CSE 451: Operating Systems
Spring 2021

Module 7.5
Midterm Review

John Zahorjan
Mechanics

• Midterm is Wednesday, 5/5
  • No class, no JZ office hour on 5/5

• It will be a Canvas quiz

• You’ll have 50 minutes to complete it once you start

• It will be available between 1:00 am PDT and 9:00 pm PDT

• If you have questions while taking it, please email cse451-staff@cs
  • Do not post to the discussion board

• I will try to answer questions promptly between 9:00 am and 5:00 pm PDT
  • Probably not at all before 9:00 am, possibly promptly nearing 8:00-9:00 pm

• I will be doing nothing else except waiting for questions between 11:00 am PDT and 1:00 pm PDT

• Some questions and answers may be posted by us to the discussion board
Material Covered

• Class material through today
  • Slide sets 1 (Introduction) through 7 (Synchronization (cont.))

• Labs 1 and 2
  • Have you been doing them?
Format / Resources

• Mix of multiple choice and short answer

• You can use any resource available to you except other people
  • Don’t convey anything related to the midterm to anyone during the period
    the exam is available, including stackoverflow and the like

• The goal is to have many, relatively simple problems
  • Simple enough that you can use what you know to answer them all within 50
    minutes
  • Enough of them that looking up answers during the exam won’t have a good
    result
Slide Set 1: Introduction

• What are the roles of the OS?

• What does it mean to share the resources of the computer?
  • Who are they shared among?
  • When does the OS itself get a chance to run?

• What is “required” to share the resources?
  • Why is isolation important? Could you build an OS that didn’t provide it? Would such a system be useful? Would there be any advantage to such a system?
  • What mechanisms does the OS provide/use to isolate
    • Memory
    • CPU
    • Disk
Thematic Issues: Policy vs. Mechanism and Deferring Policy

• What mechanisms does the hardware provide?
  • What policies does it enforce?
  • Deferring policy to higher levels is the essence of computing hardware
    • Does the hardware do anything without software?

• Which mechanisms are oriented to/vital to implementing the OS?

• What are example abstractions built by the OS upon these mechanisms?
  • One is “the OS” itself...

• The OS as an enabler
  • Simplify implementation of applications vs. efficiency of applications
    • Code time vs. run time efficiency
    • Portability as a code time consideration
Themes

• What does it mean for the OS to be efficient?
• (Logical) operations can happen at very different timescales on computers. What approaches can be applied to deal with very slow ones (long latency)?
• Policy/mecchanism distinction and the idea of deferring policy
• Interposition as a way to evolve
• Naming
• Synchronization
  • Concurrency vs. parallelism
Slide Set 2: Architectural Support

• What is the basic control flow of the system?
• Why do transitions from user code to the OS take place?
• Since they run on the same CPU, why can’t applications do everything the OS can do?
• What happens on a transition from user code into the OS?
• On a transition from the OS to user code?
• What mechanisms does the hardware provide to help the OS keep control of the system?
• When the OS is running, what stack is it using (in xk)?
• How does xk use the segmented memory system provided by x86_64?
• How is memory protected?
• How are IO devices protected?
• What is an argument against protection?
Slide Set 3: OS Components and Structure

• Why is “components and structure” a topic?
  • Why isn’t there a clear answer?
• How does OS structure help or hinder portability of the OS?
• How does OS structure help or hinder debugging of the OS?
• How does OS structure help or hinder extensibility of the OS?
• How does OS structure help or hinder run time performance of the OS?
• What are some example OS structures?
Slide Set 3: OS Components and Structure

• Processes / threads
  • Why have a process abstraction?
  • Distinction between a process and a thread?
  • Running / runnable / blocked states

• Memory management
  • Virtual address spaces (cse 351)

• I/O devices
  • How is innovation (extensibility) supported?
  • I/O device vs. file system

• Shells / Windowing / Networking

• Virtual machines
Slide Set 4: Processes

• Why have an abstraction like “process”?
• Memory layout of address space
  • What’s special about a stack?

• Process control blocks and runtime state of process
  • Running / runnable / blocked (single threaded process...) / zombie
  • Process metadata
  • Contents of address space plus CPU state (registers)

• Context switching
  • The basis for sharing
  • What is the mechanism?
  • How is it different than procedure call?
    • How is it the same?
Process Creation

• fork()/exec(path-to-executable, args) vs. createprocess(path-to-executable, ..., args)

• Relationship of fork to
  $ ./myprogram >output.txt

• Relationship of fork to
  $ cat myfile.txt | grep Due | wc

• vfork() and copy-on-write fork

• Communicating “arguments” to subprocesses
  • Inherited meta-data
  • Meta-data modified by parent code running in new process
  • Explicit args
  • Inherited Environment
Process Communication/Synchronization/Abstraction

• wait()
• signals (kill())
• Other: generic
  • Files
  • Pipes
  • Named pipes
• Other: workarounds
  • setuid executables
    • Compare/contrast with trap mechanism for entering kernel

• Process abstraction
  • Session abstraction
  • Process group abstraction
Slide Set 5: Threads

• Thread vs. process
• Why do we want threads?
• (Concurrency vs. parallelism)

• Why does each thread have its own stack?
  • (What’s special about stack memory?)
  • Is stack memory thread private?

• The key idea to a thread is a control flow
  • Has a stack
  • Can be paused and resumed simply by saving and restoring its CPU context
Kernel threads vs. User Level threads

• Saving and restoring registers is NOT privileged
• Allocating cores to threads IS privileged

• Can create a thread (control flow) abstraction at user level, including context switching among threads

• The kernel allocates a core to a kernel thread
  • When the OS is entered on that core, it can determine what kernel thread it was running and save registers in structure for that kernel thread

• Each kernel thread created by a user-level thread package is an opportunity for the application to be allocated a core
  • Kernel can’t allocate more cores to app than it has kernel threads
  • It can allocate fewer...
Scheduler Activations

• An application may create many user-level threads (using a user-level thread package that knows how to create/save state/restore state/terminate them)
• If application code executes a blocking system call (e.g., read)
  • The OS is entered, because it’s a system call
  • The OS saves registers in a structure associated with the kernel thread that the OS last allocated that core to
  • The app has just lost a core, so it needs a chance to decide if the set of threads it is running on the cores it still has is the best choice
• Conversely, if the OS allocates an additional core and restores the state of a kernel thread running in that app, the app gains a core
  • The user-level thread package should get a chance to decide what thread should run on that core
• Scheduler activations are a way for the OS to send “an upcall” to the user-level thread package when the number of cores allocated to it changes
Slide Set 6: Synchronization

• A correct concurrent program must be correct for every possible physical execution

• What are the possible physical executions?
  • Constrained by ordering semantics within a single control flow
  • Constrained by synchronization operations between control flows

• Critical sections
  • Correct execution if executed in a non-overlapping way
    • Possible incorrectly if distinct executions overlap or interleave
    • Effect is “atomic”
  • Read-modify-write of a shared variable
  • Need mutual exclusion
  • A lock is a synchronization variable that provides mutual exclusion
Locks

• acquire()/release() (or lock()/unlock())
• Semantics vs. implementation
• Implementations
  • Spinlocks
  • Mutexes (blocking locks)
• Use spinlocks when the expected spin time is reliably short
  • Use blocking locks otherwise
• Use spinlocks to implement blocking locks
  • A spin lock is used to guard access to the structure that represents the mutex
    • The lock state and a queue of waiting threads
  • The guarding spinlock is held until either the lock state is changed to locked
    or the thread has enqueued itself on the wait list
Implementing spinlocks

• Acquire(): Need to read current lock state and set it to locked if it’s unlocked
• That’s a read-modify-write, so it’s a critical section
• Can’t resolve it using spinlocks because we’re trying to implement spinlocks
• Need lower level (hardware) support

• Test-and-set: fetches contents of a memory location into a register and writes 1 there
• Exchange: swaps a register and contents of a memory location
• Disabling interrupts?!
• Blocking as a basic thread operation
  • Note that user-level threads must block by using code in the user-level thread library, and kernel threads must block using code in the kernel
    • Because that’s the level at which the data structures tracking the states of the threads live
  • That means synchronization variable implementations must exist in kernel code and in user-level code

• Yield’ing vs. sleep’ing vs. wait’ing (block’ing)
  • Yield is “I can run, but I think my progress right now probably isn’t very important so run some other thread if there any ready”
  • Sleep is an abomination
  • You block yourself; someone else wakes you up
Condition variables

- A blocking synchronization variable where the decision about when to block is deferred to the application
- The application needs to (a) evaluate the blocking condition, and then (b) block if necessary.
  - For the result of (a) to mean anything at the time (b) is performed requires mutual exclusion (i.e., a lock)
- The lock cannot be held while the thread is blocked, but...
- The lock cannot be released before the thread is blocked

- Condition variables solve this
  - Atomically release the lock and block the thread
  - Wait(cv, lock) and signal(cv) (and broadcast(cv))
Memory Consistency

- Memory consistency is how writes to memory by one core are seen by others
- Programmers would like all cores to see writes in the order they occurred on the core that wrote them
- Hardware would like the flexibility to push values to memory in a way that is most efficient
  - Programmers must reason statically; hardware would like to optimize dynamically
- Compromise:
  - Hardware provides a “memory barrier” operation that flushes all writes to memory before it finishes
  - Infrastructure software implementer includes memory barriers in the implementation of operations on synchronization variables
  - Programmer respects that correct code must use synchronization variables to achieve synchronization
    - With that restriction, the code sees updates as to shared values as though they were performed atomically
Guidelines for Multithreaded Programs

• Always use synchronization when accessing shared values
• Use locks and condition variables for synchronization
• Use the procedure as the unit of mutual exclusion
  • Acquire lock at beginning, release at end
• Always wait() in a while loop
• If your code contains a call to sleep(), most likely you’re doing it wrong
Midterm Wednesday

• Don’t stress!

• The final course grade will reflect what we think you have mastered by the end of the course, so...

• If you do really well on the midterm, great!

• If your midterm result isn’t what you were hoping for, hey, the course has a long way to go

• It is much harder to catch up than to keep up