I/O systems

CS 241

April 30, 2014

University of Illinois

Input and Output

A computer's job is to process data

- Computation (CPU, cache, and memory)
- Move data into and out of a system (between I/O devices and memory)

Challenges with I/O devices

- Different categories: storage, networking, displays, etc.
- Large number of device drivers to support
- Device drivers run in kernel mode and can crash systems

Goals of the OS

- Provide a generic, consistent, convenient and reliable way to access I/O devices
- As device-independent as possible
- High performance I/O

How does the CPU talk to devices?

Device controller: Hardware that enables devices to talk to the peripheral bus

Host adapter: Hardware that enables the computer to talk to the peripheral bus

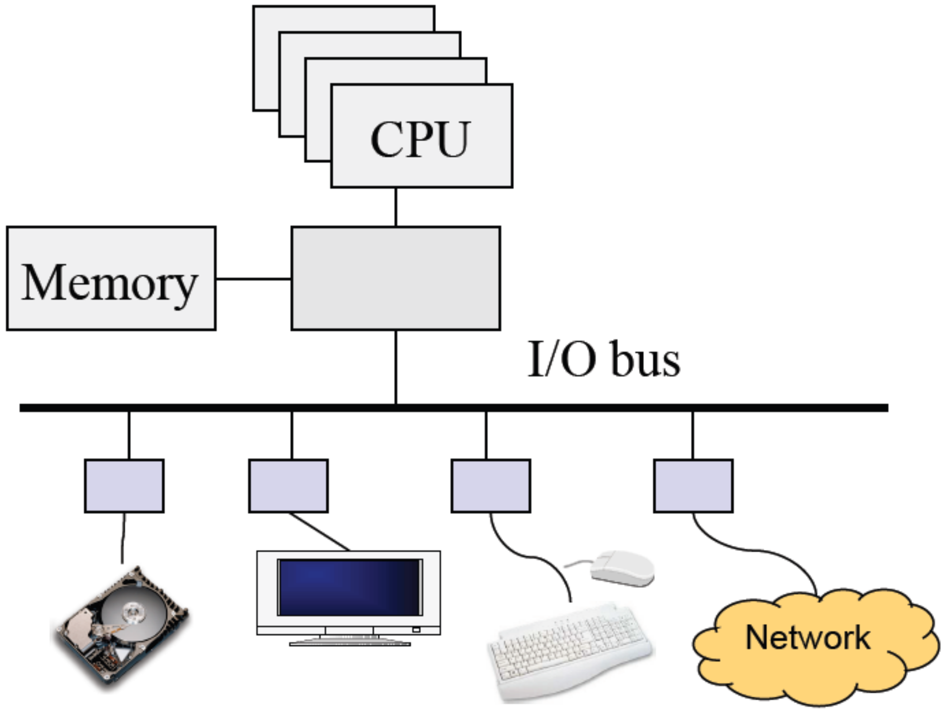
Bus: Wires that transfer data between components inside computer

Device controller allows OS to specify simpler instructions to access data

Example: a disk controller

- Translates "access sector 23" to "move head reader 1.672725272 cm from edge of platter"
- Disk controller "advertises" disk parameters to OS, hides internal disk geometry
- Most modern hard drives have disk controller embedded as a chip on the physical device

Review: Computer Architecture



Compute hardware

- CPU and caches
- Chipset
- Memory

I/O Hardware

- I/O bus or interconnect
- I/O controller or adaptor
- I/O device

Two types of I/O

- Programmed I/O (PIO)
  - CPU does the work of moving data
- Direct Memory Access (DMA)
  - CPU offloads the work of moving data to DMA controller

Programmed Input Device

Device controller

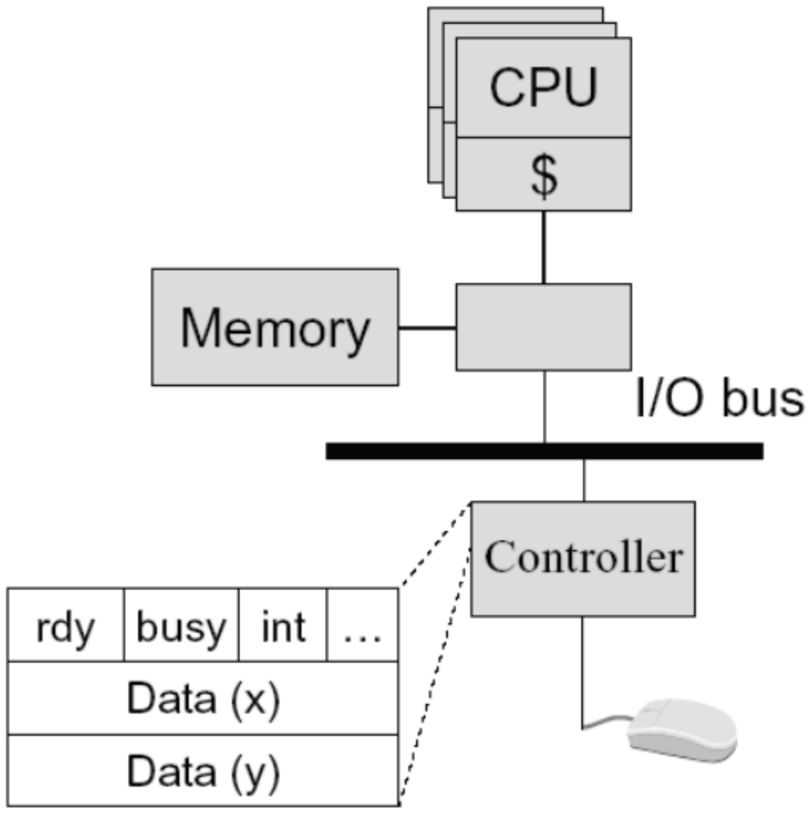
- Status registers
  - ready: tells if the host is done
  - busy: tells if the controller is done
  - int: interrupt
  - …
- Data registers

A simple mouse design

- When moved, put (X, Y) in mouse's device controller's data registers
- Interrupt CPU

Input on an interrupt

- CPU saves state of currently-executing program
- Reads values in X, Y registers
- Sets ready bit
- Wakes up a process/thread or execute a piece of code to handle interrupt



Programmed Output Device

Device

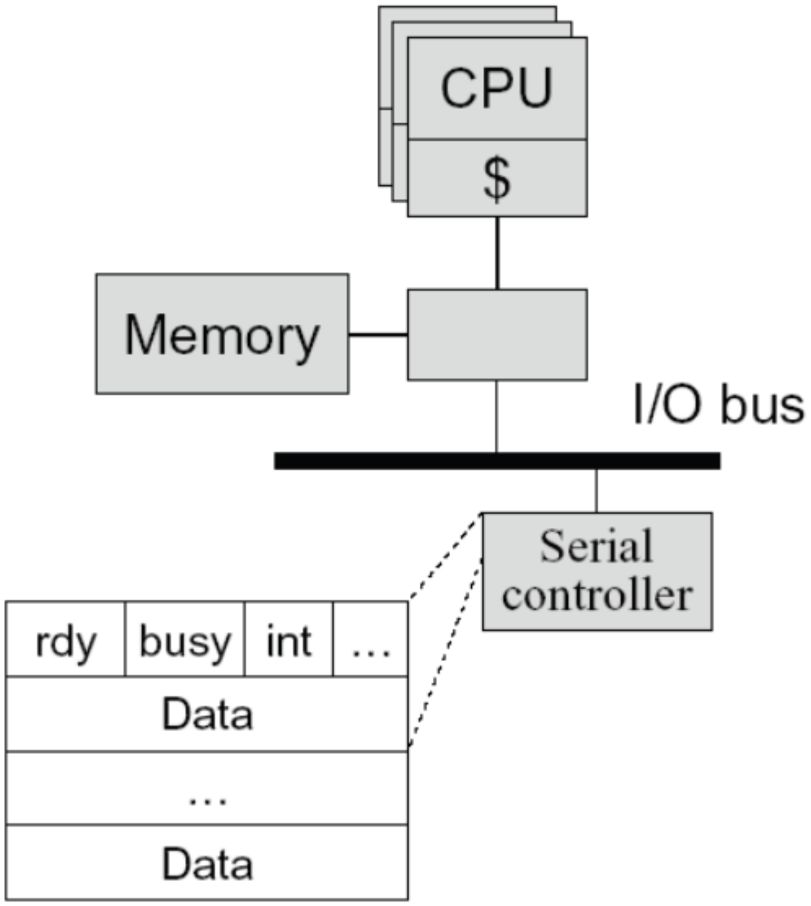
- Status registers (ready, busy, …)
- Data registers

Example

- A serial output device

Perform an output

- CPU: Poll the busy bit
- Writes the data to data register(s)
- Set ready bit
- Controller sets busy bit and transfers data
- Controller clears the busy bit



Direct Memory Access (DMA)

DMA controller or adaptor

- Status register (ready, busy, interrupt, …)
- DMA command register
- DMA register (address, size)
- DMA buffer

Host CPU initiates DMA

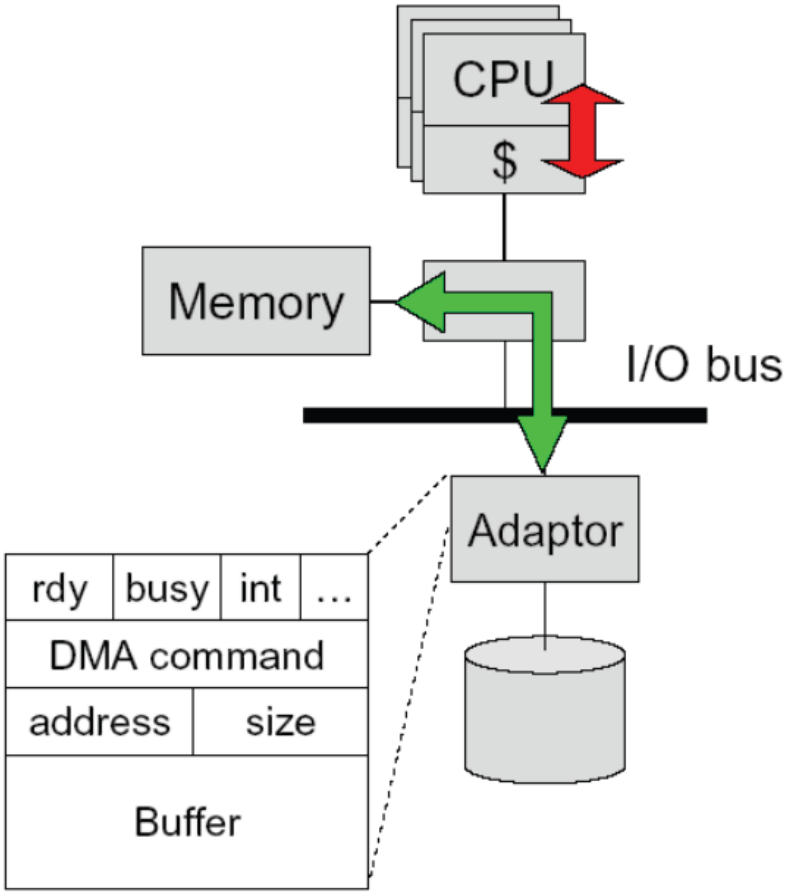
- Device driver call (kernel mode)
- Wait until DMA device is free
- Initiate a DMA transaction (command, memory address, size)
- Block

Controller performs DMA

- DMA data to device (size--; address++)
- Issue interrupt on completion (size == 0)

CPU's interrupt handler

- Wakeup the blocked process



Memory-mapped I/O

Use the same address bus to address both memory and I/O devices

- The memory and registers of I/O devices are mapped to address values
- Allows same CPU instructions to be used with regular memory and devices

I/O devices, memory controller, monitor address bus

- Each responds to addresses they own

Orthogonal to DMA

- May be used with, or without, DMA

Polling- vs. Interrupt-driven I/O

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)

- Typically done with Interrupt-driven I/O
- 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

Which is better, polling or interrupt-driven I/O?

Polling- vs. Interrupt-driven I/O

Polling

- Expensive for large transfers
- Better for small, dedicated systems with infrequent I/O

Interrupt-driven

- Overcomes CPU busy waiting
- I/O module interrupts when ready: event driven

How Interrupts are implemented

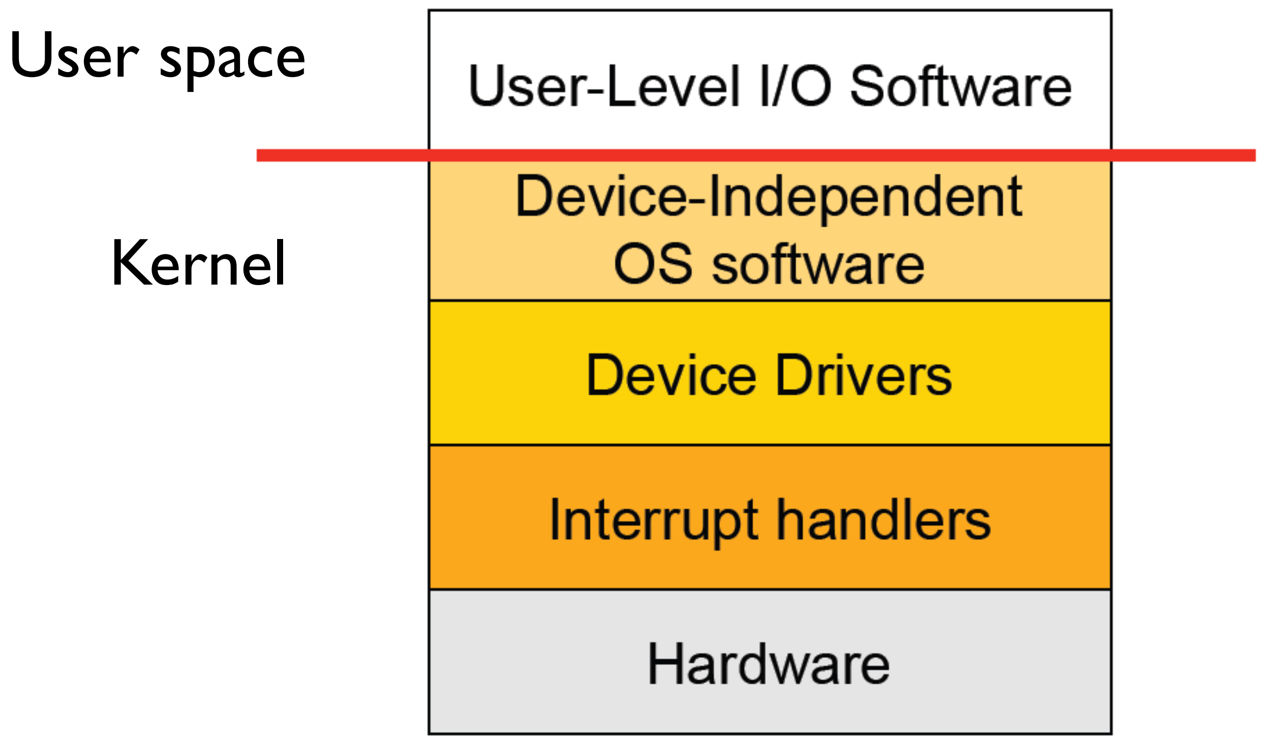
CPU hardware has an interrupt report line that the CPU tests after executing every instruction

- 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

Other uses of interrupts: exceptions

- Division by zero, wrong address
- System calls (software interrupts/signals, trap)
- Virtual memory paging

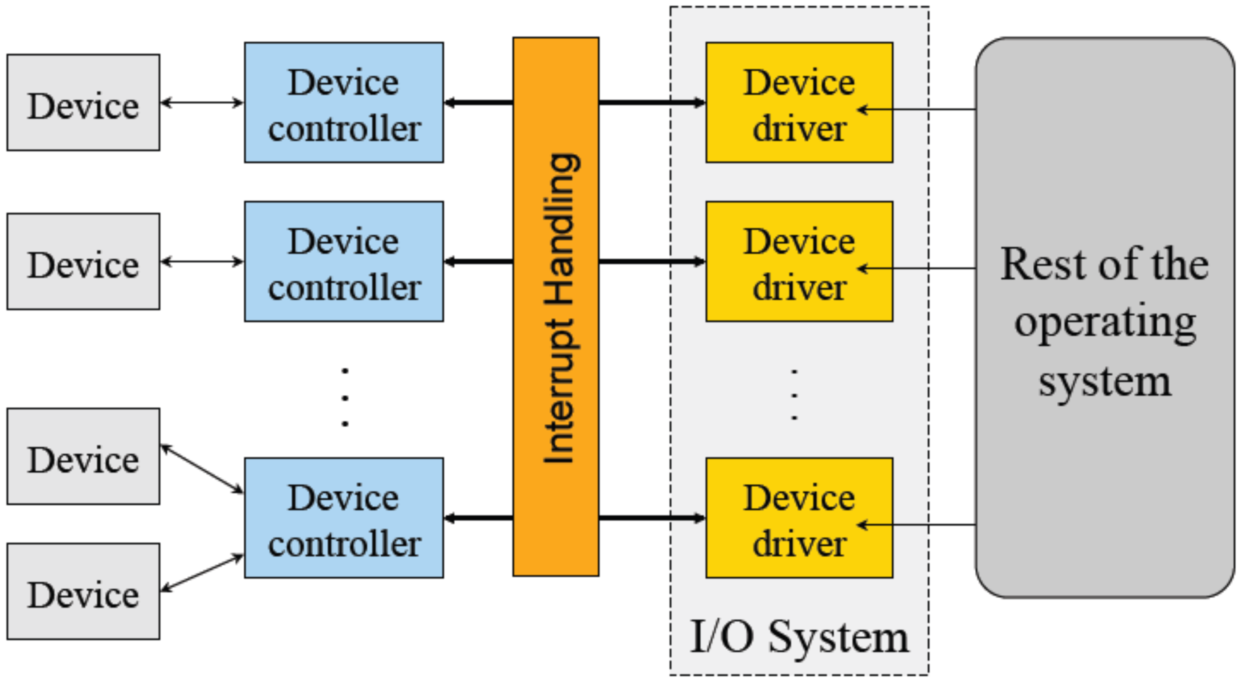
I/O Software Stack



Device drivers

Manage the complexity and differences among specific types of devices (disk vs. mouse, different types of disks …)

Special driver for each device type or group of them (eg, USB)



Typical Device Driver Design

Operating system and driver communication

- Commands and data between OS and device drivers

Driver and hardware communication

- Commands and data between driver and hardware

Driver responsibilities

- Initialize devices
- Interpreting commands from OS
- Schedule multiple outstanding requests
- Accept and process interrupts
- Manage data transfers

Device Driver Behavior

1. Check input parameters for validity, and translate them to device specific language
2. Check if device is free (wait or block if not)
3. Issue commands to control device
   - Write them into device controller's registers
   - Check after each if device is ready for next (wait or block if not)
4. Block or wait for controller to finish work
5. Check for errors, and pass data to device-independent software
6. Return status information
7. Process next queued request, or block waiting for next

Challenges:

- Must be reentrant (can be called by an interrupt while running)
- Handle hot-pluggable devices and device removal while running
- Complex and many of them; bugs in them can crash system

Types of I/O Devices

Block devices

- Organize data in fixed-size blocks
- Transfers are in units of blocks
- Blocks have addresses and data are therefore addressable
- E.g. hard disks, USB disks, CD-ROMs

Character devices

- Delivers or accepts a stream of characters, no block structure
- Not addressable, no seeks
- Can read from stream or write to stream
- Network interfaces, keyboards

Like everything, not a perfect classification

- E.g. tape drives have blocks but not randomly accessed
- Clocks are I/O devices that just generate interrupts

Char/Block Device Interfaces

Character device interface

- `read( deviceNumber, bufferAddr, size )`
  - Reads “size” bytes from a byte stream device to “bufferAddr”
- `write( deviceNumber, bufferAddr, size )`
  - Write “size” bytes from “bufferAddr” to a byte stream device

Block device interface

- `read( deviceNumber, deviceAddr, bufferAddr )`
  - Transfer a block of data from “deviceAddr” to “bufferAddr”
- `write( deviceNumber, deviceAddr, bufferAddr )`
  - Transfer a block of data from “bufferAddr” to “deviceAddr”
- `seek( deviceNumber, deviceAddress )`
  - Move the head to the correct position
  - Usually optional

User-level interfaces: syscalls

Character device interface

- read(deviceNumber, bufferAddr, size)
  - Reads “size” bytes from a byte stream device to “bufferAddr”
- write(deviceNumber, bufferAddr, size)
  - Write “size” bytes from “bufferAddr” to a byte stream device

Block device interface

- read(deviceNumber, deviceAddr, bufferAddr)
  - Transfer a block of data from “deviceAddr” to “bufferAddr”
- write(deviceNumber, deviceAddr, bufferAddr)
  - Transfer a block of data from “bufferAddr” to “deviceAddr”
- seek(deviceNumber, deviceAddress)
  - Move the head to the correct position
  - Usually not necessary

Sync vs Asynchronous I/O

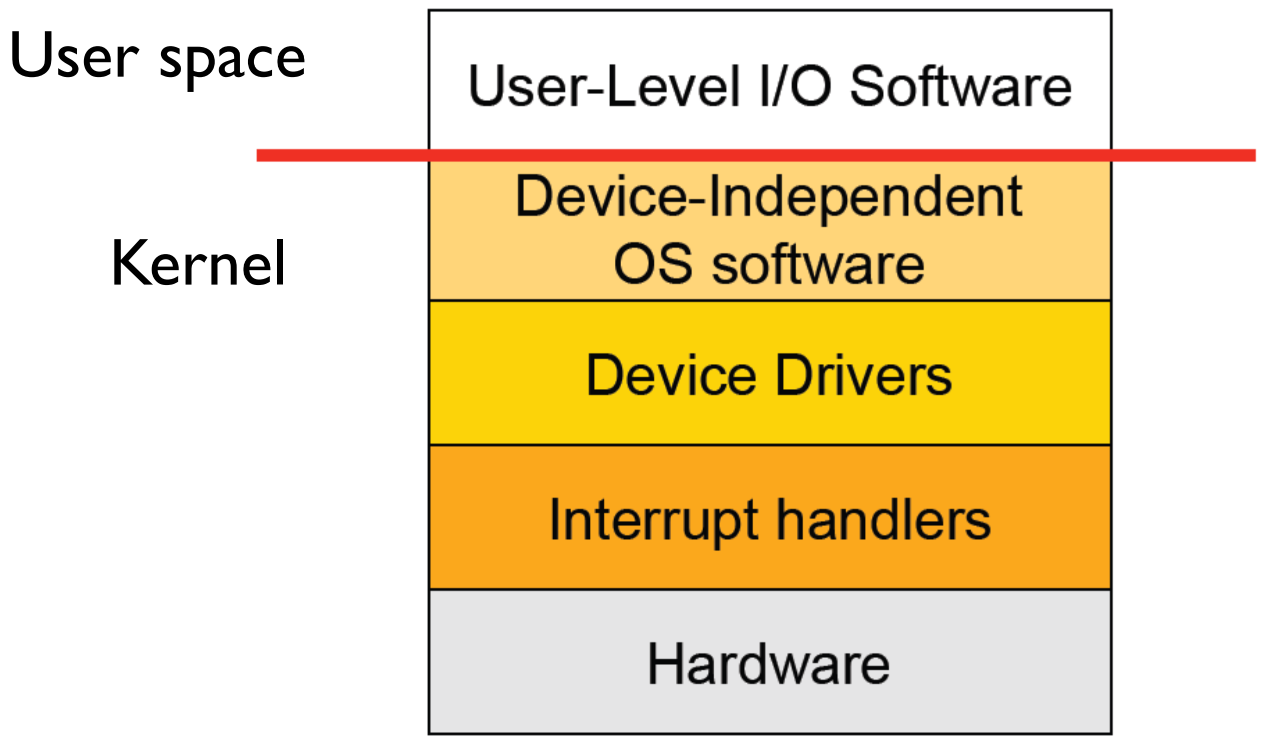
Synchronous I/O

- read() or write() will block a user process until its completion
- OS overlaps synchronous I/O with another process

Asynchronous I/O

- read() or write() will not block a user process
    - returns -1, sets error code EAGAIN or EWOULDBLOCK
- user process can do other things before I/O completion
- can determine if device is ready with select() / poll() / epoll()
- Make asynchronous with O_NONBLOCK option on open() or later via fcntl()

Finished our tour through the stack



Example: Blocked Read

A process issues a read call which executes a system call

System call code checks for correctness

If it needs to perform I/O, it will issue a device driver call

Device driver allocates a buffer for read and schedules I/O

Controller performs DMA data transfer

Block the current process and schedule a ready process

Device generates an interrupt on completion

Interrupt handler stores any data and notifies completion

Move data from kernel buffer to user buffer

Wakeup blocked process (make it ready)

User process continues when it is scheduled to run

Does I/O overhead matter?

Many steps involved in data I/O

How much can this overhead slow us down?

Experiment: `copy.c`