ECEN 449 – Microprocessor System Design

Hardware-Software Communication

Objectives of this Lecture Unit

- Learn basics of Hardware-Software communication
- Memory Mapped I/O
- Polling/Interrupts

Motivation

- Many reasons why we want hardware devices to interact with software programs
  - Input: keyboards, mice, scanners
  - Output: CRT, printer
  - Interact with real world: contact sensor, chemical analyzer, MEMS devices
  - Performance: ASIC/System-on-chip designs
    - Exploit 90-10 rule of software by implementing critical parts of application in HW, non-critical as SW
      - Issue: embedded apps getting more complex, less likely to have one module that consumes most of the execution time.

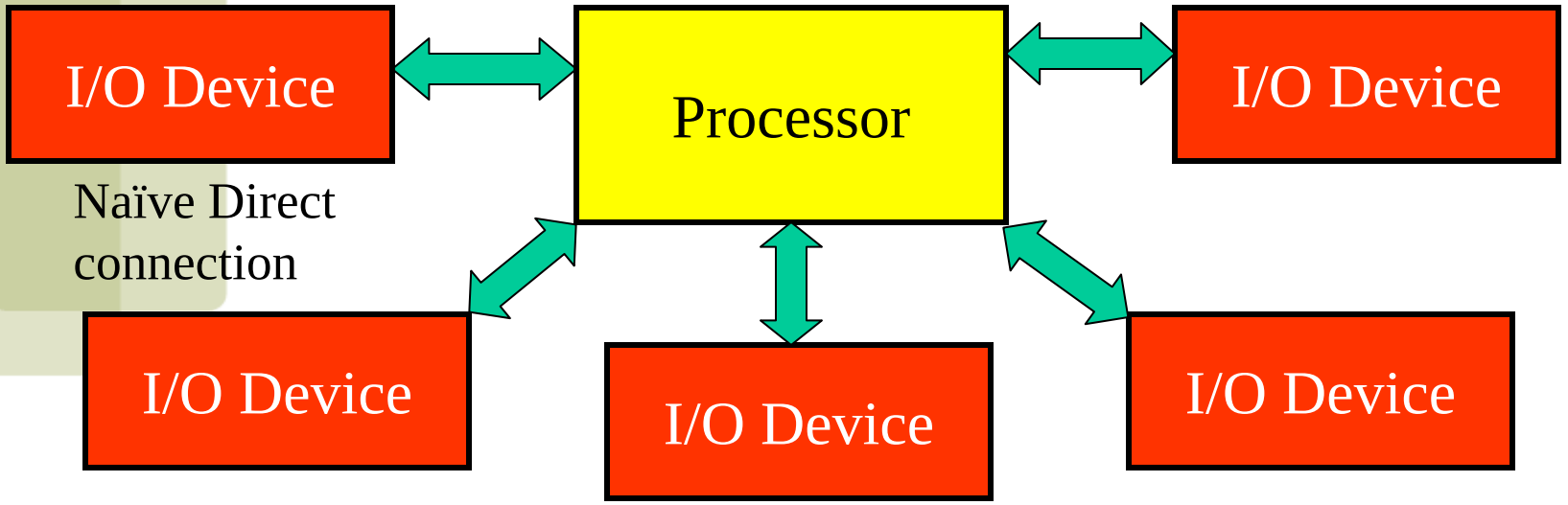
Hardware-Software Interfaces

- What features do we want?

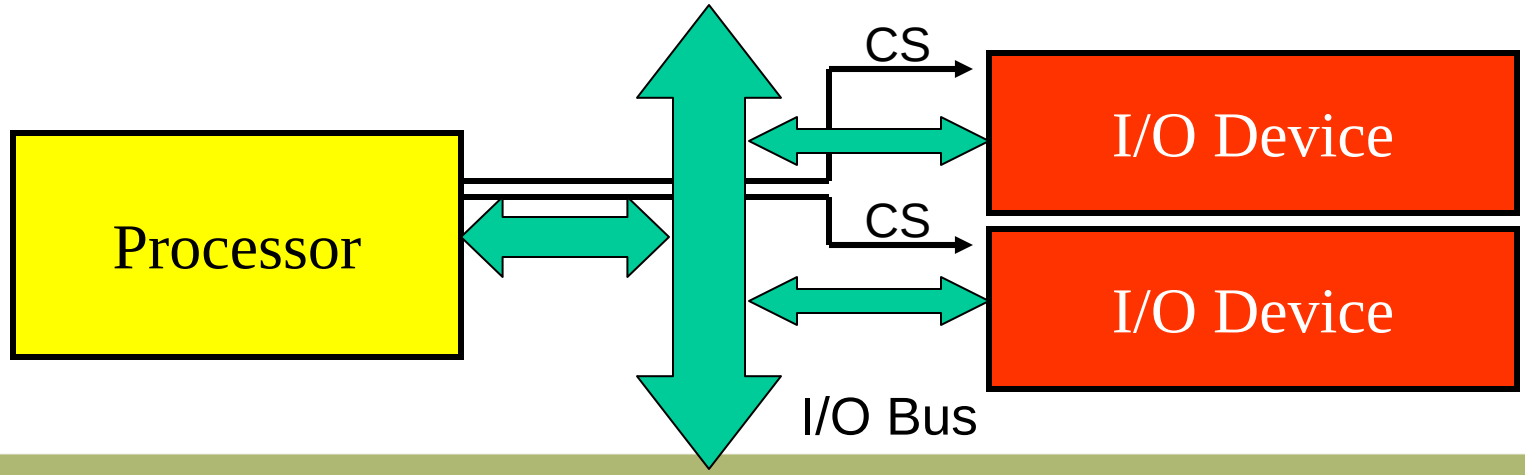
Hardware-Software Interfaces

- What features do we want?
  - Some way to get data between HW and SW components
  - Some way for HW to notify SW of asynchronous events
  - Flexibility
    - Don’t want to have to have special HW in CPU for each device
    - Only load SW into memory for devices that are present
  - Bandwidth
    - Demands may vary wildly: compare keyboard to Gb Ethernet
  - Security:
    - Multiple users/processes may need to share HW

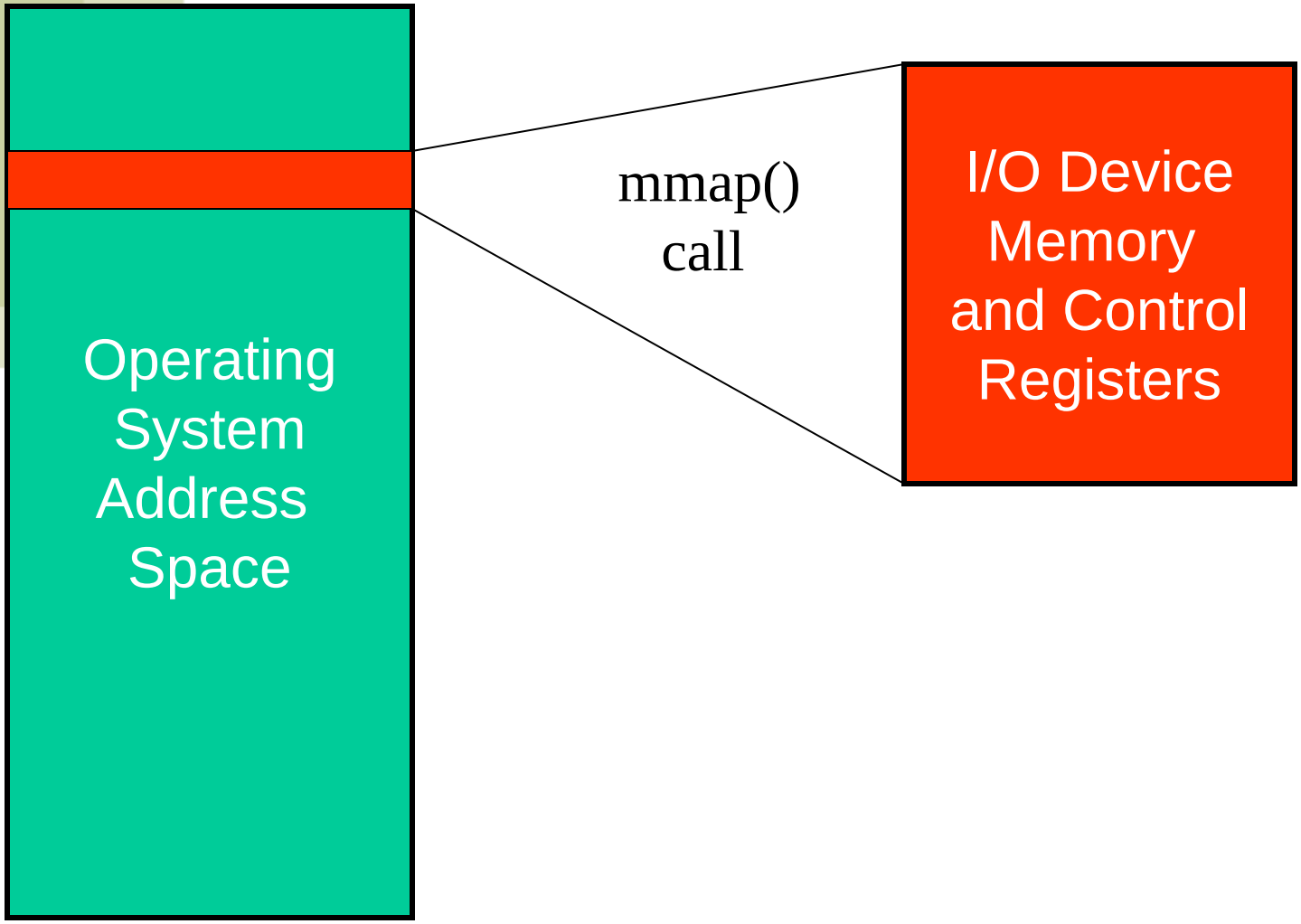
Getting Data to/from CPU: Hardware Side



Naïve Direct connection



The Software Side: Memory-Mapped I/O



Memory-Mapped I/O

- Basic Idea – Make control registers and I/O device memory appear to be part of the system's main memory
  - Reads and writes to that region of the memory are translated by OS/hardware into accesses of hardware device
  - Makes it easy to support variable numbers/types of devices – just map them onto different regions of memory
    - Managing new devices is now like memory allocation
    - Example: accessing memory on a PCMCIA card
      - Once card memory mapped into address space, just hand out pointers like conventional memory
  - Accessing I/O device registers and memory can be done by accessing data structures via the device pointers
    - Most device drivers are now written in C. Memory mapped I/O makes it possible without special changes to the compiler

Memory-Mapped I/O

- Important abstraction
  - A given processor may have multiple busses/connections to devices
    - Ex: PPC in Virtex-II Pro has OCM bus to talk to BlockRAM, PLB bus to talk to other hardware, but they look the same once you call mmap()
- Can interact a bit oddly with caches
  - References to memory-mapped addresses may not go to the right bus if the address is cached
  - Solution: declare any data structures that are memory-mapped I/O regions as volatile
    - Ex: volatile int *region;
    - Tells system that it can't keep the address in the cache

Handling Asynchronous Events

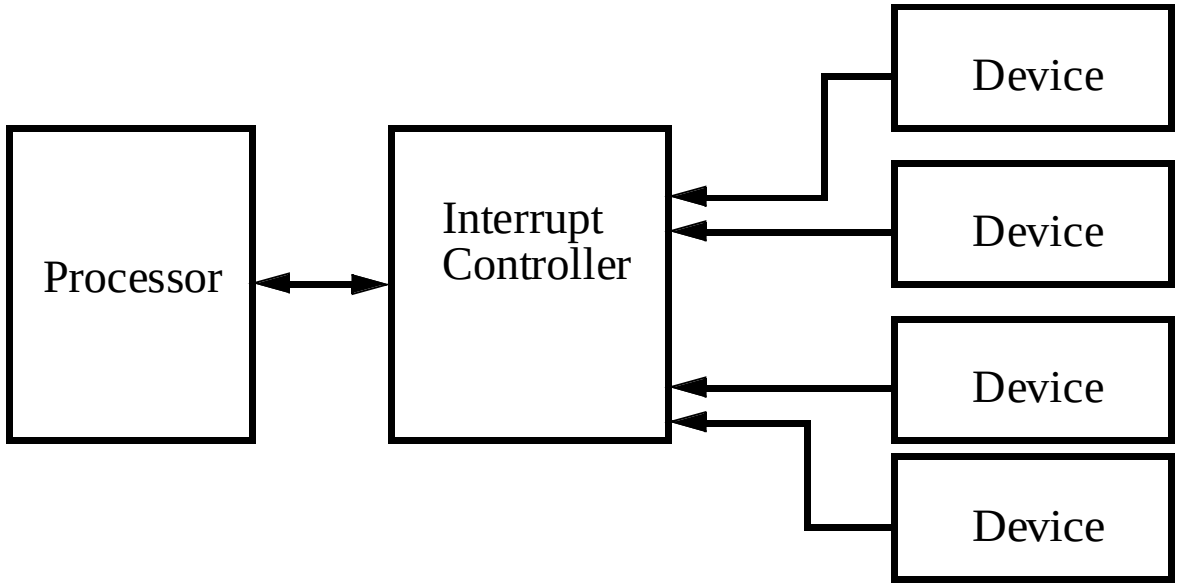
- Issue: External devices operate on different time signals than processor
  - Humans don't have clock rates
- Want:
  - Way to let processor handle events on external hardware that it can't predict
  - Processor to be able to do other things while it's waiting for an event
  - Fast response to events
  - Low overhead (few wasted CPU cycles when no event happens)

Alternative 1: Polling

- Every so often, processor checks each device to see if it has a request
  - Takes CPU time even if no requests pending
  - Tradeoff between overhead and average response time
  - How does software know when to let the system poll?

Alternative 2: Interrupts

- Give each device a wire (interrupt line) that it can use to signal the processor
  - When interrupt signaled, processor executes a routine called an *interrupt handler* to deal with the interrupt
  - No overhead when no requests pending



Polling vs. Interrupts

"Polling is like picking up the phone every few seconds to see if you have a call. Interrupts are like waiting for the phone to ring"

- Interrupts are better if the processor has other work to do and the time to respond to events isn't absolutely critical
- Polling can be better if the processor has nothing better to do and has to respond to an event ASAP

Performance of interrupt hardware is a critical factor on processors for embedded systems.

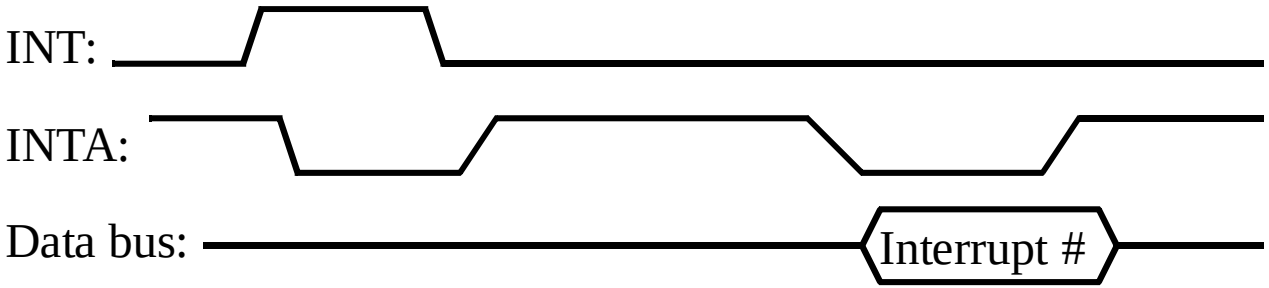
So, What Happens When an Interrupt Occurs?

Acts a lot like a context switch.

- Interrupt controller signals processor that interrupt has occurred, passes interrupt number
- Processor uses interrupt number to determine which handler to start
  - *interrupt vector* associates handlers with interrupts
- Processor halts current program
  - Multi-cycle operations may be halted or squashed and restarted
- Current program state saved (like context switch)
- Processor jumps to interrupt handler
- When interrupt done, program state reloaded and program resumes
- Interrupts are assigned priorities to handle simultaneous interrupts

Example: Interrupts on 80386EX

- 80386 core has one interrupt line, one interrupt acknowledge line
- Interrupt sequence:
  - Interrupt controller raises INT line
  - 80386 core pulses INTA line low, allowing INT to go low
  - 80386 core pulses INTA line low again, signaling controller to put interrupt number on data bus



Using Interrupts in Linux

- OS handles interaction with interrupt hardware
  - Necessary to support multiple processor types
- You need to worry about three things
  1. Writing the interrupt handler
  2. Registering the interrupt handler with the OS
     - Tells the OS that it should run the handler when the interrupt occurs
  3. Interaction between the interrupt handler and user programs
     - Handlers need to be run in very little time
     - Handlers run as part of the operating system
       - Linux doesn't allow them to access user data
     - One general approach: User program does the work, interrupt handler just signals it when it's time to do something

Interrupts in Linux, Part II

- Documentation on interrupts, etc. in Linux: **http://www.xml.com/ldd/chapter/book/index.html**
  - **Entire *Linux Device Drivers* book by Rubini and Corbet is available via the web, free distribution license**
  - **Key chapters for this stuff: Chapter 5 (Blocking I/O) and Chapter 9 (Interrupts)**

Writing an Interrupt Handler

- An interrupt handler is just a C procedure
  - `void int_handler(int irq, void *dev_id, struct pt_regs *regs)`
  - Must match this declaration
    - Limits what data you can pass in to the handler
  - Can't return any value
    - No calling procedure to return value to
- Interrupt handlers have significant limits on what they can do
  - Can't directly read or write data in user space (not associated with a process)
  - Can't do anything involving scheduling new threads or sleeping
  - Need to terminate quickly to free up interrupt hardware

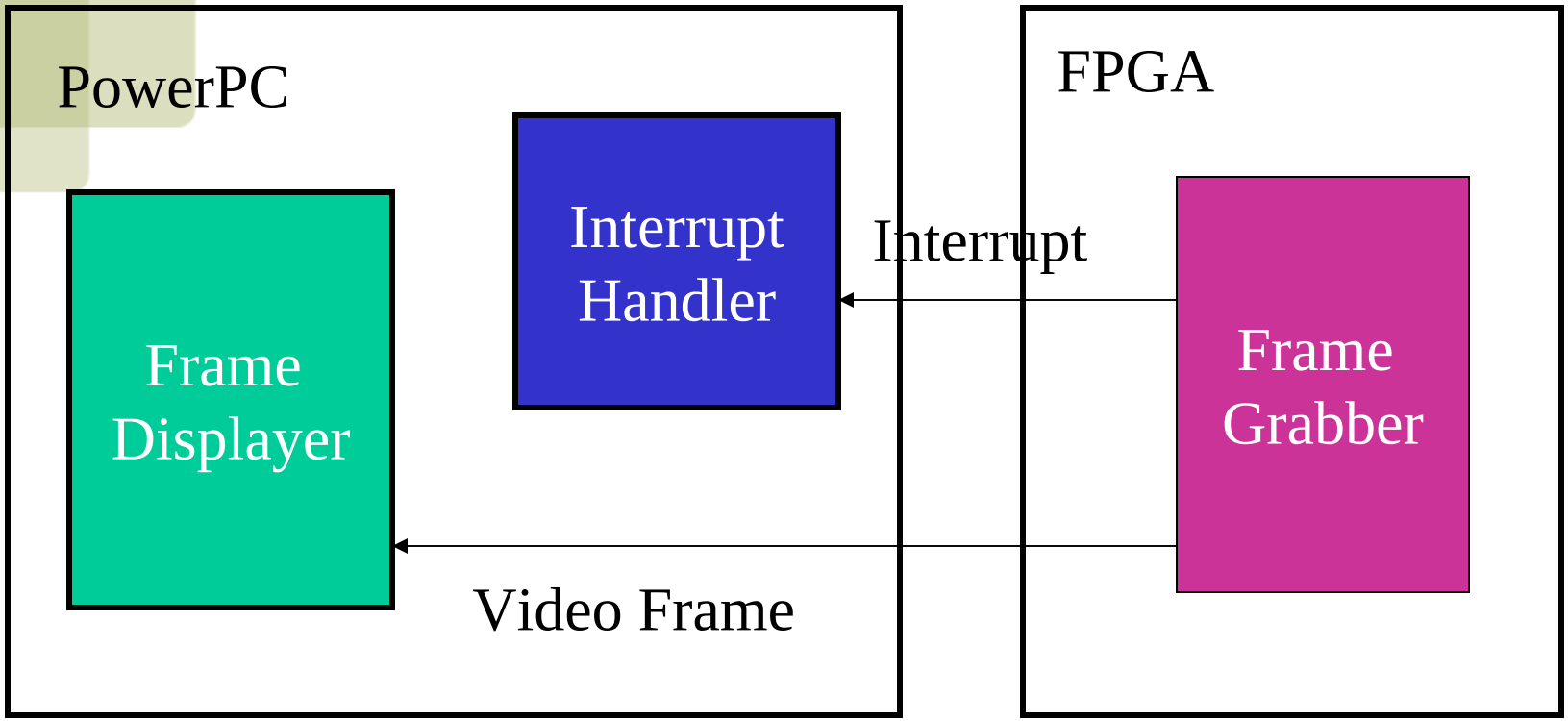
Interrupt Handlers

Most interrupt handlers follow a similar format:

1. Read/write data to/from the device that signaled the interrupt
   - Ex: get character typed on a keyboard
2. Signal any processes that are waiting for the device to complete its operation
3. Signal the device that the interrupt has been handled
   - Often called “clearing the interrupt”

Communicating With User Programs

- Problem: interrupts can't directly read/write data in a user program's space



Two Approaches

1. Interrupt handler copies data from interrupting device, places it into a buffer that is mapped onto a /dev/foo entry that user program can then read
   - Illustrated in book
   - Not the approach we recommend – buffer size/overrun issues
2. User program handles actual reading/writing of data to/from device.
   - User program sleeps after each data transfer
   - All the interrupt handler does is wake the user program up whenever the device is ready
   - In this approach, the user program, not the interrupt handler, should clear the interrupt bit on the device, to prevent the device from overwriting data before the user program has read it.

Example: Frame Grabber/Displayer

- Frame Display code on IPAQ:

```
While(1) {
    interruptible_sleep_on(&queue); /* sleep until interrupted */

    (Do PCMCIA reads to grab frame of data from XUP)
    (Do PCMCIA write that tells XUP that interrupt has been processed)
    (Do frame data munging and display)
}
```

Interrupt handler

```
void int_handler(int irq, void *dev_id, struct pt_regs *regs) {
    wake_up_interruptible(&queue);
    /* Just wake up anything waiting for the device */
}
```

Arguments to int_handler:

- irq: the number of the interrupt that caused the handler to be run
  - One handler could be associated with multiple interrupts
  - Somewhat provided for backwards-compatibility (UNIX, etc.)
- dev_id: Pointer to device data structure
  - Current best way for one interrupt handler to handle multiple devices
- regs: Snapshot of the processor's register state before the processor jumped to the interrupt handler
  - Rarely used, mostly for debugging

Registering an Interrupt Handler

- Once you've written your handler, you have to let the OS know that the handler should be associated with a specific interrupt

```
int request_irq(unsigned int irq,
   void (*handler)(int, void *, struct pt_regs *),
   unsigned long flags,
   const char *dev_name,
   void *dev_id);
```

Declared in <linux/sched.h>

Inputs to request_irq

- flags: bit vector that gives details about the interrupt handler's behavior
  - You'll want to use SA_INTERRUPT, which indicates that the interrupt handler is "fast," and can't be interrupted by another interrupt

- irq: number of the interrupt that the handler should be associated with
  - Needs to match the interrupt that the device will signal (duh!)
  - Bad things happen if two devices/handlers use the same irq and don't expect it
    - Can share interrupts if the devices and handlers know about it

Security and the Operating System

- Need to control access to hardware devices
- Approach:
  - Instructions that directly manipulate hardware resources are privileged, can only be run by the kernel
  - User programs make kernel calls to request that the OS access hardware, allows OS to set policies
  - Note: being logged in as root is not the same thing as running in kernel mode
    - Being root just means that the OS will almost never say “no” to your request