Administrivia

Lab 0
- Due next Thursday (09/09) noon EST, done individually
- Overview session this Friday 4-5 pm (encourage you to start now)

Project groups
- If you don’t have a group, search for teammates on Campuswire

Office hours
- All Zoom links posted on Campuswire

Lecture questions
- Please feel free to interrupt and ask questions anytime during the lecture
Why Start With Hardware?

OS functionality fundamentally depends upon the architectural features

- Key goals of an OS are to enforce protection and resource sharing
- If done well, applications can be oblivious to HW details

Architectural support can greatly simplify or complicate OS tasks

- Early DOS, MacOS lacked virtual memory in part because the h/w did not support it
- Early Sun 1 computers used two M68000 CPUs to implement virtual memory
Architectural Features for OS

Features that directly support the OS include

- Bootstrapping (Lab 0)
- Protection (kernel/user mode)
- Protected instructions
- Memory protection
- System calls
- Interrupts and exceptions
- Timer
- I/O control and operation
- Synchronization
Types of Arch Support

I. Manipulating privileged machine state
   - Protected instructions
   - Manipulate device registers, TLB entries, etc.
What Is Inside A Computer?

A Motherboard (Intel DH87MC)

Source: Intel® Desktop Board DH87MC Technical Product Specification
Thought Experiment: A World of Anarchy

Any program in the system can...
- Directly access I/O devices
- Write anywhere in memory
- Read content from any memory address
- Execute machine halt instruction

Do you trust such system?
- use Facebook app in this system
- use Banking app in this system

Challenge: protection
- How to execute a program with restricted privilege?
A Solution

How can we implement execution with limited privilege?
- Execute each program instruction through a simulator (OS)
- If the instruction is permitted, do the instruction
- Otherwise, stop the process
- Basic model in Javascript and other interpreted languages

How do we go faster?
- Observation: most instructions are perfectly safe!
- Run the *unprivileged* code directly on the CPU
- Do the check in h/w
H/W Support: Dual-Mode Operation in CPU

**User mode**
- Limited privileges
- Only those granted by the operating system kernel

**Kernel mode**
- Execution with the full privileges of the hardware
- Read/write to any memory, access I/O device, read/write disk sector, send/read packet

On the x86, the Current Privilege Level (CPL) in the CS register

On the MIPS, the status register
A Simple Model of a CPU

Basic operation

- Fetch next instruction → Decode instruction → Execute

![Diagram of CPU operations]

New PC

Program Counter

CPU Instructions Fetch and Execute

Branch Address

Select PC

Opcode

Instruction stream

Program counter

load
add
sub
cmp
jne 36
store
mov
call
jmp 44
...

4
8
12
16
20
24
28
32
36
...
A CPU with **Dual-Mode Operation**

1. **Handler PC**
2. **Branch Address**
3. **New PC**

- **Select PC**
- **Program Counter**
  - **CPU Instructions Fetch and Execute**

- **New Mode**
  - **Select Mode**

**Check if an instruction can be executed**

**Change mode upon certain instructions (e.g., trap)**
Protected Instructions

A subset of instructions 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?)
INVLPG—Invalidate TLB Entries

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En Mode</th>
<th>64-Bit Mode</th>
<th>Comp/Exec Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01/7</td>
<td>INVLPG m</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Invalidate TLB entries for page containing m.</td>
</tr>
</tbody>
</table>

**NOTES:**
* See the IA-32 Architecture Compatibility section below.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**
Invalidate any translation lookaside buffer (TLB) entries specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes all TLB entries for that page.¹

The **INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL must be 0 to execute this instruction.**

The INVLPG instruction normally flushes TLB entries only for the specified page; however, in some cases, it may flush more entries, even the entire TLB. The instruction is guaranteed to invalidate only TLB entries associated with the current PCID. (If PCIDs are disabled — CR4.PCID = 0 — the current PCID is 000H.) The instruction also invalidates any global TLB entries for the specified page, regardless of PCID.

For more details on operations that flush the TLB, see “MOV—Move to/from Control Registers” and Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

This instruction's operation is the same in all non-64-bit modes. It also operates the same in 64-bit mode, except if the memory address is in non-canonical form. In this case, INVLPG is the same as a NOP.

**IA-32 Architecture Compatibility**
The INVLPG instruction is implementation dependent, and its function may be implemented differently on different families. See the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for implementation differences.
Beyond Dual-Mode Operations

(Modern) CPU may provide more than 2 privilege levels

- Called hierarchical protection domains or protection rings
- x86 supports four levels:
  - bottom 2 bits (CPL) of the CS register indicate execution privilege
  - ring 0 (CPL=00) is kernel mode, ring 3 (CPL=11) is user mode
- Multics provides 8 levels of privilege
- ARMv7 CPUs in modern smartphones have 8 different protection levels

Why?

- Protect the OS from itself (software engineering)
- reserved for vendor, e.g., virtualization
Memory Protection

Why?
- 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
- Raises question of whether programs should trust the OS
- Untrusted operating systems? (Intel SGX)

Memory management hardware (MMU) provides the mechanisms
- Base and limit registers
- Page table pointers, page protection, segmentation, TLB
- Manipulating the hardware uses protected (privileged) operations
Simple Memory Protection

Memory access bounds check

Processor

510: <500, 100> ✔
610: >500, 100

Base

Bound

Raise Exception

Physical Address

Physical Memory

Base

Base + Bound

Raise Exception

Bound

Base

510

610

500, 100

1: [500, 600]
Simple Memory Protection

Memory access bounds check

Problem

- Inflexible:
  - fixed allocation, difficult to expand heap and stack
- Memory fragmentation
- Physical memory
  - require changes to mem instructions each time the program is loaded
Idea: Virtual Address

Programs refer to memory by virtual addresses
- start from 0
- illusion of “owning” the entire memory address space

The virtual address is translated to physical address
- upon each memory access
- done in hardware (MMU) using a table
- table setup by the OS
Types of Arch Support

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

II. Generating and handling “events”
   - Interrupts, exceptions, system calls, etc.
   - Respond to external events
   - CPU requires software intervention to handle fault or trap
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
- The specific types of events are defined by the architecture
  - e.g., timer event, I/O interrupt, system call trap
- In effect, the operating system is one big event handler
OS Control Flow

After OS booting, **all entry to kernel is a result of some event**
- event immediately stops current execution
- changes mode to kernel mode
- invoke a piece of code to handle event (event handler)

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 executes core OS functionality, e.g., writing data to disk
- handler **restores program state**, returns to program
Illustration of OS Control Flow

User Process

Save caller state

Trap
Mode bit = 0

Kernel Mode
Mode bit = 0

Handler

Restore caller state

Resume Process

User Mode
Mode bit = 1

Return
Mode bit = 1

Kernel boundary
Event: Interrupt vs. Exceptions

Two kinds of events, \textit{interrupts} and \textit{exceptions}

\textbf{Interrupts are caused by an external event (asynchronous)}
- Device finishes I/O, timer expires, etc.

\textbf{Exceptions are caused by executing instructions (synchronous)}
- x86 \texttt{int} instruction, page fault, divide by zero, etc.
Interrupts

Interrupts signal asynchronous events
- Indicates some device needs services
- I/O hardware interrupts
- Software and hardware timers

Why?
- A computer is more than CPU
  - keyboard, disk, printer, camera, etc.
- These devices occasionally need attention, but cannot predict when!
One Solution: Polling

CPU periodically checks if each device needs service

**Problem?**
- takes CPU time when there are no events pending
- reduce checking frequency → longer response time
+ can be efficient if events arrive rapidly

“Polling is like picking up your phone every few seconds to see if you have a call…”
Idea: Give Each Device a Wire (Interrupt Line)

I/O devices wired with Interrupt Request Lines (IRQs)

Problem?
- CPU might get interrupted non-stop
- Some device may overwhelm CPU
- Critical interrupts delayed
- Interrupts handling inflexible ("hard-coded")

"Interrupts are like waiting for the phone to ring."
I/O devices have (unique or shared) **Interrupt Request Lines** (IRQs)

IRQs are mapped by special hardware to **interrupt vectors**, and passed to the CPU

This hardware is called a **Programmable Interrupt Controller (PIC)**
The “Interrupt Controller”

PIC: Programmable Interrupt Controller (8259A)
- Responsible for telling the CPU *when and which* device wishes to ‘interrupt’
- Has 16 wires to devices (IRQ0 – IRQ15)

PIC translates IRQs to CPU interrupt *vector number*
- Vector number is signaled over INTR line
- *In Pintos:* IRQ0...15 delivered to vector 32...47 (*src/threads/interrupt.c*)

Interrupts can have varying priorities
- PIC also needs to prioritize multiple requests

Possible to “mask” (disable) interrupts at PIC or CPU
Software Interface: *Interrupt Vector Table*

A data structure to associate interrupt requests with handlers

- each entry is called an interrupt vector (specifies the address of the handler)
- architecture-specific implementation

```c
handleTimerInterrupt() {
    ...
}

handleDivideByZero() {
    ...
}

handleSystemCall() {
    ...
}
```
Software Interface: *Interrupt Vector Table*

A data structure to associate interrupt requests with handlers

- each entry is called an interrupt vector (specifies the address of the handler)
- architecture-specific implementation

In x86 called *Interrupt Descriptor Table (IDT)*

- supports 256 interrupts, so the IDT contains 256 entries
- each entry specifies the address of the handler plus some flags
- programmed by the OS
  - In Pintos: `make_intr_gate` ([src/threads/interrupt.c](src/threads/interrupt.c))
Interrupt Usage Scenario: Timer

Timer is critical for an operating system

It is the fallback mechanism for OS to reclaim 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., `sleep()`)
Timer in Pintos

Needed in Pintos Lab 1’s **Alarm Clock** exercise

```c
/* Sets up the timer to interrupt TIMER_FREQ times per second, 
   and registers the corresponding interrupt. */

void 
timer_init (void)
{
    pit_configure_channel (0, 2, TIMER_FREQ);
    intr_register_ext (0x20, timer_interrupt, "8254 Timer");
}

/* Timer interrupt handler. */
static void timer_interrupt (struct intr_frame *args UNUSED)
{
    ticks++;
    thread_tick ();
}

/* Called by the timer interrupt 
handler at each timer tick. */
void thread_tick (void)
{
    struct thread *t = thread_current ();
    /* Update statistics. */
    if (t == idle_thread)
        idle_ticks++;
    else
        kernel_ticks++;
    /* Enforce preemption. */
    if (++thread_ticks >= TIME_SLICE)
        intr_yield_on_return ();
}
```
Interrupt Usage Scenario 2: I/O Control

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

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 an interrupt vector table (IVT)
- CPU looks up IVT by interrupt number, context switches to routine
Event: Interrupt vs. Exceptions

Two kinds of events, interrupts and exceptions

Interrupts are caused by an external event (asynchronous)
- Device finishes I/O, timer expires, etc.

Exceptions are caused by executing instructions (synchronous)
- x86 int instruction, page fault, divide by zero, etc.
- a deliberate exception is a “trap”, unexpected exception is a “fault”
- CPU requires software intervention to handle a fault or trap
Deliberate Exception: Trap

A trap is an intentional software-generated exception
- the main mechanism for programs to interact with the OS
- On x86, programs use the `int` instruction to cause a trap
- On ARM, SVC instruction

Handler for trap is defined in interrupt vector table
- Kernel chooses one vector for representing system call trap
- e.g., `int $0x80` is used to in Linux to make system calls
- Pintos uses `int $0x30` for system call trap
System Call Trap

For a user program to “call” OS service
- Known as crossing the protection boundary, or protected control transfer

The system call instruction
- Causes an exception, which vectors to a kernel handler
- Passes a parameter determining the system routine to call

```
movl $20, %eax  # Get PID of current process
int $0x80     # Invoke system call!
# Now %eax holds the PID of the current process
```

- Saves caller state (PC, regs, mode) so it can be restored
- Returning from system call restores this state

Requires architectural support to:
- Restore saved state, reset mode, resume execution
System Call

Firefox: read()

Trap to kernel mode, save state

Trap handler

Find read handler in vector table

read() kernel routine

Restore state, return to user level, resume execution

User mode

Kernel mode
LINUX System Call Quick Reference
Jialong He
Jialong.he@gmail.com
http://www.eecs.berkeley.edu/~jialong_he

Introduction
System call is the service provided by Linux kernel. In C programming, it often uses functions defined in the library which provides a wrapper for many system calls. Manpage section 2 provides more information about system calls. To get an overview, use "man 2 man" in a command shell.

It is also possible to invoke syscall() function directly. Each system call has a function number defined in "syscalls.h" or "<sysctl.h>". Internally, system call is invoked by software interrupt (SIG) to transfer control to the kernel. System call table is defined in Linux kernel source file: "arch/X86/kernel/entry.S".

System Call Example

```c
#include <syscalls.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
    long pid, tid;
    /* direct system call */
    // SYS_getpid (func no. is 20) */
    // ID1 = syscall(SYS_getpid);
    printf("%s\n", ID1);
    /* ------------------------------- */
    /* "like" wrapped system call */
    // SYS_getpid (func no. is 20) */
    // ID2 = getpid();
    printf("%s\n", ID2);
    return(0);
}
```

System Call Quick Reference

<table>
<thead>
<tr>
<th>No</th>
<th>Func Name</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>exit</code></td>
<td>Terminate the current process</td>
<td><em>kernel.c</em></td>
</tr>
<tr>
<td>2</td>
<td><code>fork</code></td>
<td>Create a child process</td>
<td><em>kernel.c</em></td>
</tr>
<tr>
<td>3</td>
<td><code>read</code></td>
<td>Read from a file descriptor</td>
<td><em>linux_read.c</em></td>
</tr>
<tr>
<td>4</td>
<td><code>write</code></td>
<td>Write to a file descriptor</td>
<td><em>linux_write.c</em></td>
</tr>
<tr>
<td>5</td>
<td><code>open</code></td>
<td>Open a file or device</td>
<td><em>linux_open.c</em></td>
</tr>
<tr>
<td>6</td>
<td><code>close</code></td>
<td>Close a file descriptor</td>
<td><em>linux_close.c</em></td>
</tr>
<tr>
<td>7</td>
<td><code>waitpid</code></td>
<td>Wait for process termination</td>
<td><em>kernel.c</em></td>
</tr>
<tr>
<td>8</td>
<td><code>open</code></td>
<td>Create a file or device (&quot;man 2 open&quot; for information)</td>
<td><em>linux_open.c</em></td>
</tr>
<tr>
<td>9</td>
<td><code>link</code></td>
<td>Make a new name for a file</td>
<td><em>kernel.c</em></td>
</tr>
<tr>
<td>10</td>
<td><code>unlink</code></td>
<td>Delete a name and possibly the file it refers to</td>
<td><em>kernel.c</em></td>
</tr>
<tr>
<td>11</td>
<td><code>execve</code></td>
<td>Execute program</td>
<td><em>kernel.c</em></td>
</tr>
<tr>
<td>12</td>
<td><code>chdir</code></td>
<td>Change working directory</td>
<td><em>linux.c</em></td>
</tr>
<tr>
<td>13</td>
<td><code>time</code></td>
<td>Get time in seconds</td>
<td><em>time.c</em></td>
</tr>
<tr>
<td>14</td>
<td><code>malloc</code></td>
<td>Create a special or ordinary file</td>
<td><em>malloc.c</em></td>
</tr>
<tr>
<td>15</td>
<td><code>close</code></td>
<td>Change permissions of a file</td>
<td><em>linux.c</em></td>
</tr>
<tr>
<td>16</td>
<td><code>rename</code></td>
<td>Change ownership of a file</td>
<td><em>linux.c</em></td>
</tr>
<tr>
<td>17</td>
<td><code>stat</code></td>
<td>Get file status</td>
<td><em>stat.h</em></td>
</tr>
<tr>
<td>18</td>
<td><code>lseek</code></td>
<td>Reposition read/write file offset</td>
<td><em>linux_file.c</em></td>
</tr>
<tr>
<td>19</td>
<td><code>getuid</code></td>
<td>Get process identification</td>
<td><em>security.h</em></td>
</tr>
<tr>
<td>20</td>
<td><code>mount</code></td>
<td>Mount filesystems</td>
<td><em>fs.h</em></td>
</tr>
<tr>
<td>21</td>
<td><code>umount</code></td>
<td>Unmount filesystems</td>
<td><em>fs.h</em></td>
</tr>
<tr>
<td>22</td>
<td><code>setuid</code></td>
<td>Set real user ID</td>
<td><em>security.h</em></td>
</tr>
<tr>
<td>23</td>
<td><code>getuid</code></td>
<td>Get real user ID</td>
<td><em>security.h</em></td>
</tr>
<tr>
<td>24</td>
<td><code>seteuid</code></td>
<td>Set system time and date</td>
<td><em>security.h</em></td>
</tr>
<tr>
<td>25</td>
<td><code>nice</code></td>
<td>Allows a parent process to control the execution of a child process</td>
<td><em>kernel.c</em></td>
</tr>
<tr>
<td>26</td>
<td><code>alarm</code></td>
<td>Set an alarm clock for delivery of a signal</td>
<td><em>syscalls.h</em></td>
</tr>
<tr>
<td>27</td>
<td><code>fstat</code></td>
<td>Get file status</td>
<td><em>linux.h</em></td>
</tr>
<tr>
<td>28</td>
<td><code>pause</code></td>
<td>Suspend process until signal</td>
<td><em>arch/i386/kernel/pause.c</em></td>
</tr>
<tr>
<td>29</td>
<td><code>setfset</code></td>
<td>Set file access and modification times</td>
<td><em>linux.c</em></td>
</tr>
<tr>
<td>30</td>
<td><code>gettimeofday</code></td>
<td>Check user's permissions for a file</td>
<td><em>linux.c</em></td>
</tr>
<tr>
<td>31</td>
<td><code>nice</code></td>
<td>Change process priority</td>
<td><em>security.h</em></td>
</tr>
<tr>
<td>32</td>
<td><code>setpgid</code></td>
<td>Update the process group</td>
<td><em>linux.c</em></td>
</tr>
<tr>
<td>33</td>
<td><code>kill</code></td>
<td>Send signal to a process</td>
<td><em>security.h</em></td>
</tr>
<tr>
<td>34</td>
<td><code>rename</code></td>
<td>Change the name or location of a file</td>
<td><em>linux.c</em></td>
</tr>
<tr>
<td>35</td>
<td><code>mknod</code></td>
<td>Create a directory or special file</td>
<td><em>security.h</em></td>
</tr>
<tr>
<td>36</td>
<td><code>rmdir</code></td>
<td>Remove a directory</td>
<td><em>security.h</em></td>
</tr>
<tr>
<td>37</td>
<td><code>dup2</code></td>
<td>Duplicate an open file descriptor</td>
<td><em>linux.c</em></td>
</tr>
<tr>
<td>38</td>
<td><code>pipe</code></td>
<td>Create an interprocess channel</td>
<td><em>arch/i386/kernel_pipe.c</em></td>
</tr>
<tr>
<td>39</td>
<td><code>times</code></td>
<td>Get process times</td>
<td><em>security.h</em></td>
</tr>
<tr>
<td>40</td>
<td><code>getrho</code></td>
<td>Get real group ID</td>
<td><em>security.h</em></td>
</tr>
<tr>
<td>41</td>
<td><code>getpeerid</code></td>
<td>Get effective user ID</td>
<td><em>security.h</em></td>
</tr>
<tr>
<td>42</td>
<td><code>getppid</code></td>
<td>Get effective group ID</td>
<td><em>security.h</em></td>
</tr>
</tbody>
</table>
System Call Questions

What would happen if the kernel did not save state?

What if the kernel executes a system call?

How to reference kernel objects as arguments or results to/from syscalls?

- A naming issue
- Use integer object handles or descriptors
  - E.g., Unix file descriptors, Windows HANDLEs
  - Only meaningful as parameters to other system calls
- Also called capabilities (more later when we cover protection)
- Why not use kernel addresses to name kernel objects?
Unexpected Exception: 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

Faults are not necessarily “bad”
- Modern OSes use virtual memory faults for many functions
  • debugging, end-of-stack, garbage collection, copy-on-write

Fault exceptions are essentially a performance optimization
- Could detect faults by inserting extra instructions into code (at a significant performance penalty)
Handling Faults

Some faults are handled by “fixing”…
- “Fix” the exceptional condition and return to the faulting context
- Page faults cause the OS to place the missing page into memory
- Fault handler resets pc 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
- Handler must be registered with OS
- Unix signals or Win user-mode Async Procedure Calls (APCs)
  - SIGALRM, SIGHUP, SIGTERM, SIGSEGV, etc.
Handling Faults (2)

Kernel may handle unrecoverable faults by killing the 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
Types of Arch Support

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

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

III. Mechanisms to support synchronization
   - Interrupt disabling/enabling, atomic instructions
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
  - \texttt{xchg} instruction on x86
Summary

Protection
- User/kernel modes
- Protected instructions

Interrupts
- Timer, I/O

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)

<table>
<thead>
<tr>
<th></th>
<th>Unexpected</th>
<th>Deliberate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions (sync)</td>
<td>fault</td>
<td>syscall trap</td>
</tr>
<tr>
<td>Interrupts (async)</td>
<td>interrupt</td>
<td>software interrupt</td>
</tr>
</tbody>
</table>
Next Time…

Read Chapters 4-6 (Processes)

Lab 0