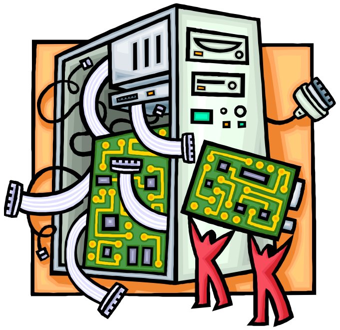
I/O = Input/Output Devices



MP7

```
int main()
{
  int *ptr = malloc(sizeof(int));

  *ptr = 4;
  free(ptr);

  printf("Memory was allocated, used, and freed!\n");
  return 0;
}
```

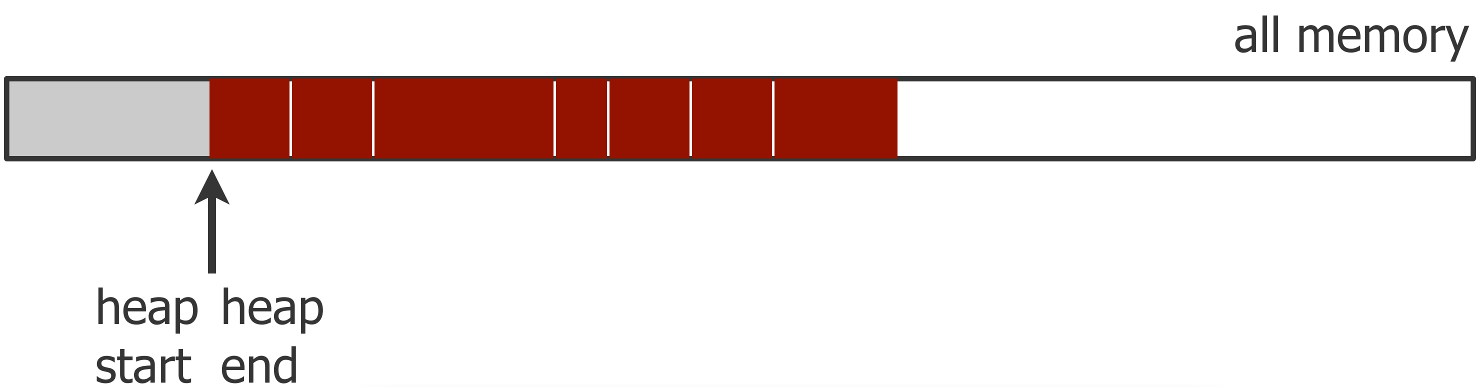
MP7

```
int main()
{
   int *ptr = malloc(sizeof(int));

   *ptr = 4;
   free(ptr);

   printf("Memory was allocated, used, and freed!\n");
   return 0;
}
```

Super-simple malloc

```
void* malloc(size_t size) {
  return sbrk(size);
}

void free(void* ptr) {
}
```

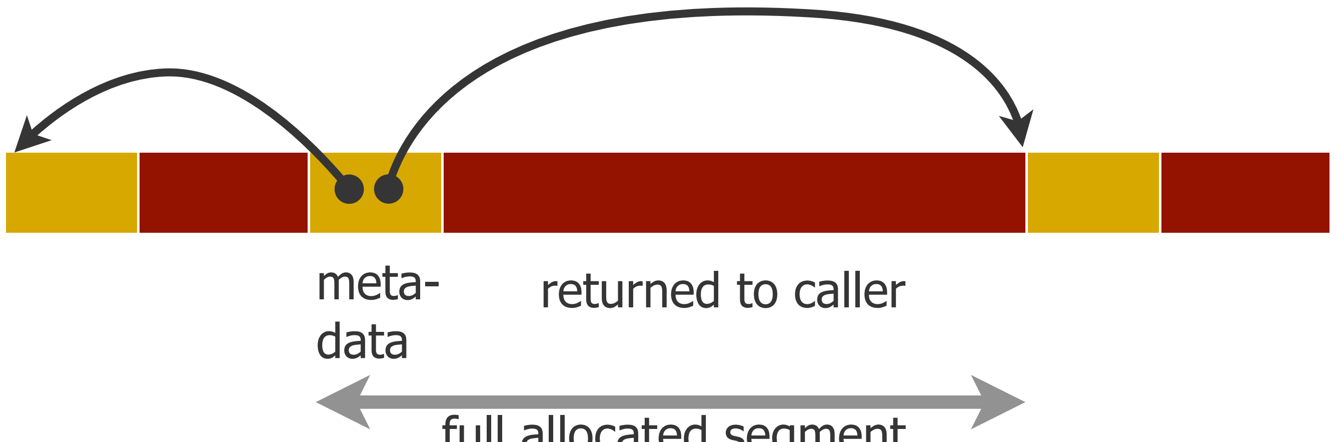
What does memory allocation look like with Super-Simple malloc?



The holes are wasted memory, allocated to the program but unused!

Keeping track of allocated and free memory segments

- Need some data structure (list, array, tree, ...) to track all segments
- One clever way:



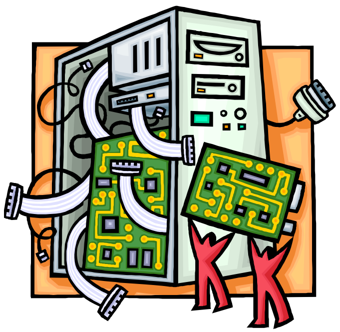
- ...but it's all up to you!

...and away you go

- Try it out with programs yourself
- We'll test based on
  - Average heap size
  - Max heap size
  - Execution time
- Fabulous prizes await you!
  - Contest details announced on Monday
  - Everyone who submits the MP participates, but anonymous by default

I/O = Input/Output Devices

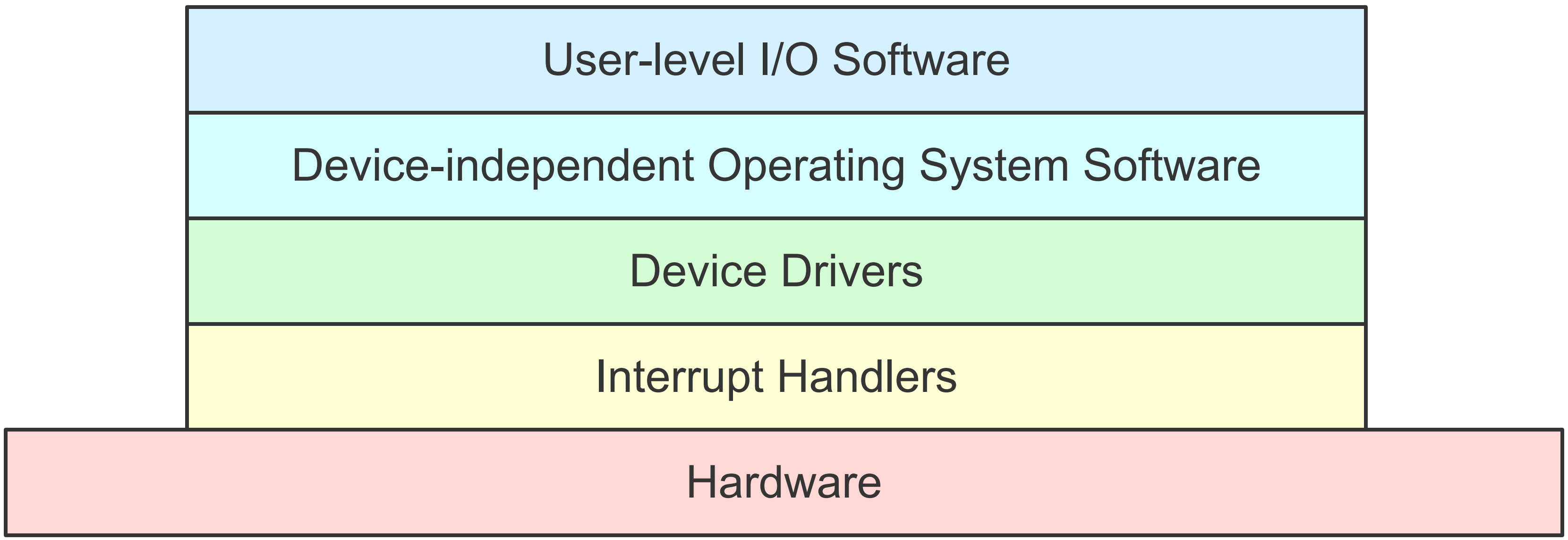
(for real this time!)



Overview

- Basic I/O hardware
  - ports, buses, devices and controllers
- I/O Software
  - Interrupt Handlers, Device Driver, Device-Independent Software, User-Space I/O Software
- Important concepts
  - Three ways to perform I/O operations
    - Programmed I/O, Interrupt and DMAs

I/O Software Layers



Layers of the I/O Software System

Devices

- Storage devices
  - Disk, tapes
- Transmission/Communication devices
  - Network card, modem
- Human interface devices
  - Screen, keyboard, mouse
- Specialized devices
  - Joystick

Input/Output Problems

- Wide variety of peripherals (external devices)
  - Delivering different amounts of data
  - At different speeds
  - In different formats
- All slower than CPU and RAM
  - Need I/O modules

I/O Device Characteristics

- Application usage
  - Disk for storing files or virtual memory pages
- Complexity of control
  - Simple vs. complex
- Data representation
  - Diversity of encoding schemes
- Error conditions
  - Devices respond to errors differently

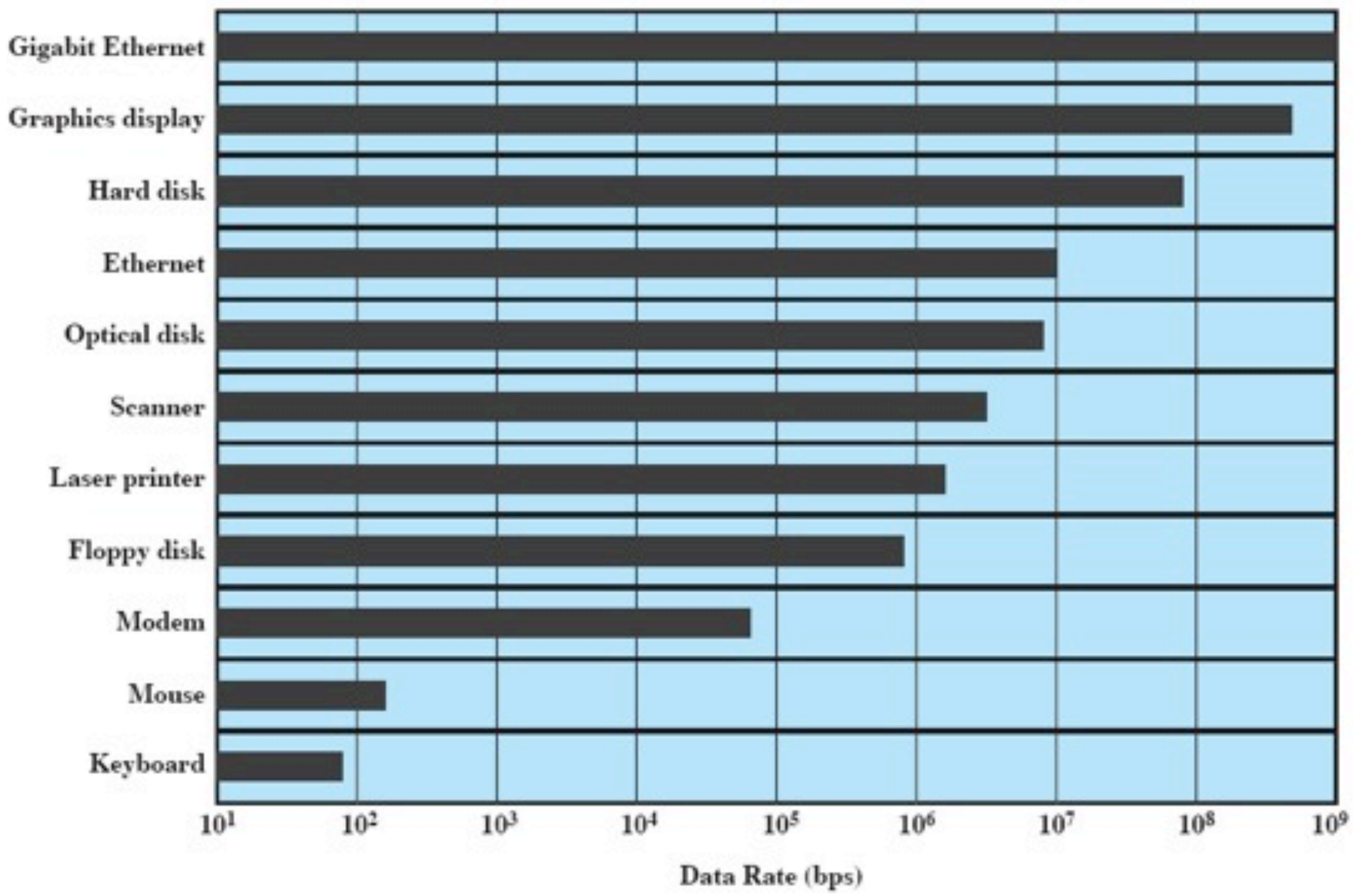
I/O Device Characteristics

- Unit of transfer
  - Data may be transferred as a stream of bytes for a terminal or in larger blocks for a disk
  - Block devices
    - Disk drives
    - Commands include read, write, seek
    - Raw I/O or file-system access
    - Memory-mapped file access possible
  - Character devices
    - Keyboards, mice, serial ports
    - Commands include get, put
    - Libraries layered on top allow line editing

I/O Device Characteristics

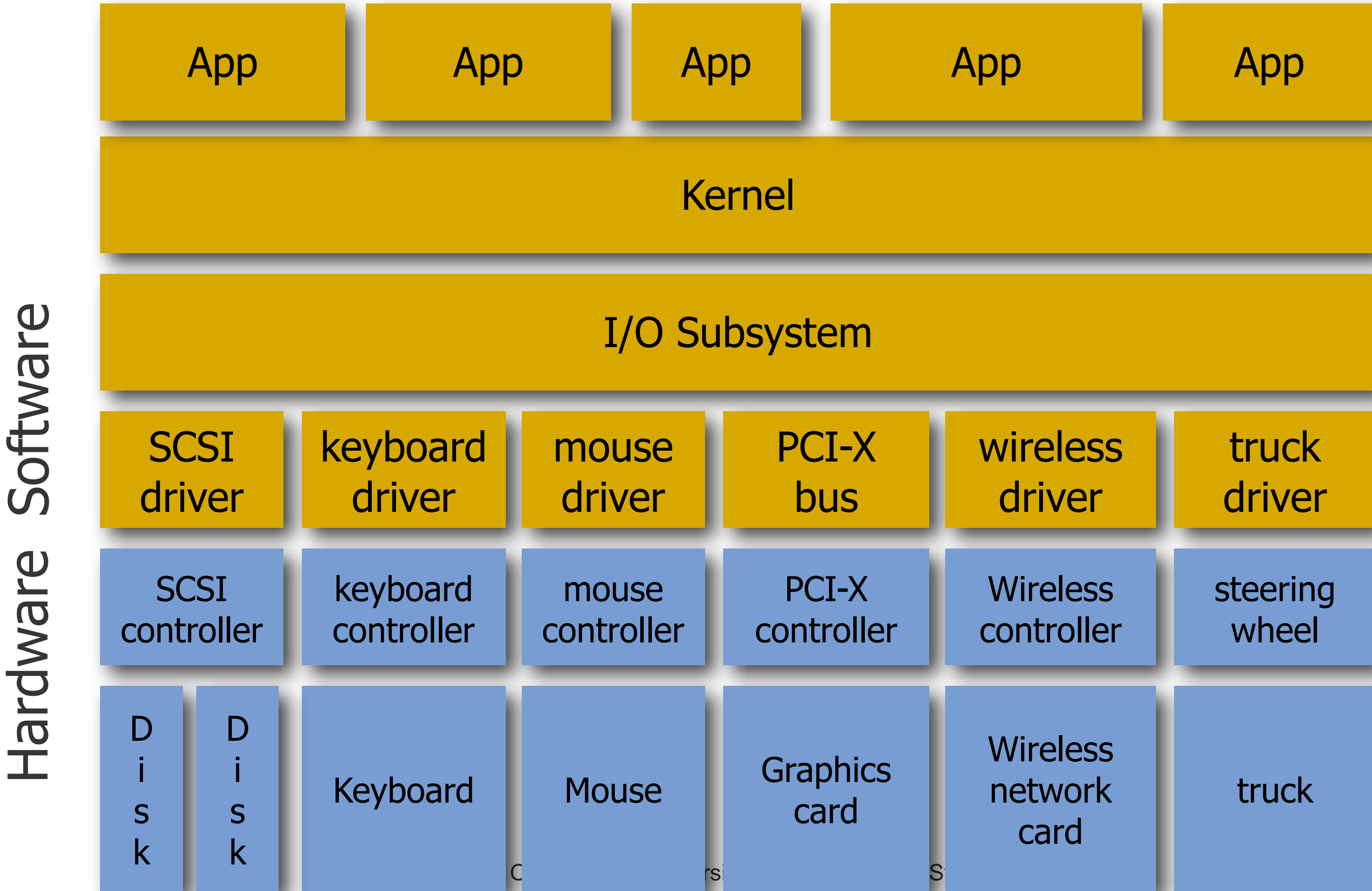
- Data rate
  - May be differences of several orders of magnitude between the data transfer rates

Typical I/O Device Data Rates



[Fig. from Silberschatz & Galvin]

Need some abstraction to deal with all this complexity!



Need some abstraction to deal with all this complexity!

- Kernel provides several abstractions that can represent many kinds of devices, such as:
  - block I/O (files)
  - character stream I/O (keyboard)
  - memory-mapped files
  - network sockets
  - `ioctl` for everything else

But first... Hardware-software interface

- Device controller
  - A hardware element
  - Accepts simple device hardware instructions into registers to read and write data
- Device driver
  - Part of the OS that runs in software on the CPU
  - Makes calls to the device controller

Device controller

- I/O units typically consist of
  - Mechanical component
    - The device itself
  - Electronic component
    - The device controller or adapter
- Interface between controller and device is a very low level interface
- Example: Disk controller
  - Take serial bit streams coming off the drive
  - Convert into a block of bytes
  - Perform error correction
  - Caching

Device controller

- Controller has I/O registers/ports for data and control
- CPU and controllers communicate via
  - I/O instructions and registers
  - Interrupts
  - Memory-mapped I/O

I/O Registers/Ports

- 4 registers, 1 to 4 bytes
  - Status
    - Whether the current command is completed, byte is available, device has an error, etc.
  - Control
    - Host determines to start a command or change the mode of a device
  - Data-in
    - Host reads to get input
  - Data-out
    - Host writes to send output

I/O Registers/Ports

- Instructions and Data
  - Format is device-dependent
  - Device driver code needs to be aware of this format
    - Each device from each vendor typically needs a separate device driver
- How should the CPU communicate with the control registers and the data buffer?

How to converse with devices

- Polling
  - CPU issues I/O command
  - CPU directly writes instructions into device's registers
  - CPU busy waits for completion
- Interrupt-driven I/O
  - CPU issues I/O command
  - CPU directly writes instructions into device's registers
  - CPU continues operation until interrupt
- Direct Memory Access (DMA)
  - CPU asks DMA controller to perform device-to-memory transfer
  - DMA issues I/O command and transfers new item into memory
  - CPU module is interrupted after completion

Polling

- Polling sequence:
  1. CPU requests I/O operation
  2. I/O module performs operation
  3. I/O module sets status bits
  4. CPU checks status bits periodically
- I/O module does not inform CPU directly
- I/O module does not interrupt CPU
- CPU may wait or come back later

- Also called “Programmed I/O”: each piece of I/O data is transferred by a program (kernel), not hardware

Polling

- Driver operation to input sequence of chars

```
i = 0;
while (…) {
   write_reg(opcode, read);
   while (busy_flag == true); /* wait */
   buffer[i] = data_buffer;
   i++;
   compute;
}
```

Polling

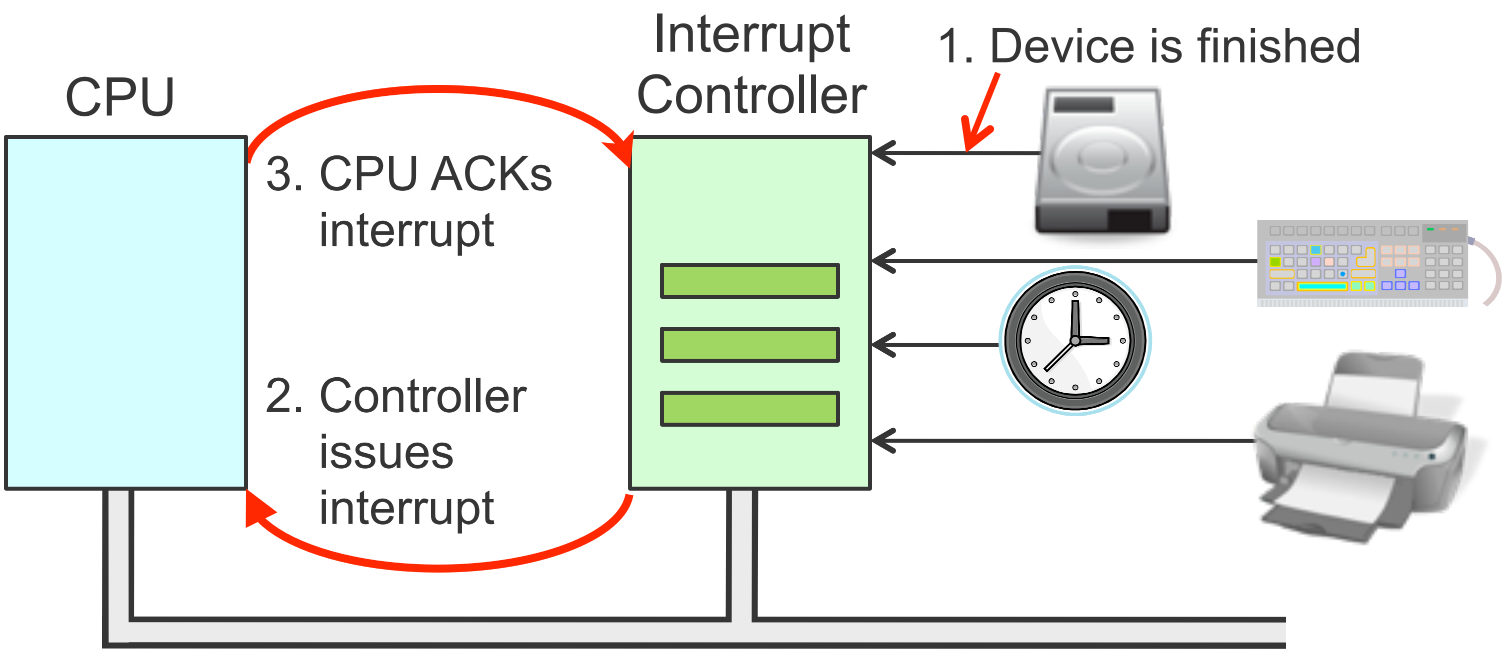
- Expensive for large transfers
  - What devices make large transfers?
  - What devices make small transfers?
- Acceptable only if
  - Small dedicated system
  - Not too few processes
  - Character devices (as opposed to block devices)

Interrupt-driven I/O

- Approach
  1. CPU issues read command
  2. I/O controller gets data while CPU does other work
  3. I/O controller interrupts CPU
  4. CPU requests data
  5. I/O controller transfers data
- Advantage: Overcomes CPU busy waiting loops
  - No repeated CPU checking of device
  - I/O module interrupts when ready: Event-driven!
- But, like polling, it's still "Programmed I/O"

Interrupt-driven I/O

Key Idea: Inform device controller of IO request, go to blocked state, wait for device to finish request



- Connections between devices and interrupt controller use shared interrupt lines on the bus rather than dedicated wires

Interrupt-driven I/O

- Driver operation to input sequence of chars

```
i = 0;
while (…) {
   write_reg(opcode, read);
   block to wait for interrupt;
   buffer[i] = data_buffer;
   i++;
   compute;
}
```

was:

```
while (busy_flag == true);
```

Host-controller interface: Interrupts

- CPU hardware has an interrupt report line that the CPU tests after executing every instruction
  - If a(ny) device raises an interrupt by setting interrupt report line
    - CPU catches the interrupt and saves the state of current running process into PCB
    - CPU dispatches/starts the interrupt handler
    - Interrupt handler determines cause, services the device and clears the interrupt report line
- Real life analogy for interrupts
  - An alarm sets off when the food/laundry is ready
  - So you can do other things in between

Support for Interrupts

- Need the ability to defer interrupt handling during critical processing
  - Why?
- Need efficient way to dispatch the proper interrupt handler
  - Interrupt comes with an id
  - Interrupt vector maintains addresses of interrupt handler functions (one per device) – an array of function pointers
  - Id is index into vector of device driver functions
- Need multilevel interrupts - interrupt priority level
  - Some interrupts more important than others, e.g., clock more than network

Interrupt Handler

- Discovery
  - At boot time, OS probes the hardware buses to
    - Determine what devices are present
    - Install corresponding interrupt handlers into the interrupt vector
- During I/O interrupt
  - Device controller implicitly signals that device is ready for next request

Other Uses of Interrupts

- Besides I/O devices
- Interrupt mechanisms are used to handle a wide variety of exceptions:
  - Division by zero, wrong address
  - System calls (software interrupts/signals, trap)
  - Multi-threaded systems
    - Examples?
  - Virtual memory paging

How to converse with devices

- Polling
  - CPU issues I/O command
  - CPU directly writes instructions into device's registers
  - CPU busy waits for completion
- Interrupt-driven I/O
  - CPU issues I/O command
  - CPU directly writes instructions into device's registers
  - CPU continues operation until interrupt
- Next time: Direct Memory Access (DMA)
  - CPU asks DMA controller to perform device-to-memory transfer
  - DMA issues I/O command and transfers new item into memory
  - CPU module is interrupted after completion