15-410 Mid-Semester Review

Dave Eckhardt
Brian Railing
Synchronization

• First Project 3 checkpoint
  – Wednesday, March 4th during class time
  – Meet in Wean 5207
    • Watch e-mail for your personal arrival time
Synchronization

• Exam scheduling
  – If you have received mail from me about filling out a form but figured you didn't need to, actually we do need you to fill it out. Before noon today, please!
Synchronization

• Exam study materials
  – HW1 out soon, due before the exam
    • NOT AT MIDNIGHT
  – Archive of old mid-term exams (course web site)
Synchronization

VM is not on the exam

- It could be, but it'll be more fun on the final

• Threading vs. this exam
  - You are responsible for conceptual material covered in class.
  - Your thread library may not be perfect...
  - ...but we expect you to solidly understand what the thread-library primitives do and how to correctly use them. This includes avoiding standard “anti-patterns”!
Synchronization

- Exam will be closed-book
- Who is reading comp.risks?
- About today's review
  - Mentioning key concepts
  - No promise of exhaustive coverage
  - Reading *some* of the textbook is advisable!
- Will attempt a 4-slide summary at end
Hardware

- Inside the box – bridges
- User registers and other registers
- Fairy tales about system calls
- Kinds of memory, system-wide picture
  - User vs. kernel
  - Code, data, stack
  - Per-process kernel stack
- Device driver, interrupt vector, masking interrupts
Hardware

- System clock
  - “Time of day” clock (aka “calendar”)
  - Countdown timer
Process

- Pseudo-machine (registers, memory, I/O)
- Life cycle: fork()/exec()
  - specifying memory, registers, I/O, kernel state
  - the *non-magic* of stack setup (argv[])
  - the *non-magic* function that calls main()
- States: running, runnable, blocked, zombie
- Process cleanup: why, what
Thread

• Core concept: schedulable set of registers
  – With access to some resources
    • Address space, system-level objects
      – (Mach terminology: “task”)
  – Thread stack

• Why threads?
  – Cheap context switch
  – Cheap access to shared resources
  – Responsiveness
  – Multiprocessors
Thread types

• Internal (N:1)
  - optional user-space library
  - register save/restore (incl. stack swap)

• Features
  - only one outstanding system call (without tricks)
  - “cooperative” scheduling might not be
  - no win on multiprocessors
Thread types

- **Kernel threads (1:1)**
  - resources (memory, ...) shared & reference-counted
  - kernel manages: registers, k-stack, scheduling

- **Features**
  - good on multiprocessors
  - may be “heavyweight”
Thread types

• M:N
  – M user threads share N kernel threads
    • dedicated or shared

• Features
  – Best of both worlds
  – Or maybe worst of both worlds
Thread cancellation

- Asynchronous/immediate
  - Don't try this at home
  - How to garbage collect???
- Deferred
  - Requires checking or cancellation points
Race conditions

• Lots of “++x vs. --x” examples using table format
• “Race-condition party” algorithms
  – e.g., Bakery
• The setuid shell script attack
  – (as an example in a different arena)
• This is a core concept
  – (not limited to one part of the course, or to the course as a whole)
Wacky Memory, “Modern” Machines

- Memory writes may be re-ordered or coalesced
- That's not a bug, it's a feature!
- You may generally assume old-fashioned memory for this class
1. Atomic sequences

- "short"
- require non-interference
- typically nobody *is* interfering
- store->cash += 50;
- Encapsulate in "mutex" / "latch"

1. Which data items must be operated on as a unit?
2. Assign them a synchronization object
3. Code sequences using the data must go through the object
2. Voluntary de-scheduling

- “Are we there yet?”
- We *want* somebody else to have our CPU
- **Not-running** is an OS service!
- Atomic:
  - release state-guarding mutex
  - suspend execution
- Encapsulate in “condition variable”
Critical Section Problem / Protocol

• Three goals
  − Mutual exclusion
  − Progress – choosing time must be bounded
  − Bounded waiting – choosing cannot be unboundedly unfair

• Two “historical example” solutions
  − “Taking Turns When Necessary” algorithm (more generally known as “Peterson's Algorithm”)
  − Bakery algorithm
Mutex implementation

- Hardware flavors
  - XCHG, Test&Set
  - Load-linked, store-conditional
  - i860 magic lock bit
  - Basically isomorphic
- Lamport's algorithm *(not on test)*
- “Passing the buck” to the OS (or why not!)
- [Oddity not on mid-term: Kernel-assisted instruction sequences]
Bounded waiting

- One approach discussed
- Are there others?
- How critical in real life?
  - Why or why not?
  - When?
Environment matters

- Spin-wait on a uniprocessor????
- How reasonable is your scheduler?
  - Maybe approximate bounded waiting is approximately free?
Condition variables

- Why we want them
- How to use them
- What's inside?
- The “atomic sleep” problem
Semaphores

• Concept
  – Thread-safe integer
  – wait()/P()
  – signal()/V()

• Use
  – Can be mutexes or condition variables

• 42 flavors
  – Binary, non-blocking, counting/recursive
Monitor

- Concept
  - Collection of procedures
  - Block of shared state
  - Compiler-provided synchronization code
- Condition variables (again)
Deadlock

• Definition
  – Group of N processes
  – Everybody waiting for somebody else in the group

• Four requirements

• Process/Resource graphs

• Dining Philosophers example
Prevention

• “Four Ways To Forgiveness”
• One is used particularly frequently
  – You should know it
  – You should also not believe that is “the way to solve deadlock”
Avoidance

- Keep system in “safe” states
  - States with an “exit strategy”
    - Assume some process will complete, release resources
    - Make sure this enables another to finish, etc.
    - Banker's Algorithm
Detection

- Don't be paranoid (but don't be oblivious)
- Scan for cycles
  - When?
  - What to do when you find one?
Starvation

• Always a danger
  – Understand vs. deadlock

• Solutions probably application-specific
Context switch

- **yield() by hand (user-space threads)**
  - No magic!

- **yield() in the kernel**
  - Built on the magic `process_switch()`
  - Inside the *non-magic* `process_switch()`
    - Scheduling
    - Saving
    - Restoring

- **Clock interrupts, I/O completion**
Addresses

• Where addresses come from
  – Program counter
  – Stack pointer
  – Random registers

• Parts / areas / segments / regions of a process/program
Summary – What is an OS?

- Parts of a machine
  - Memory, registers
  - Interrupts/traps and their handlers

- Parts of a process (incl. thread)
  - Memory, registers, stack
  - System calls (stubs, handlers)

[Next slide: covered, but not coded, so not on test]
Summary – What is an OS?

- How to assemble machine parts into process parts
  - How to make virtual memory from physical memory
  - How to make a process from memory and registers
    - And an executable file

- How to share a machine among processes
  - (and how to share a process among threads)
  - Context switch/yield
Summary – Synchronization

- Basic RAM-based algorithms
  - Be able to read one and think about it
  - Avoid common mistakes
    - Don't claim a thread runs at zero speed
- Mutex, condition variable
  - When to use each one, and why
  - What's inside each one, and why
Summary – Deadlock

• A fundamental OS problem
  – Affects every OS
  – No “silver bullet”

• What you need for deadlock

• Prevention, Avoidance, Detection/Recovery
  – What each is, how they relate

• Starvation
Preparation

- Archive of old mid-terms
  - *Write down* answers!
- Homework 1
- Don't forget to get some sleep