CSE 451: Operating Systems
Spring 2020

Module 6.5
Something of a Midterm Review

John Zahorjan

Modules

1. Course Introduction
2. Architectural Support for Operating Systems
3. Operating System Components and Structure
4. Processes
5. Threads
6. Synchronization

Labs

- Lab 1
- Lab 2

1. Course Introduction

- What is an OS?
- What does it do?
  - Library-like shared functionality
  - Allocates hardware resources
  - Protection, while allowing apps to exeute directly on hardware when it's safe to do so
- OS abstraction of hardware
  - processes (virtual address spaces)
  - files
  - sockets
  - streams
- OS provides a measure of portability

1. Course Introduction

- Policy / Mechanism separation
- OS and concurrency
  - Why?
  - Why run more than one application concurrently?
  - Why does execution of the OS itself involve concurreny?
- Concurrency vs. Parallelism

2. Architectural Support for Operating Systems

- Why does OS “require” hardware support?
- Hardware support:
  - CPU modes (privileged vs. unprivileged)
  - Privileged instructions
- OS runs first, at boot
- OS established safe execution context for user-level process before dispatching it
  - Establishes memory mapping to limit memory access
  - Establishes CPU mode to prevent execution of privileged instructions
    - Changing execution context is privileged
- Protection violation
  - Hardware exception: mechanism
  - OS handler: policy

2. Architectural Support for Operating Systems

- **ONLY** way to transition CPU from unprivileged to privileged mode is the exception mechanism, implemented in hardware
- Exception mechanism **always branches to a location** stored in a privileged register
  - A user program turn on privilege, for example when it wants to make a system call
  - A user program can branch anywhere it wants
  - A user program can't do both together
- Hardware saves some processor state as part of the transition
  - What state **must** be saved by hardware?
  - Where does it save it?
  - How does it know where to save it?
- "Exception" vs. "interrupt" vs. "trap"

2. Architectural Support for Operating Systems

- The slides show a lot of detail of the x86_64 exception mechanism
- Is it important?
  - Not really
  - But it's kind of interesting
  - Lessons:
    - Need to know what address to transition to
    - Need eventually to get to an event-specific handler routine
    - Need to establish a kernel stack
    - Putting a lot of semantics into the hardware seems like a mistake
      - Modern OS's work around the complicated mechanisms in the x8 architecture
- Mechanism/Policy Dichotomy and Upcalls
  - level that detects event does so
  - it's reaction is as generic as possible – invoke code at the level above

2. Architectural Support for Operating Systems

- Protecting Memory
  - Virtual Address Space – protection via naming
  - Page-level access rights
- Protecting IO devices
  - Privileged instructions
  - VAS
- Making sure the OS will run again, even if app loops
  - Timer
- CPU / IO overlap
  - IO completion interrupts

3. Operating System Components and Structure

- Process concept/component
  - basic operations
- Address space concept/component
  - virtual memory
  - allocation of physical memory
- IO concept/component
  - device drivers and integration with OS
- File systems
  - as storage abstraction
  - as a name space with system-wide scope
- Other components
  - protection; text shell; windowing system; networking stack

3. Operating System Components and Structure

- monolithic structure
  - Pro's
  - Con's
- Purely layered structure
  - Pro's
  - Con's
  - HAL layer survives to this day
- Microkernels
  - Put major functionality is user-level services, minimize kernel code
  - Why?
  - Why not?

3. Operating System Components and Structure

- Support for Virtual Machines
  - Exporting hardware interfaces up as the API and running directly on hardware looking down
  - Exporting something close the hardware interfaces up as the API and making use of a standard OS looking down
  - exporting OS interfaces looking up and running on a standard OS looking down
  - Exokernel: exporting abstractions of hardware devices looking up so that all traditional OS functionality can run inside the user-level process
    - mechanism / policy split
  - QEMU: binary translation

4. Processes

- Processes as abstractions of hardware
- Processes for isolation
  - Unit of failure
- Process = address space + thread + meta-data
- Memory layout: text, data, heap, and stack segments
- Process control blocks and meta-data
  - pid as process name
- PCB data structure – allocation
- PCB chaining – can be blocked on at most one thing at a time, so a single pointer suffices
  - not exactly true, but general idea is right
- Process (thread) states: ready/runnable, running, blocked

4. Processes

- Process creation
  - fork()
    - Why?
    - Policy vs. mechanism...
  - vs. exec()
  - vfork() and COW fork()
- Inheritance of meta-data
  - shells and redirection of input/output
  - pipes
- Inter-process-communication (IPC)
  - command line (invocation) arguments – explicitly passed
  - environment variables – implicitly passed
  - files
  - pipes
  - named pipes / named shared memory regions
  - internet protocols / sockets

4. Processes

- signals - software exception mechanism
  - Why?
  - Separation of mechanism and policy
    - E.g., a zero-divide occurred. What should happen?

- Aggregates of processes
  - Why might you want a level of abtraction above a single process?
    - A “job”?

5. Threads

- Threads vs. processes
- Motivation: “granularity”
- Thread (CPU) execution state
  - PC (next instruction)
  - SP (bottom of this thread’s stack)
  - *other registers*
- Thread state
  - running, runnable, blocked
- Threads/processes and scheduling
  - it depends
  - OS can be oblivious, and just schedule threads equally
  - OS can provide some notion of “job” – an aggregate of threads – and try to treat jobs equally

5. Threads

- A process is created with a single thread
  - A copy of the thread that executed fork() in the parent
- An existing thread can create new threads
- thread_ create() vs. process_fork()
  - No new address space created/copied, but
    unused address space in existing address space found and allocated as stack
  - user-level interface allows creator to point to a method where created thread starts execution
    - but that functionality may be implemented in user-level library code that wraps the actual system call
  - Child thread has a new thread id, but other components of process meta-data are (likely) shared
- who (what code) should decide where new stack is allocated?

5. threads

- kernel threads vs. user threads
  - kernel controls allocation of cores to kernel threads
    - so you need kernel threads
  - “context switching” doesn’t involved anything privileged, though
    - so you can build a user-level library that creates abstract execution contexts (threads) and switches (the core it has) among them
- user-level threads have lower cost operations
  - creation/termination, synchronization operations (lock/unlock, join, etc.)
  - why?
  - why does it matter?
- when a user-level thread blocks, what the kernel blocks is the kernel thread that was running
  - if synchronization variables are implemented at user-level, kernel can’t tell if thread it just blocked holds one or not
  - blocking a thread holding a lock, say, is an unfortunate scheduling choice...

5. threads

- this is a classic policy vs. mechanism confusion issue
  - the kernel necessarily implements the core allocation mechanism
  - it's also making the policy decision of which threads should have cores and which shouldn't
    - the user-level thread package should be making that decision
- Solution approach: scheduler activations
  - when kernel adds or removes a core allocation to/from a processs, it does an "upcall" to allow user-level code to make the scheduling decisions
    - which user-level threads should have cores and which shouldn't
  - how must the upcall mechanism work?
    - how does the kernel know what code in the app should be run?
    - how does it cause that code to be run?

6. Synchronization

- Temporal relationships
  - A and B are simultaneous/unordered/concurrent is neither “A is before B” nor “B is before A” is guaranteed
- What is a critical section?
- How do you recognize that some block of code is a critical section?
  - concurrent...
  - read-modify-write of...
  - variable that is shared
- Eliminating races: mutual exclusion
  - Ensure that at most one thread executes the critical section code at a time

6. Synchronization

- Critical section mechanism:
  - mutual exclusion
  - progress
  - bounded waiting
  - performance
- Locks
  - a mechanism with acquire/release (lock/unlock) interface
  - we build more sophisticated mechanisms on top of locks
- Spinlocks
  - locks where a thread attempting an acquire() just keeps attempting until it succeeds

6. Synchronization

- Spinlocks require some hardware assistance
- Need to atomically read and write some shared location
  - If read-then-write isn't atomic, many threads can do the read and all get a value that indicates the lock is free
  - To prevent that, we need to guarantee that if we read the value indicating free that we overwrite it with a value indicating not-free before another thread can read it
- Example hardware instructions
  - test-and-set(address): atomically return the value read and set the memory location to 1, no matter what the value was
  - compare-and-swap(address, r1, r2): atomically compare the value at the address with the contents of register 1, and iff they're equal then write the contents of register 2 to the address
- See lecture slides for spinlock implementation using test-and-set

6. Synchronization

- Are spin locks a good idea?
  - If the lock is normally free, you acquire use of the critical section in just a couple of instructions
  - Releasing the lock is just a write to memory
- On the other hand...
  - if lock is busy:
    - it could be the lock holder will give it up soon and you'll get it. Check..
    - it could be the lock holder will give it up soon but there are many threads waiting for it. Not so check.
    - it could be the critical section is really long, so you don't expect the lock to become free soon. Uncheck.
    - it could be the critical section is very short and there aren't typically a lot of threads contending for the lock but, by extreme bad luck, the thread holding the lock was preempted and isn't even running right now. Very uncheck.
- Rule of thumb
  - Use spinlocks only for extremely short code critical section code sequences
  - For example, to implement blocking locks