Today’s agenda

• Administrivia
  – overloading, tweaked course schedule
• Architecture and OS’s
  – what an OS needs from hardware
Tweaked course schedule

- I changed the midterm dates
  - to better fit with the project due dates
  - to better fit with natural lecture topic groupings
  - the new dates (won't be moved again):
    - Midterm 1: Friday January 26th
    - Midterm 2: Friday Feb 16th

- An outline of course topics and readings is up
  - make sure you've done the readings before the lecture!

Architecture affects the OS

- Operating system functionality is dictated, at least in part, by the underlying hardware architecture
  - includes instruction set (synchronization, I/O, …)
  - also hardware components like MMU or DMA controllers

- Architectural support can vastly simplify (or complicate!) OS tasks
  - e.g.: early PC operating systems (DOS, MacOS) lacked support for virtual memory, in part because at that time PCs lacked necessary hardware support
Architectural Features affecting OS’s

- These features were built primarily to support OS’s:
  - timer (clock) operation
  - synchronization instructions (e.g. atomic test-and-set)
  - memory protection
  - I/O control operations
  - interrupts and exceptions
  - protected modes of execution (kernel vs. user)
  - protected instructions
  - system calls (and software interrupts)

Protected Instructions

- some instructions are restricted to the OS
  - known as protected or privileged instructions
- e.g., only the OS can:
  - directly access I/O devices (disks, network cards)
    - why?
  - manipulate memory state management
    - page table pointers, TLB loads, etc.
    - why?
  - manipulate special ‘mode bits’
    - interrupt priority level
    - why?
  - halt instruction
    - why?
OS Protection

- So how does the processor know if a protected instruction should be executed?
  - the architecture must support at least two modes of operation: kernel mode and user mode
    - VAX, x86 support 4 protection modes
    - why more than 2?
  - mode is set by status bit in a protected processor register
    - user programs execute in user mode
    - OS executes in kernel mode (OS == kernel)

- Protected instructions can only be executed in the kernel mode
  - what happens if user mode executes a protected instruction?

Crossing Protection Boundaries

- So how do user programs do something privileged?
  - e.g., how can you write to a disk if you can’t do I/O instructions?
- User programs must call an OS procedure
  - OS defines a sequence of system calls
  - how does the user-mode to kernel-mode transition happen?
- There must be a system call instruction, which:
  - causes an exception (throws a software interrupt), which vectors to a kernel handler
  - passes a parameter indicating which system call to invoke
  - saves caller’s state (regs, mode bit) so they can be restored
  - OS must verify caller’s parameters (e.g. pointers)
  - must be a way to return to user mode once done
A Kernel Crossing Illustrated

user mode

Netscape: read( )

trap to kernel mode; save app state

trap handler

find read( ) handler in vector table

read( ) kernel routine

kernel mode

restore app state, return to user mode, resume

System Call Issues

• What would happen if kernel didn’t save state?
• Why must the kernel verify arguments?
• How can you reference kernel objects as arguments or results to/from system calls?
Memory Protection

- OS must protect user programs from each other
  - maliciousness, ineptitude
- OS must also protect itself from user programs
  - integrity and security
  - what about protecting user programs from OS?
- Simplest scheme: base and limit registers
  - are these protected?

![Diagram showing base and limit registers]

More sophisticated memory protection

- coming later in the course
- virtual memory
  - paging, segmentation
  - page tables, page table pointers
  - translation lookaside buffers (TLBs)
OS control flow

- after the OS has booted, all entry to the kernel happens as the result of an event
  - event immediately stops current execution
  - changes mode to kernel mode, event handler is called
- kernel defines handlers for each event type
  - specific types are defined by the architecture
    - e.g.: timer event, I/O interrupt, system call trap
  - when the processor receives an event of a given type, it
    - transfers control to handler within the OS
    - handler saves program state (PC, regs, etc.)
    - handler functionality is invoked
    - handler restores program state, returns to program

Interrupts and Exceptions

- Two main types of events: interrupts and exceptions
  - exceptions are caused by software executing instructions
    - e.g. the x86 ‘int’ instruction
    - e.g. a page fault, write to a read-only page
    - an expected exception is a “trap”, unexpected is a “fault”
  - interrupts are caused by hardware devices
    - e.g. device finishes I/O
    - e.g. timer fires
I/O Control

• Issues:
  – how does the kernel start an I/O?
    • special I/O instructions
    • memory-mapped I/O
  – how does the kernel notice an I/O has finished?
    • polling
    • interrupts

• Interrupts are basis for asynchronous I/O
  – device performs an operation asynch to CPU
  – device sends an interrupt signal on bus when done
  – in memory, a vector table contains list of addresses of kernel routines to handle various interrupt types
    • who populates the vector table, and when?
  – CPU switches to address indicated by vector specified by interrupt signal

Timers

• How can the OS prevent runaway user programs from hogging the CPU (infinite loops?)
  – use a hardware timer that generates a periodic interrupt
  – before it transfers to a user program, the OS loads the timer with a time to interrupt
    • “quantum”: how big should it be set?
  – when timer fires, an interrupt transfers control back to OS
    • at which point OS must decide which program to schedule next
    • very interesting policy question: we’ll dedicate a class to it

• Should the timer be privileged?
  – for reading or for writing?
Synchronization

• **Interrupts cause a wrinkle:**
  – may occur any time, causing code to execute that interferes with code that was interrupted
  – OS must be able to synchronize concurrent processes

• **Synchronization:**
  – guarantee that short instruction sequences (e.g. read-modify-write) execute atomically
  – one method: turn off interrupts before the sequence, execute it, then re-enable interrupts
    • architecture must support disabling interrupts
  – another method: have special complex atomic instructions
    • read-modify-write
    • test-and-set
    • load-linked store-conditional