Administrivia

• Project 2: Userprog was due last night
• Midterm: Tuesday, Feb 8
• Project 3: VM is due Thursday, Feb 24

• In class midterm, same place, same time
  - Local SCPD students are encouraged to take exam in person

• Midterm is open book and notes, closed computers

• Exam grade is max, but you must take the midterm
  max (midterm > 0 ? final : 0, (midterm + final) /2)

• Covers all material up to now.
Lectures

- Introduction
- Processes & Threads
- Concurrency
- Scheduling
- Advanced scheduling
- Synchronization
- Linking
- Virtual memory HW
- Virtual memory OS
- Memory allocation
Introduction

• **OSes make hardware useful to the programmer**

• **OSes Provide:**
  - Well-defined hardware abstraction (through syscalls)
  - Resource allocation (e.g. CPU time, files)
  - Protection (e.g. memory)

• **Execution Contexts:**
  - user level
  - kernel process context (e.g. syscalls)
  - kernel w/o process context (e.g. timer interrupt)

• **Throughput vs Fairness**
Processes & Threads

- **A process is an instance of a program running**
  - Process Control Block (PCB)
  - Allocated Resources (e.g. files, memory mappings)
  - Credentials (user, group)

- **A thread is a schedulable execution context**
  - Stack, registers, etc. **Context Switch!**
  - Provide concurrency
  - 1:1 – multiprocessor, everything goes through kernel (slow)
  - n:1 – single processor, user library
  - n:m – multiprocessor, managing multiple schedulers

- **Scheduling – How to pick which process to run**
  - balance fairness, priority, deadlines, throughput, efficiency
Concurrency

• **Sequential Consistency (SC):** execute in order written
  - Maintain *program order* in processor, ensure *write atomicity*
  - SC thwarts hardware and compiler optimizations

• **Data races (examples in lecture)**

• **mutual exclusion, progress, and bounded waiting**
  - Peterson’s Solution works (for only fixed *n* threads)

• **synchronization primitives (mutex)**
  - multiple synch primitives (e.g. lock, sema, condvar in *pintos*)
  - implementation abstracted away from programmer
  - disable interrupts (only works in uniprocessor)
  - *spinlocks* need atomic *read-write* ops, (*test_and_set()*, *xchg()*
Scheduling

- **Criteria:** *throughput, turnaround time, response time*
  - secondary criteria: *CPU utilization, waiting time*

- **Processes exhibit bursts of I/O and computation**

- **Strawman examples (analyze in terms of criteria)**
  - FCFS: simple, but problematic (e.g. convoy effect)
  - SJF: minimizes waiting time (and RT), but starves longer jobs
  - RR: fair and prevents starvation (bad for same size jobs)

- **Costs of context switching**
  - CPU, cache, TLB, page swapping, buffer cache, etc.

- **Priority Scheduling**
  - inter-thread dependencies, priority donation (inversion!)
  - BSD: tunable, prevents starvation, no donation
Advanced Scheduling

• Linux 2.6 scheduler, task lists
• Lottery Scheduling: tickets for CPU usage
  - ticket donation, compensation, splitting, etc.
  - latency prediction? (multimedia) error?
• Stride Scheduling: ideas from lottery and WFQ
  - lessens error and latency
• Borrowed Virtual Time:
  - effective virtual time, weights, warp factor
• SMART: separate importance and urgency
  - manages real-time conventional tasks
  - schedules based on (priority, biased virtual finishing time)
• Linux today: pluggable schedulers (real-time, CFS)
Synchronization

- deal with multiple readers and writers
- shared locks: multiple reader, single writer
  - implemented with spinlocks; may lead to starvation
- relaxed consistency model: memory barriers in locks
- cc-NUMA: each CPU has fast access to close memory
  - spinlocks are unfair, and cause lots of traffic on mem bus
  - either avoid spinlocks: single writer, non-blocking (CAS())
  - or use better spinlock: MCS lock (spins on local memory)
- Locks & scheduling, expensive to switch to kernel
  - futex: ask kernel to sleep only if mem hasn’t changed
- Deadlock: mutex, no preempt, hold and wait, cycle
Linking

• referring to and merging separate name spaces

• Compiler produces:
  - Symbolic references to external data/functions
  - Table of locally defined data/functions

• Linker
  - Pass 1: Arrange segments in memory
  - Pass 2: Patch addresses (relocate, replace symbols)

• Dynamic vs Static
  - dynamic linking: defer loading code until needed
  - static libs: at same address in programs
  - dynamic libs: let any library be loaded at any VA

• code and data segments
Virtual Memory HW

• issues of protection, transparency, resources
• Give each program its own virtual address space
• Segmentation: base + bounds registers
  - simple, not transparent, external fragmentation
• Paging: map virtual pages to physical pages
  - simplifies alloc and free, transparent, internal fragmentation
• x86 page translation
Virtual Memory OS

- more virtual memory than physical memory
  - save unused virtual pages to disk
  - on use, swap them in
  - what to fetch? what to eject?

- LRU: Least Recently Used
  - clock algorithm: use accessed bit

- Global vs Local allocation

- Memory Mapping: mmap

- Thrashing and working set
Memory Allocation

- dynamic allocation’s main problem is fragmentation
  - different lifetimes, different sizes

- decisions: where in free memory to put a block?
  - there will always be pathological cases
  - best fit: allocate to leave the smallest fragment
  - first fit: pick the rst block that fits

- Other:
  - memory usage patterns (peaks, ramp)
  - fault resumption = power
  - distributed shared memory
  - garbage collection
  - reference counting
Good Luck!

Any Questions?