Why Architecture?

- Operating systems mediate between applications and the physical hardware of the computer
  - Key goals of an OS are to enforce protection and resource sharing
  - If done well, applications can be oblivious to HW details
  - Unfortunately for us, the OS is left holding the bag

- The details of the HW architecture matter
  - Small differences in hardware may have large repercussions
  - Early PC operating systems did not support virtual memory, due in large part to the fact that the H/W provided no support
  - Sun 1 machines used an entirely separate processor just to manage virtual memory! (M68000 chips did not have hardware support for virtual memory.)
Types of Arch Support

- Providing protected machine state
  - Device registers, memory management state, etc.
  - Manipulated with privileged instructions
  - Allows OS to enforce protection

- Generating and handing asynchronous signals
  - Method to respond to external events
  - Interrupts, exceptions, system calls, etc.
  - Allows OS to enforce sharing

- Mechanisms to handle concurrency
  - *More on this in a few lectures…*
Architectural Features

- Privileged instructions
- Protection modes (user/kernel)
- Memory protection mechanisms
- Interrupts and exceptions
- Timer (clock)
- I/O control and operation
- System calls
- Synchronization primitives (e.g., atomic instructions)
Privileged Instructions

- A select few CPU instructions available only to OS
  - Allows access to protected state
  - Perform global operations

- For example, only the OS should be able to:
  - Directly access I/O devices (disks, printers, etc.)
    » Allows OS to enforce security and fairness
  - Manipulate memory management state
    » E.g., page tables, protection bits, TLB entries, etc.
  - Adjusted protected control registers
    » Change between user/kernel mode or raise/lower interrupt level
  - Execute the halt instruction
How does CPU know when an instruction is OK?

- Architecture must support (at least) two modes of operation: kernel and user mode
  - VAX, x86 support four modes (rings); early archs even more
  - Why? Protect the OS from itself (software engineering)
- Mode is indicated by a status bit in a protected control register
- User programs execute in a user mode
- OS executes in kernel mode (OS == “kernel”)

Protected instructions only execute in kernel mode

- CPU checks mode bit when protected instruction executes
- Setting mode bit must be a protected instruction
- Attempts to execute in user mode are detected and prevented
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 protection
  - Base and limit registers
  - Segmentation
  - Page table pointers, page protection, TLB
  - Virtual memory
- Manipulating memory management hardware uses protected (privileged) instructions
Events

- An event is an “unnatural” change in control flow
  - Events immediately stop current execution
  - Change 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
- Events are the only entry into the kernel
  - System boot is the first event, loads OS
  - Once in user mode, only way back to kernel is an event
- The OS is really just one big event handler
Two kinds of events: exceptions and interrupts

Exceptions are caused by executing instructions
- Exceptions indicate software intervention is needed

Interrupts are caused by external event
- Usually hardware devices, timers, etc.

Events can be both expected and unexpected
- Unexpected, a.k.a. asynchronous events
- Sometimes events are caused deliberately
### Types of Events

<table>
<thead>
<tr>
<th></th>
<th>Unexpected</th>
<th>Deliberate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions</td>
<td>Fault</td>
<td>Syscall trap</td>
</tr>
<tr>
<td>Interrupts</td>
<td>(H/W) Interrupt</td>
<td>Software interrupt</td>
</tr>
</tbody>
</table>

- Each type has its own special term
  - Terms vary slightly across architectures, OSes…
  - Software interrupt is also known as async system trap (AST), async or deferred procedure call (A/DPC)
Faults

- Hardware detects and reports “exceptional” conditions
  - Page fault, unaligned memory access, divide by zero
- Upon detecting an exception, CPU “faults”
  - Needs to save context (PC, registers, mode bit, etc.) so that faulting instruction can be resumed
- OSes use page faults to implement many things
  - Debugging, garbage collection, copy-on-write, etc.
- Faults are strictly a performance optimization
  - Not needed for correctness
  - Extra code could be used to test for fault conditions, but at significant performance expense
Handling Faults

- Some faults are silently fixed by the OS
  - Once exceptional condition is dealt with, process resumes
    » E.g., page faults cause the OS to load the missing page
  - Fault handler reloads the context of the faulting process and reissues the instruction that caused the fault

- Others are passed on to the application itself
  - OS fault handler modifies the saved context to transfer control to a user-mode handler on return from fault
  - Applications must pre-register handlers with the OS
  - Unix signals or NT user-mode Async Procedure Calls (APCs)
    » SIGALRM, SIGHUP, SIGTERM, etc.
Fatal Faults

- Some faults cannot be properly handled
  - OS has no default handler for the particular fault
  - Application also failed to register a handler
  - App is halted, state dumped to a (core) file, process destroyed

- Hopefully such faults do not occur inside the kernel
  - Unhandled faults in the kernel itself are fatal; OS crashes
    » Divide by zero, dereference NULL pointer, invalid instruction, etc.
  - Unix kernel panic, Windows “Blue screen of death”
    » Kernel is halted, state dumped to a core file, machine locked up
System Calls

- What if an application needs something privileged?
  - Must ask the OS to perform instruction on its behalf
  - Needs to make a protected procedure call or cross the protection boundary

- Application issues a special system call instruction
  - Raises an exception, which is trapped by a kernel handler
  - Application passes a parameter which handler reads to determine which system call routine to invoke
  - Kernel saves caller context and restores on completion of call

- Requires architectural support
  - Validate input parameters (e.g., buffer addresses)
  - Restore saved state, reset mode bit, resume execution
Anatomy of a System call

App: read()

User Mode

Kernel Mode

System call handler

Specify read system call; 
trap to kernel mode; 
save context

Lookup call parameter; 
Invoke read() routine

Restore context; 
return to user mode; 
Resume execution

read() procedure
System Call Complications

- What if the kernel invokes a system call?
- What if user mode tries to ‘return’ from a system call?
- How does the kernel return parameters to user mode?
  - Why not simply pass the address of kernel memory?
  - The first of many naming issues we’ll face
  - OS uses integer object handles or descriptors
    - Only meaningful as parameters to other system calls
    - E.g., Unix file descriptors
  - Kernel’s job to ensure descriptors are valid
    - Will allow us to enforce resource protection
    - In this context, descriptors are called capabilities
    - More later in the term…
Interrupts

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

- Two different implementation models
  - **Precise**: CPU transfers control on instruction boundaries
  - **Imprecise**: Control potentially transferred in the middle of instruction execution
    - What does that mean?
  - OS designers like precise interrupts, hardware folks like to build imprecise interrupts
    - Why?
Timers

- Operating system timer is a critical building block
  - Many resources are time-shared; e.g., CPU
  - Allows OS to prevent infinite loops

- Fallback mechanism by which OS regains control
  - Timer is set to generate an interrupt at periodic intervals
  - When timer expires, generates an interrupt
  - Handled by kernel, which controls resumption context
    » Basis for OS scheduler; more later...
  - Setting (and clearing) a timer is a privileged instruction

- Also used to provide time services for applications
  - E.g., sleep() system call
For Next Class...

- Homework 1 due in one week
- Read Chapter 3
- Email cs120s1@ieng6.ucsd.edu with project groups
- Log into your account and check out Lab 0
  - Also due in one week