CSE 153
Design of Operating Systems
Fall 2016
Lecture 2: Architectural Support for Operating Systems
Why Start With Architecture?

- Recall: Key goals of an OS are 1) to enable virtualization/abstraction; 2) to enforce protection and resource sharing; and 3) manage concurrency
  - If done well, applications can be oblivious to HW details
    » e.g., fread() assumes nothing about underlying storage

- Architectural support can greatly simplify – or complicate – OS tasks
  - Easier for OS to implement a feature if supported by hardware
  - OS needs to implement everything hardware doesn’t
Architectural support of OS

- As OS evolves, architecture evolves to provide better support (e.g., CPU, MMU, DMA, ...)

Diagram:
- Operating System on Hardware
- Operating System on Hardware
Architectural Features for OS

- Features that directly support the OS include
  - Protection
    » Protection (kernel/user mode)
    » Protected instructions
    » Memory protection
  - Event-driven interface
    » System calls
    » Interrupts and exceptions
    » Timer (clock)
  - Handling concurrency
    » I/O control and operation
    » Synchronization
Review: Computer Organization

Program Counter

CPU Instructions Fetch Exec

Branch Address

Select PC

New PC

opcode

opcode

+4
Types of Arch Support for OS

● Manipulating privileged machine state
  ♦ Protected instructions
  ♦ Manipulate device registers, TLB entries, etc.

● Generating and handling “events”
  ♦ Interrupts, exceptions, system calls, etc.
  ♦ Respond to external events
  ♦ CPU requires software intervention to handle fault or trap

● Mechanisms to handle concurrency
  ♦ Interrupts, atomic instructions, memory consistency model
Types of Arch Support

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.

- Generating and handling “events”
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap

- Mechanisms to handle concurrency
  - Interrupts, atomic instructions
## Protected Instructions

- A subset of instructions of every CPU is restricted to use only by the OS
  - Known as protected (privileged) instructions
- Only the operating system can
  - Directly access I/O devices (disks, printers, etc.)
    - Security, fairness (why?)
  - Manipulate memory management state
    - Page table pointers, page protection, TLB management, etc.
  - Manipulate protected control registers
    - Kernel mode, interrupt level
  - Halt instruction (why?)
OS Protection

- How does HW know if protected instr. can be executed?
  - Architecture must support (at least) two modes of operation: kernel mode and user mode
    - VAX, x86 support four modes; earlier archs (Multics) even more
    - Why?

- Mode kept in a status bit in a protected control register
  - User programs execute in user mode
  - OS executes in kernel mode (OS == “kernel”)

- Protected instructions only execute in kernel mode
  - CPU checks mode bit when protected instruction executes
  - Attempts to execute in user mode are detected and prevented
  - Need for new protected instruction?
    - Setting mode bit
CPU Modes/Privileges (x86)

- Ring 0  →  Kernel Mode
- Ring 3  →  User Mode
Memory Protection

- OS must be able to protect programs from each other
  - OS must protect itself from user programs
  - May or may not protect user programs from OS
- Memory management hardware provides memory protection mechanisms
  - Base and limit registers
  - Page table pointers, page protection, TLB
  - Virtual memory
  - Segmentation
- Manipulating memory management hardware uses protected (privileged) operations
Example memory protection

Physical Memory

<table>
<thead>
<tr>
<th>INSTR</th>
<th>DATA</th>
<th>HEAP</th>
<th>STACK</th>
</tr>
</thead>
</table>

Base  Bounds

OK?

Yes

Continue

No

Exception

Memory Reference

CPU
Types of Arch Support

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.

- **Generating and handling “events”**
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap

- Mechanisms to handle concurrency
  - Interrupts, atomic instructions
Events

- An event is an “unnatural” change in control flow
  - Events immediately stop current execution
  - Changes mode, context (machine state), or both

- The kernel defines a handler for each event type
  - Event handlers always execute in kernel mode
  - The specific types of events are defined by the machine

- OS model is sleeping beauty
  - Once the system is booted, all entry to the kernel occurs as the result of an event
    » In effect, the operating system is one big event handler
Categorizing Events

- Two kinds of events: synchronous and asynchronous
- Sync events are caused by executing instructions
  - Example?
- Async events are caused by an external event
  - Example?
handleTimerInterrupt() {
    ...
}

handleDivideByZero() {
    ...
}

handleSystemCall() {
    ...
}
Categorizing Events

- Two *kinds* of events: **synchronous** and **asynchronous**
  - Sync events are caused by executing instructions
  - Async events are caused by an external event

- Two *reasons* for events: **unexpected** and **deliberate**
  - Unexpected events are, well, unexpected
    - Example?
  - Deliberate events are scheduled by OS or application
    - Why would this be useful?
Categorizing Events

- This gives us a convenient table:

<table>
<thead>
<tr>
<th></th>
<th>Unexpected</th>
<th>Deliberate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>fault</td>
<td>syscall trap</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>interrupt</td>
<td>software interrupt</td>
</tr>
</tbody>
</table>

- Terms may be used slightly differently by various OSes, CPU architectures…
- Software interrupt – a.k.a. async system trap (AST), async or deferred procedure call (APC or DPC)
- Will cover faults, system calls, and interrupts next
Faults

- Hardware detects and reports “exceptional” conditions
  - Page fault, unaligned access, divide by zero

- Upon exception, hardware “faults” (verb)
  - Must save state (PC, regs, mode, etc.) so that the faulting process can be restarted
Handling Faults

- Some faults are handled by “fixing” the exceptional condition and returning to the faulting context
  - Page faults cause the OS to place the missing page into memory
  - Fault handler resets PC of faulting context to re-execute instruction that caused the page fault

- Some faults are handled by notifying the process
  - Fault handler changes the saved context to transfer control to a user-mode handler on return from fault
  - Handler must be registered with OS
  - Unix signals or NT user-mode Async Procedure Calls (APCs)
    » SIGALRM, SIGHUP, SIGTERM, SIGSEGV, etc.
Handling Faults

● The kernel may handle unrecoverable faults by killing the user process
  ◆ Program fault with no registered handler
  ◆ Halt process, write process state to file, destroy process
  ◆ In Unix, the default action for many signals (e.g., SIGSEGV)

● What about faults in the kernel?
  ◆ Dereference NULL, divide by zero, undefined instruction
  ◆ These faults considered fatal, operating system crashes
  ◆ Unix panic, Windows “Blue screen of death”
    » Kernel is halted, state dumped to a core file, machine locked up
## Categorizing Events

<table>
<thead>
<tr>
<th></th>
<th>Unexpected</th>
<th>Deliberate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>fault</td>
<td>syscall trap</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>interrupt</td>
<td>software interrupt</td>
</tr>
</tbody>
</table>
System Calls

- For a user program to do something “privileged” (e.g., I/O) it must call an OS procedure
  - Known as crossing the protection boundary, or a protected procedure call

- Hardware provides a system call instruction that:
  - Causes an exception, which invokes a kernel handler
  - Passes a parameter determining the system routine to call
  - Saves caller state (PC, regs, mode) so it can be restored
    - Why save mode?
  - Returning from system call restores this state
System Call

User mode

Kernel mode

emacs: read()

Trap to kernel mode, save state

Trap handler

Find read handler

read() kernel routine

Restore state, return to user level, resume execution
System Call Questions

● What if the kernel executes a system call?

● How to reference kernel objects as arguments or results to/from system calls?
  - A naming issue
  - Use integer object handles or descriptors
    » E.g., Unix file descriptors
    » Only meaningful as parameters to other system calls
  - Also called capabilities (more later when we do protection)
  - Why not use kernel addresses to name kernel objects?
CPU Modes/Privileges

- System call
  - Ring 3 → Ring 0
Another view

Kernel Stack

Kernel Code

User Stack

User Code

Address Space

0x00000000

0xC0000000

0xFFFFFFFF

0x00000000

1G

3G

SP2

PC2

SP1

PC1
System Call Questions

- There are hundreds of syscalls. How do we let the kernel know which one we intend to invoke?
  - Before issuing `int $0x80` or `sysenter`, set `%eax` with the syscall number

- System calls are like function calls, but how to pass parameters?
  - Just like calling convention in syscalls, typically passed through `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp

- How to reference kernel objects (e.g., files, sockets)?
  - Naming problem – an integer mapped to a unique object
    » `int fd = open("file"); read(fd, buffer);`
  - Why can’t we reference the kernel objects by memory address?
System calls in xv6

- Look at trap.h and trap.c
  - Interrupt handlers are initialized in two arrays (idt and vectors)
    » Tvinit() function does the initialization
  - Syscalls have a single trap handler (T_SYSCALL, 64)
  - Trap() handles all exceptions, including system calls
    » If the exception is a system call, it calls syscall()

- Keep digging from there to understand how system calls are supported
  - You will be adding a new system call in Lab 1
## Categorizing Events

<table>
<thead>
<tr>
<th></th>
<th>Unexpected</th>
<th>Deliberate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>fault</td>
<td>syscall trap</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>interrupt</td>
<td>software interrupt</td>
</tr>
</tbody>
</table>
Interrupts

- Interrupts signal asynchronous events
  - I/O hardware interrupts
  - Software and hardware timers

- Two flavors of interrupts
  - Precise: CPU transfers control only on instruction boundaries
  - Imprecise: CPU transfers control in the middle of instruction execution
    » What does that mean?
  - OS designers like precise interrupts, CPU designers like imprecise interrupts
    » Why?
The timer is critical for an operating system.

It is the fallback mechanism by which the OS reclaims control over the machine:
- Timer is set to generate an interrupt after a period of time
  - Setting timer is a privileged instruction
- When timer expires, generates an interrupt
- Handled by kernel, which controls resumption context
  - Basis for OS scheduler (more later…)

Prevents infinite loops:
- OS can always regain control from erroneous or malicious programs that try to hog CPU

Also used for time-based functions (e.g., \texttt{sleep()})
I/O Control

- I/O issues
  - Initiating an I/O
  - Completing an I/O

- Initiating an I/O
  - Special instructions
  - Memory-mapped I/O
    - Device registers mapped into address space
    - Writing to address sends data to I/O device
I/O using Interrupts

- Interrupts are the basis for asynchronous I/O
  - OS initiates I/O
  - Device operates independently of rest of machine
  - Device sends an interrupt signal to CPU when done
  - OS maintains a vector table containing a list of addresses of kernel routines to handle various events
  - CPU looks up kernel address indexed by interrupt number, context switches to routine
I/O Example

1. Ethernet receives packet, writes packet into memory
2. Ethernet signals an interrupt
3. CPU stops current operation, switches to kernel mode, saves machine state (PC, mode, etc.) on kernel stack
4. CPU reads address from vector table indexed by interrupt number, branches to address (Ethernet device driver)
5. Ethernet device driver processes packet (reads device registers to find packet in memory)
6. Upon completion, restores saved state from stack
Interrupt Questions

- Interrupts halt the execution of a process and transfer control (execution) to the operating system
  - Can the OS be interrupted? (Consider why there might be different interrupt levels)

- Interrupts are used by devices to have the OS do stuff
  - What is an alternative approach to using interrupts?
  - What are the drawbacks of that approach?
Synchronization

- Interrupts cause difficult problems
  - An interrupt can occur at any time
  - A handler can execute that interferes with code that was interrupted

- OS must be able to synchronize concurrent execution

- Need to guarantee that short instruction sequences execute atomically
  - Disable interrupts – turn off interrupts before sequence, execute sequence, turn interrupts back on
  - Special atomic instructions – read/modify/write a memory address, test and conditionally set a bit based upon previous value
    » XCHG on x86
Summary

- **Protection**
  - User/kernel modes
  - Protected instructions

- **System calls**
  - Used by user-level processes to access OS functions
  - Access what is “in” the OS

- **Exceptions**
  - Unexpected event during execution (e.g., divide by zero)

- **Interrupts**
  - Timer, I/O
Next Time...

- Processes
- Project:
  - Continue to get familiar with the environment
    » In particular, Chapter 0
  - Read the system call/interrupt chapter in the xv6 book (Chapter 3)
  - If you have time, work through at least some of the booting sequence tutorial
    » Read appendix A and B in xv6 book
  - Ask questions on Piazza