalls are invoked: hardware mechanisms
  - System call tables
  - Parameter passing through registers, stack, memory
  - System calls and privilege changes
- System calls must treat user data with care
  - Copying data to/from userspace

Signals

- Interrupts to processes
  - Notification from kernel to process
  - Also IPC mechanism between processes
  - Synchronous vs. Asynchronous
  - Catchable, different default actions
  - Watch for race conditions

Inter Process Communication

- Message passing and shared memory
  - Pros and Cons
  - Implementation issues
    - Synchronous vs. asynchronous
    - Blocking vs. non-blocking
    - Buffering
    - Addressing
- Many different examples
  - Message passing: pipes, sockets, RPC, Binder
  - Shared memory: Sys V shmem, mmap

Threads

Threads

- Threads: separate streams of executions that share an address space
  - Allows one process to have multiple point of executions, can potentially use multiple CPUs
- Thread control block (TCB)
  - Program counter (EIP on x86)
  - CPU Registers, Stack
- Why threads?
  - Concurrency, Multicore, Efficient data sharing

Threading Models

- User threads
  - Pros: fast context switch, Cons: block on syscalls, no multicore
- Kernel threads
  - Pros: no blocking on syscalls, multicore Cons: overhead
- Many-to-many threads
  - Pros: no blocking, multicore, more efficient than user or kernel threads, Cons: complex
- Scheduler activations
  - Pros: no blocking, low overheads, Cons: multicore, complex, need upcalls

Linux/Android Process Lifecycle

- Linux PCB
  - task_struct: same for processes and threads
  - Per process/thread kernel stack
- How processes and threads are differentiated
  - Thread groups, pid
  - Threads share same address space
- fork() vs vfork() vs clone()
  - Performance
  - Semantics
- Role of distinguished processes
  - init, zygote
- Process termination
  - exit() call, zombie on exit

Interrupts

- Why?
  - Preemptive multitasking, efficient control of I/O devices
  - Role of timer interrupt in scheduling
- Types of interrupts
  - Hardware interrupts, exceptions
  - Faults, traps, aborts: examples
- Handling interrupts
  - Role of PICs, APICs
  - Interrupt descriptor table
  - Interrupt service routines
  - Nested interrupts, exceptions
- Interrupts in Linux
  - Interrupt stacks
  - Deferred work: softirqs, tasklets, work queues
  - When to use deferred work?

Synchronization

- Why needed?
  - Shared data access, synchronization of actions
  - Race conditions
  - Atomic operations
- Critical section problem
  - What is it? Requirements?
  - Mutual exclusion, progress, bounded waiting
  - Desirable properties: efficient, fair, simple
- Solutions to critical section problem: locks (mutex)
  - Disabling interrupts. When does it work?
  - Hardware based implementation: atomic instructions, test and set, atomic exchange
  - Software algorithms: Peterson's, Bakery algorithm
  - Spinlocks, sleeplocks, adaptive mutexes: when to use
  - Reader-writer locks

Lamport's Bakery Algorithm

- Support more than 2 processes
  - Integer tokens (increasing numbers)
  - Each customer gets next largest token
  - Same token? Smaller thread_id gets priority
  - Smallest token enters critical region

```
bool flag[1..NUM_THREADS] = {0};     // Want to enter
int token[1..NUM_THREADS] = {0};    // My token
lock(i) {    // Lock by thread i
   flag [i] = 1;
   token[i] = 1 + max(token[0]+…+token[NUM_THREADS-1]);
   flag[i] = 0;
   for (j = 1; j <= NUM_THREADS; j++) {
      while (flag[j]); // Is j getting token?
      while ((token[j] && ((token[j], j) < (token[i], i))); // j has smaller token?
}
```

```
unlock(integer i) {
   token[i] = 0;
}
```

Reference: A New Solution of Dijkstra's Concurrent Programming Problem. L. Lamport. Communications of the ACM, 1974. http://research.microsoft.com/en-us/um/people/lamport/pubs/bakery.pdf

More Synchronization

- Memory barriers
  - Prevent reordering on superscalar processors
  - Needed to ensure locking algorithms work
- RCU (read-copy-update)
  - Lock-free synchronization
  - Readers require no locks
  - Relies on atomic updates by writers
  - Garbage collect old data after readers done
  - Writers must synchronize between themselves

Semaphores and Monitors

- Higher level constructs
  - Semaphores
    - An simple integer with atomic, race-free access
    - Post: increment by 1 and return immediately
    - Wait: wait until > 0, then decrement and return
    - No strict "locking" semantics, different process can post and wait
    - Allows solution to ordering problems in addition to critical section
  - Monitors
    - Protect a set of functions that access common data from being executed concurrently
    - Need additional signaling primitives: condition variables, wait, signal
    - Using condition variables makes code susceptible to races, deadlocks
- Learn to:
  - Solve problems using these constructs (including locks)
  - E.g., dining philosopher, producer-consumer
  - Identify problems: race conditions, deadlocks, etc.

Synchronization Errors

- Races: what they are
  - Identify them when they occur
  - Techniques to avoid races
  - Techniques to detect races: Eraser lock-set algorithm
- Deadlocks: what they are
  - Identify them when they occur
  - Techniques for deadlock detection/avoidance
  - Ordered access to resources
  - Cycle detection
  - Banker's algorithm

CPU Scheduling

- Scheduler
  - High-level policy
  - Responsibility: deciding which process to run
- When is a scheduler invoked?
  - Co-operative vs. pre-emptive scheduling
- Scheduler metrics
  - Waiting time, utilization, throughput, response time, fairness

Scheduler Algorithms

- General scheduling algorithms:
  - RR, FCFS, SJF
  - Role of priority
  - Pre-emptive vs. non-preemptive versions of algorithms
- Specialized scheduling algorithms: realtime
  - Rate monotonic scheduling
  - EDF
  - Optimality
- Linux Scheduling
  - Completely fair scheduler
  - Scheduling latency
  - Which process to pick: pick process with least runtime
  - What time slice to set: based on fair share of timeslice

Scheduling Evaluation

- Gantt Charts
  - Evaluation against simple workloads
  - Deterministic
  - Learn how to use to compute scheduling metrics (wait time, response time, etc.)
- Probabilistic evaluations
  - Queuing networks
  - Little's law: $n = \lambda \times W$
- Trace based Simulation
  - Can simulate complex systems
  - Can evaluate realistic workloads

Advanced Scheduling

- Hierarchical Scheduling
  - Combine multiple scheduling policies
  - Achieve different outcomes for different classes of processes
  - Feedback vs. non-feedback scheduling
- Multiprocessor scheduling
  - Single run queue vs. per-CPU run queue
  - Impact on cache
  - Processor affinity
  - Load balancing: push vs. pull
  - Gang scheduling
- Additional issues
  - Fairness and aging
  - Priority and priority inversion