CS 153
Design of Operating Systems

Winter 2016

Lecture 2: Intro and Architectural Support for Operating Systems

Administrivia

- Course website is up
  - http://www.cs.ucr.edu/~zhiyunq/teaching/cs153/
    - » Check the website for slides
  - Piazza link: https://piazza.com/ucr/winter2016/cs153/home
    - » Also posted on course webpage
    - » Enroll yourself (let me know if you have issues)

- Project group formation
  - Preferably pick a partner in the same lab (coordination)
  - You need to signed up before class next Friday

- Reminder: 4% for extra credit and class participation
  - 2% extra for not using the slack days (at all)

Brief History of OS design

In the beginning…

- OSes were runtime libraries
  - The OS was just code you linked with your program and loaded into the computer
  - First computer interface was switches and lights, then punched tape and cards
- Batch systems were next
  - OS was permanently stored in primary memory
  - It loaded a single job (card reader, mag tape) into memory
  - Executed job, created output (line printer)
  - Loaded the next job, repeat…
  - Card readers, line printers were slow, and CPU was idle while they were being used
  - MS-DOS: single job at a time

Spooling

- The bottleneck of slow I/O and idling CPU motivated development of spooling (Simultaneous Peripheral Operation On-Line)
  - Use faster I/O to hide the latency of slower I/O
    - » Copy documents to printer buffer so printer can work at its own rate and free the CPU
  - But, CPU still idle when job reads/writes to disk

Multiprogramming

- Multiprogramming increased system utilization
  - Keeps multiple runnable jobs loaded in memory
  - Overlaps I/O processing of a job with computation of another
  - Benefits from I/O devices that can operate asynchronously
  - Requires the use of interrupts (from I/O) and DMA
  - Requires memory protection and sharing
  - Optimizes system throughput (number of jobs finished in a given amount of time) at the cost of response time (time until a particular job finishes)

Timesharing

- Timesharing supports interactive use of computer by multiple users
  - Terminals give the illusion that each user has own machine
  - Optimizes response time (time to respond to an event like a keystroke) at the cost of throughput
  - Based on timeslicing – dividing CPU time among the users
  - Enabled new class of applications – interactive!
  - Users now interact with viewers, editors, debuggers
- The MIT Multics system (mid-late 60s) was an early, aggressive timesharing system
- Unix and Windows are also timesharing systems…

Distributed Operating Systems

- Distributed systems facilitate use of geographically distributed resources
  - Machine connected by wires
- Supports communication between parts of a job or different jobs on different machines
  - Interprocess communication
- Sharing of distributed resources, hardware, and software
  - Exploit remote resources
- Enables parallelism, but speedup is not the goal
  - Goal is communication

Parallel Operating Systems

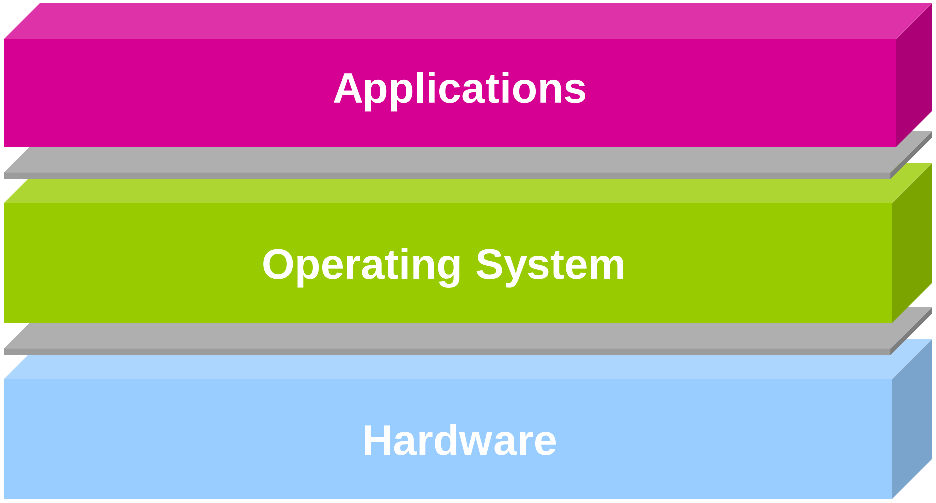
- Support parallel applications trying to get speedup of computationally complex tasks across multiple CPUs
- Requires basic primitives for dividing single task into multiple parallel activities
- Supports efficient communication among activities
- Supports synchronization of activities to coordinate data sharing
- Early parallel systems used dedicated networks and custom CPUs, now common to use networks of high-performance PC/workstations

Embedded Operating Systems

- Decreased cost of processing makes computers ubiquitous
  - Your car has dozens of computers in it
  - Think of everything that has electric motor in it, and now imagine that it also has a computer
- Each embedded application needs its own OS
  - Smart phones
  - Smart home, smart grid
- Very soon
  - Your house will have 100s of embedded computers in it
  - Your electrical lines and airwaves will serve as the network
  - All devices will interact as a distributed system

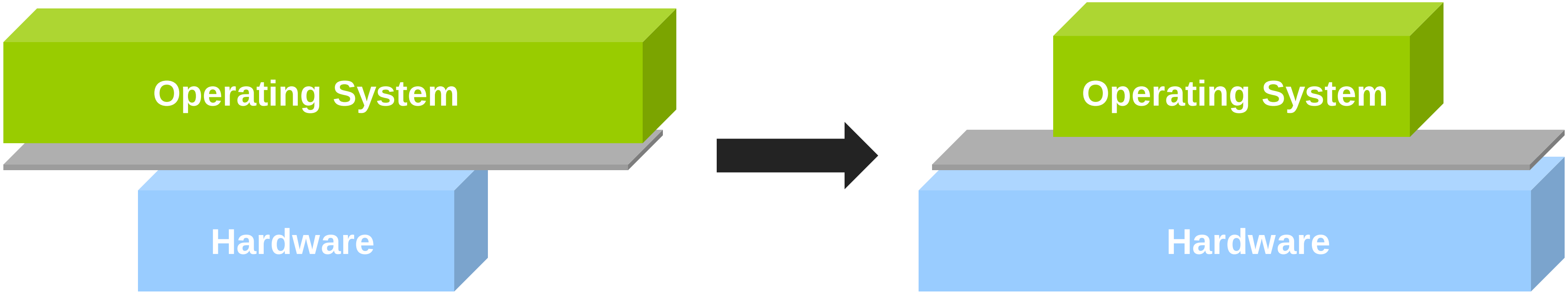
What is an operating system?

- OS is "*all the code that you didn't have to write*" to implement your application
- OS is "*code for all features not offered by hardware*"



Architectural support of OS

- As OS evolves, complex tasks are pushed down to the hardware (e.g., CPU, MMU) – hence the architectural support

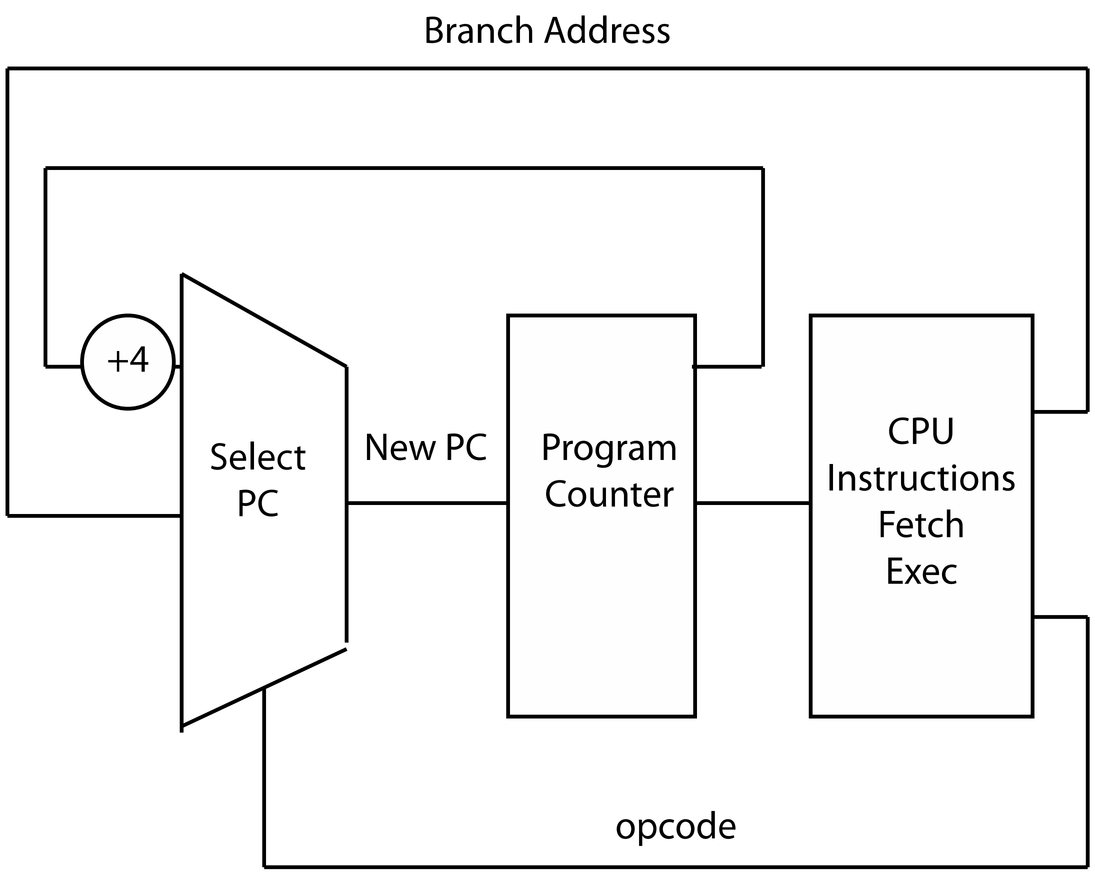


Why Start With Architecture?

- 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

Review: Computer Organization



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

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

Protected Instructions

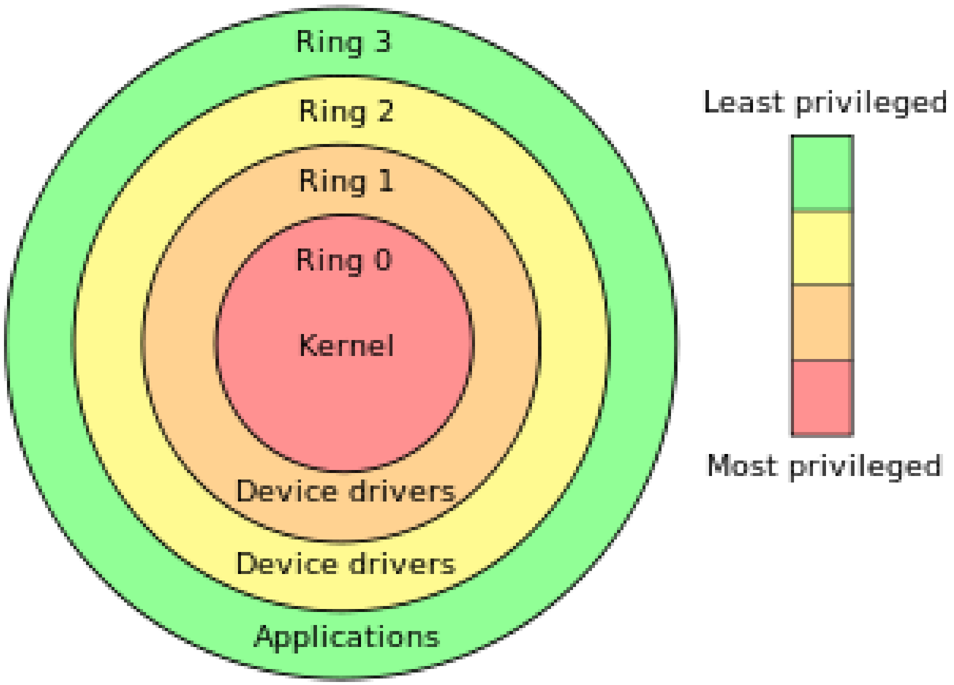
- A subset of instructions of every CPU is restricted to use only by the OS
  - Known as protected (privileged) instructions
- Only the operating system can
  - Directly access I/O devices (disks, printers, etc.)
    - » Security, fairness (why?)
  - Manipulate memory management state
    - » Page table 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 (See next slide)
    - » VAX, x86 support four modes; earlier archs (Multics) even more
    - » Why?
  - Mode is indicated by 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

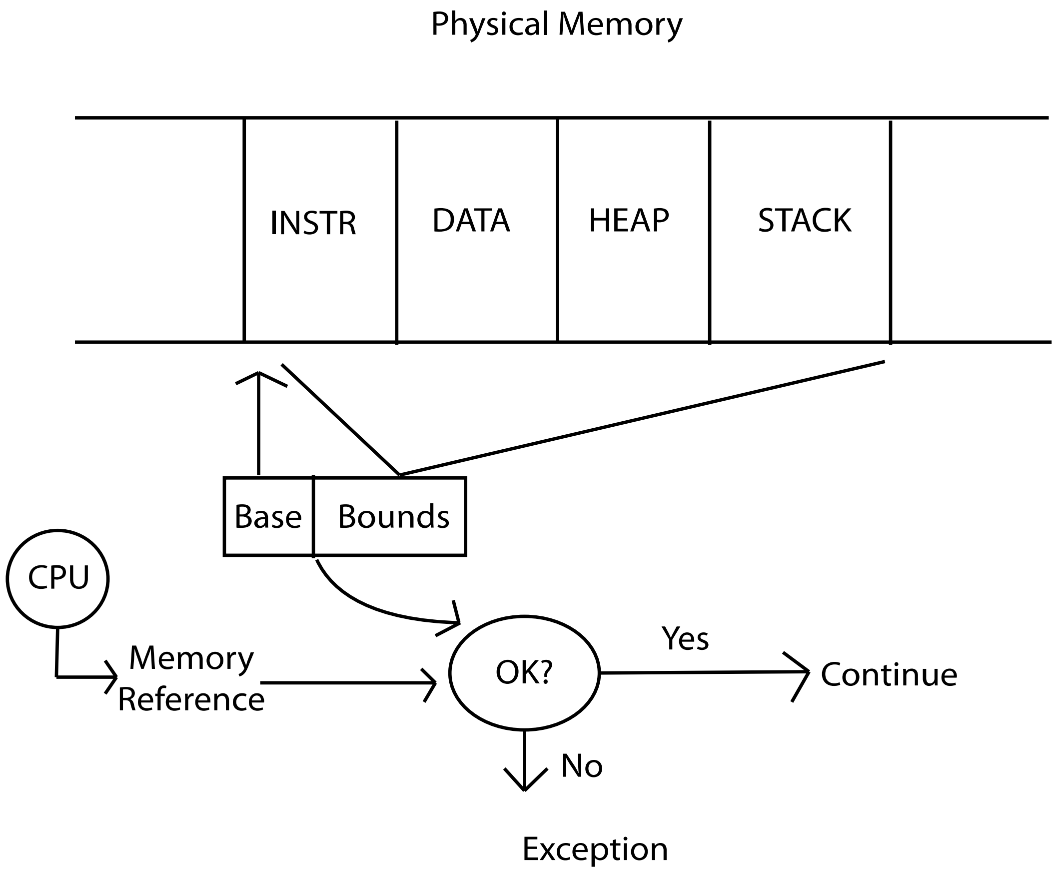
- Ring 0 → Kernel Mode
- Ring 3 → User Mode



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

Base and Bound Example



Types of Arch Support

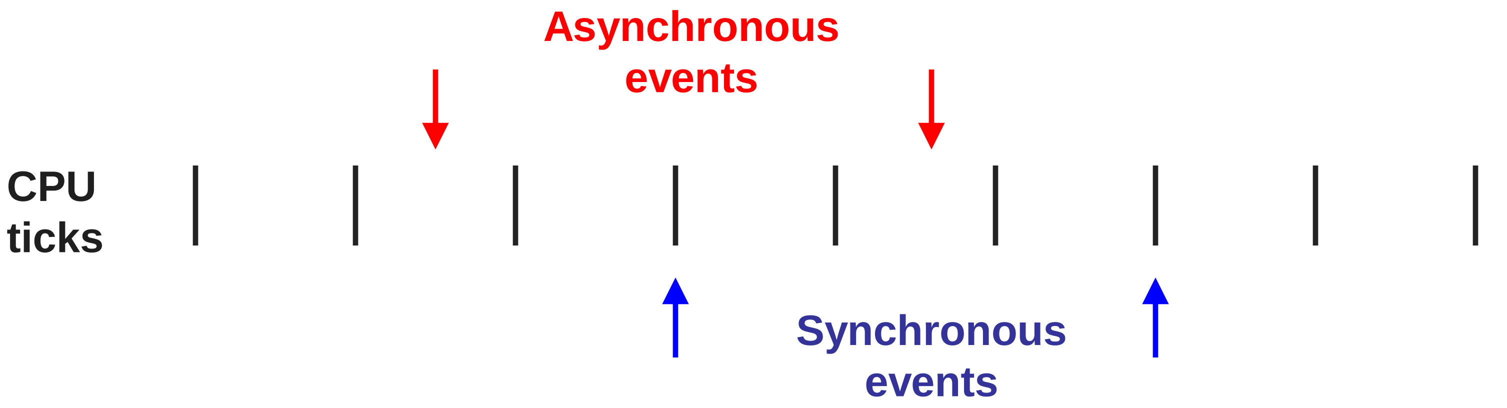
- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.
- Generating and handling "events"
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap
- Mechanisms to handle concurrency
  - Interrupts, atomic instructions

Events

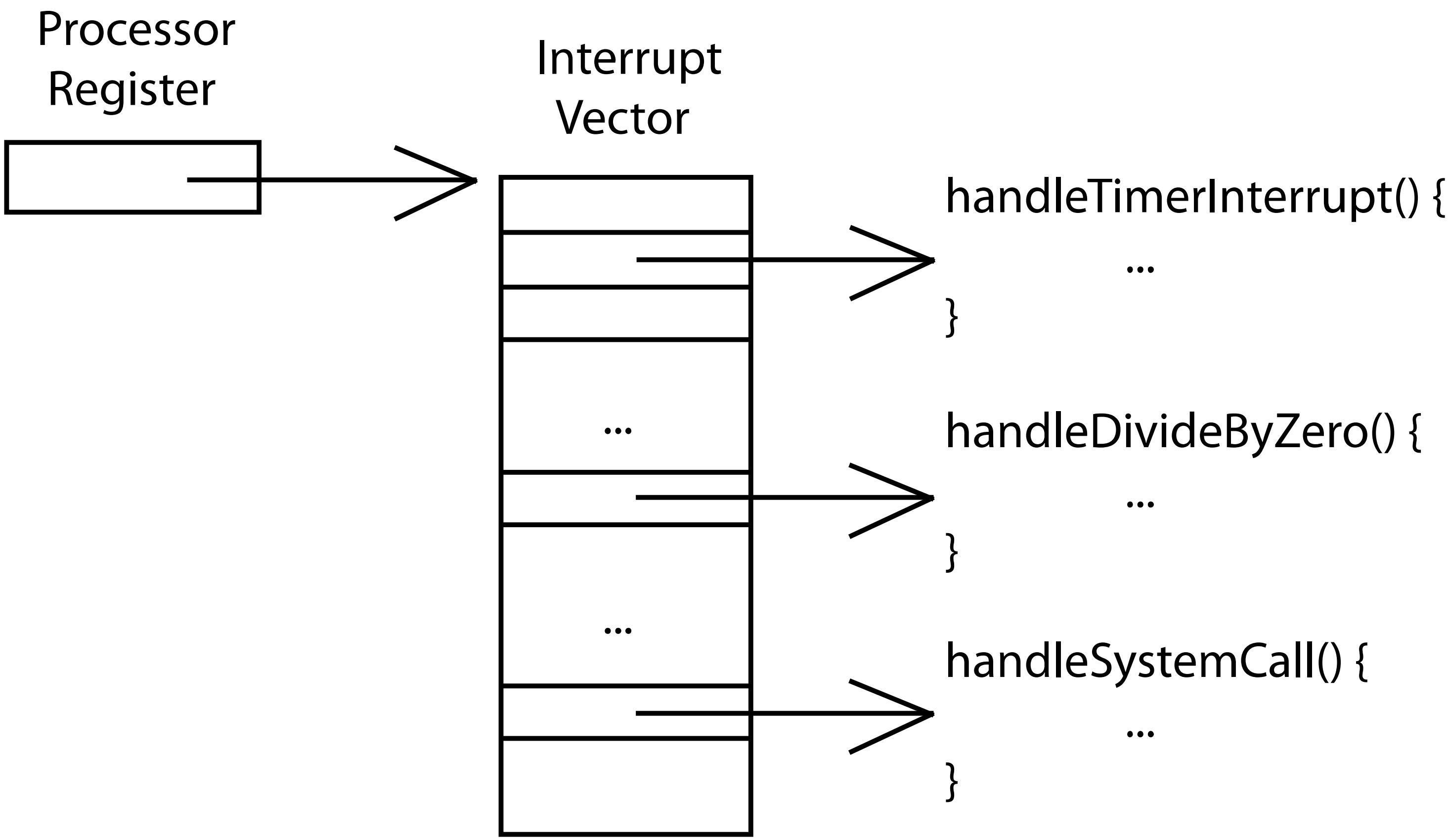
- An event is an “unnatural” change in control flow
  - Events immediately stop current execution
  - Changes mode, context (machine state), or both
- The kernel defines a handler for each event type
  - Event handlers always execute in kernel mode
  - The specific types of events are defined by the machine
- 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?



Interrupt Handler Illustration



In PintOS, they are done in “threads/intr-stubs.S, threads/interrupt.c”

Summary

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

Next Time...

- Processes
  - Read Chapter 3