CSCI 3753 Operating Systems
Spring 2019

Christopher Godley

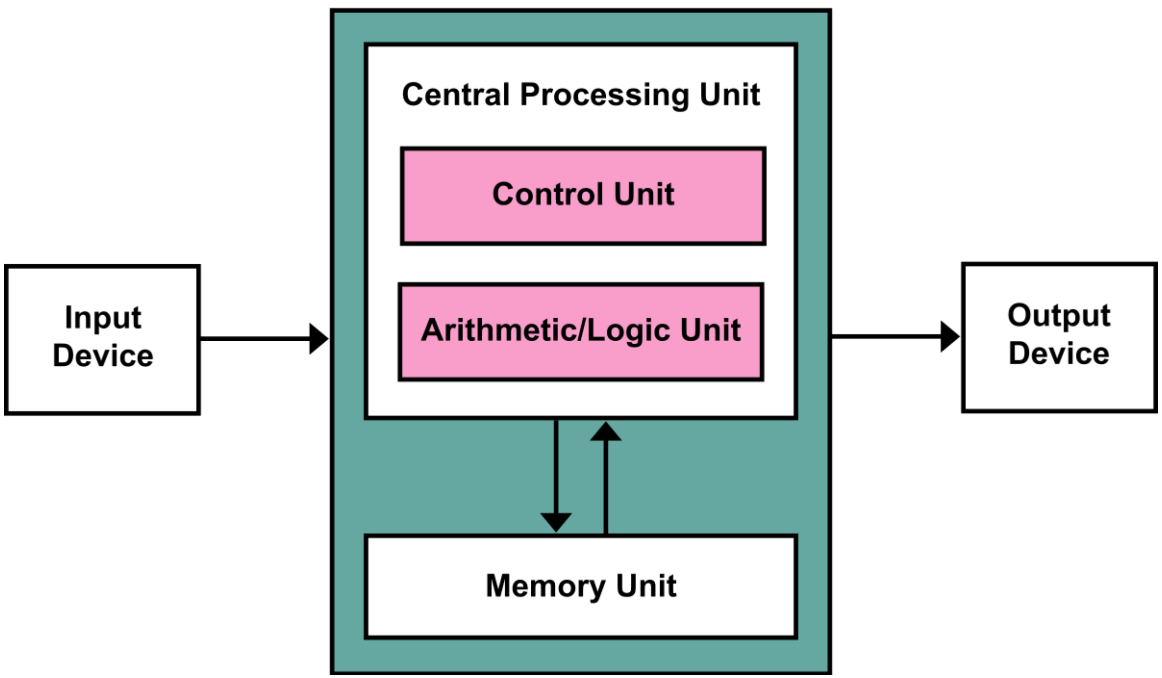
PhD Student

Department of Computer Science

University of Colorado Boulder

Lecture 3
Device Management

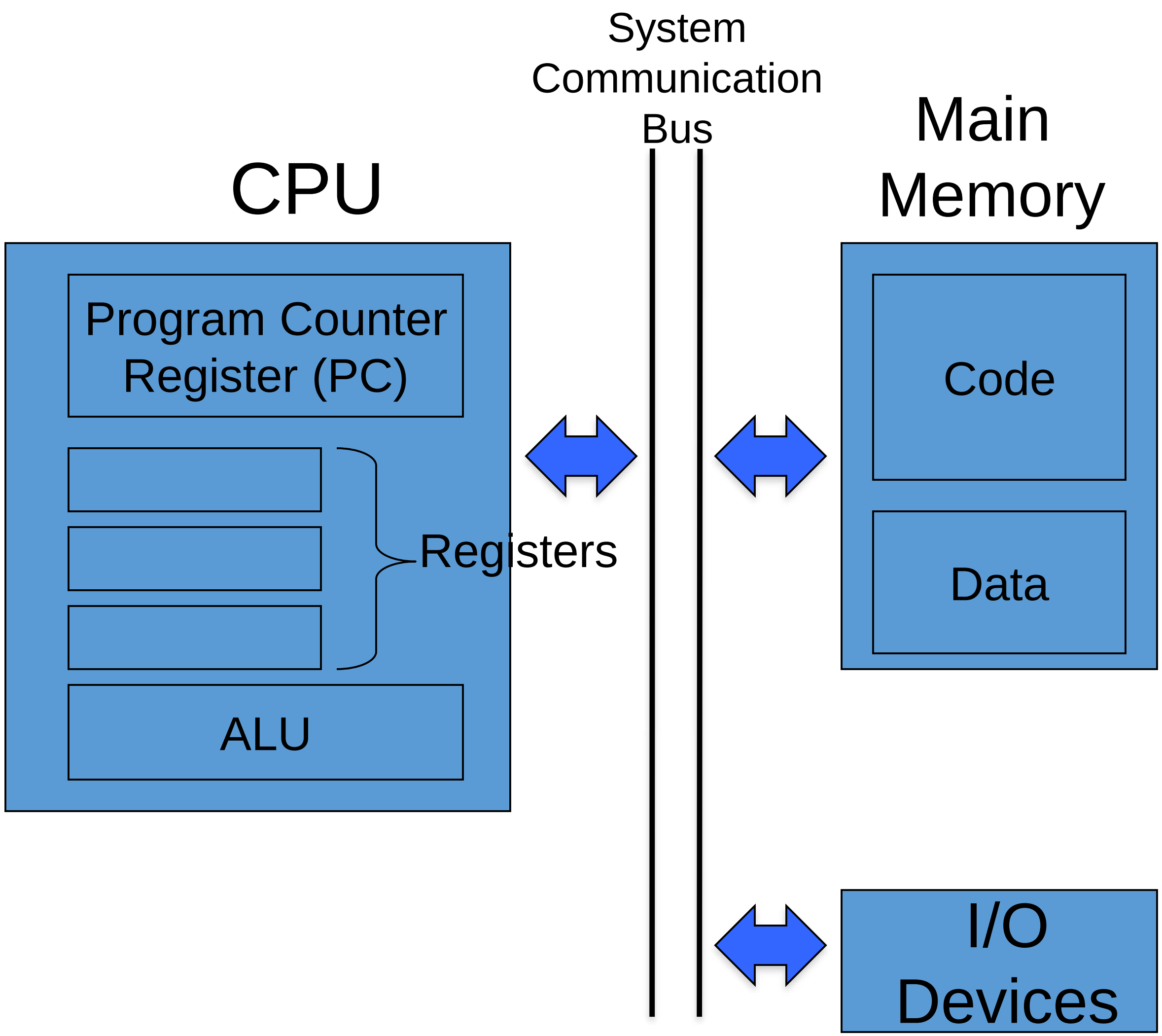
Von Neumann Computer Architecture



In 1945, von Neumann described a “stored-program” digital computer in which memory stored both instructions *and* data

This simplified loading of new programs and executing them without having to rewire the entire computer each time a new program needed to be loaded

Von Neumann Computer Architecture

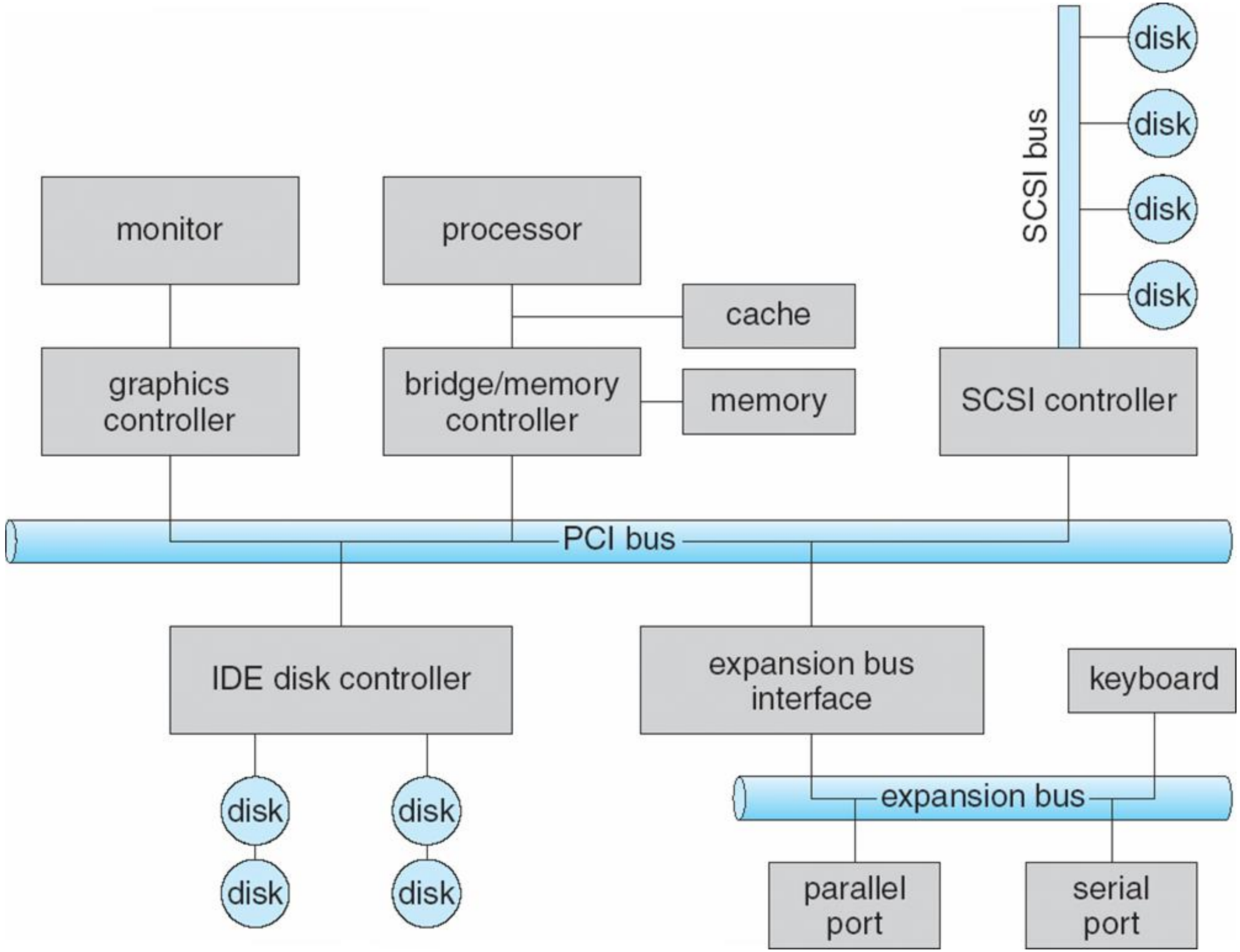


Want to support more devices: card reader, magnetic tape reader, printer, display, disk storage, etc.

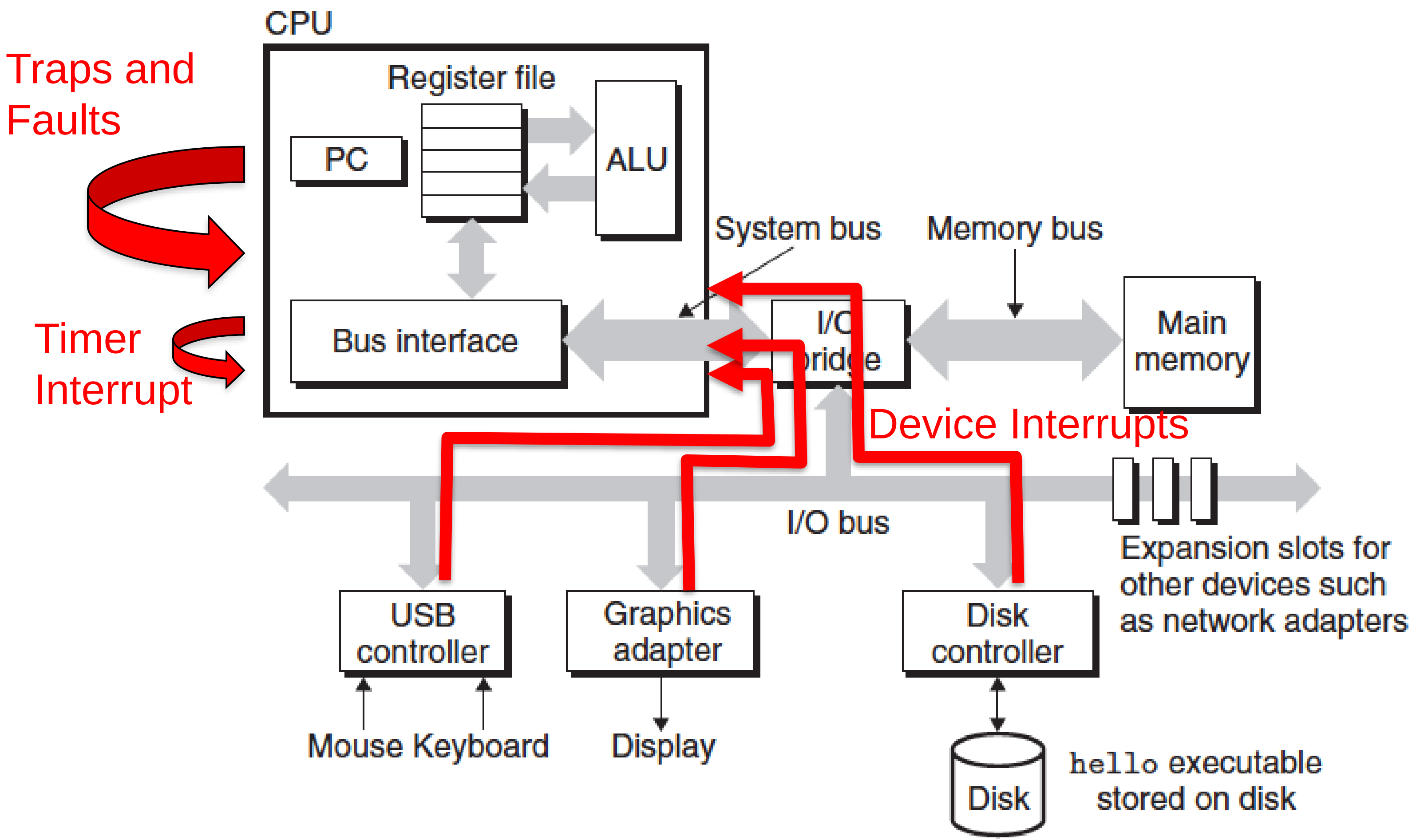
System bus evolved to handle multiple I/O devices.

Includes control, address and data buses

A Typical PC Bus Structure



Modern Computer Architecture: Devices and the I/O Bus



Classes of Exceptions

Class	Cause	Examples	Return behavior
Trap	Intentional exception, i.e. “software interrupt”	System calls	always returns to next instruction, synchronous
Fault	Potentially recoverable error	Divide by 0, stack overflow, invalid opcode, page fault, segmentation fault	might return to current instruction, synchronous
(Hardware) Interrupt	signal from I/O device	Disk read finished, packet arrived on network interface card (NIC)	always returns to next instruction, asynchronous
Abort	nonrecoverable error	Hardware bus failure	never returns, synchronous

Examples of x86 Exceptions

Exception Table

0-31 reserved for hardware

Exception Number	Description	Exception Class	Pointer to Handler
0	Divide error	fault	---
13	General protection fault	fault	---
14	Page fault	fault	---
18	machine check	abort	---
32-127	OS-defined	Interrupt or trap	---
128	System call	Trap	---
129-255	OS-defined	Interrupt or trap	---

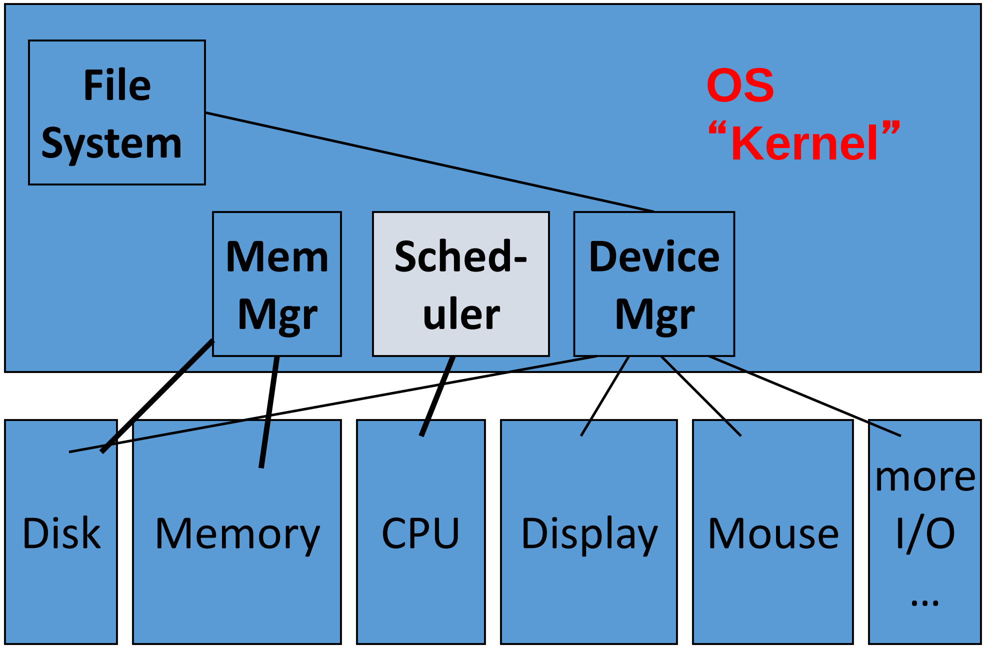
OS assigns

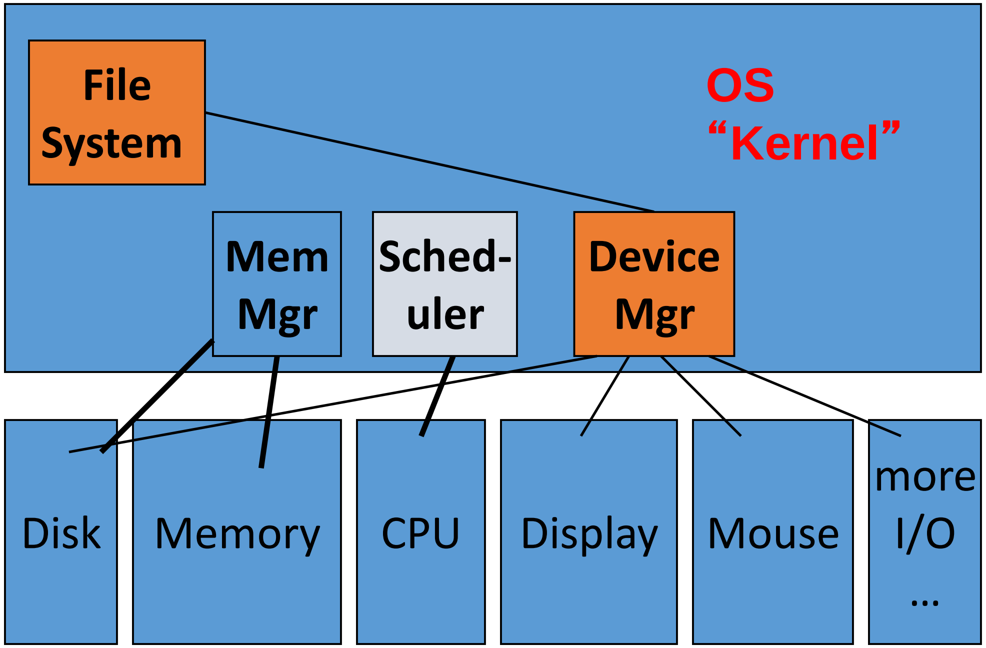
offsets form *interrupt vector*

Examples of x86 Exceptions

- x86 Pentium: Table of 256 different exception types
  - some assigned by CPU designers (divide by zero, memory access violations, page faults)
  - some assigned by OS, e.g. interrupts or traps

- Pentium CPU contains exception table base register that points to this table, so it can be located anywhere in memory


File
System
OS
“Kernel”
Mem
Mgr
Sched-
uler
Device
Mgr
Disk
Memory
CPU
Display
Mouse
more
I/O
…

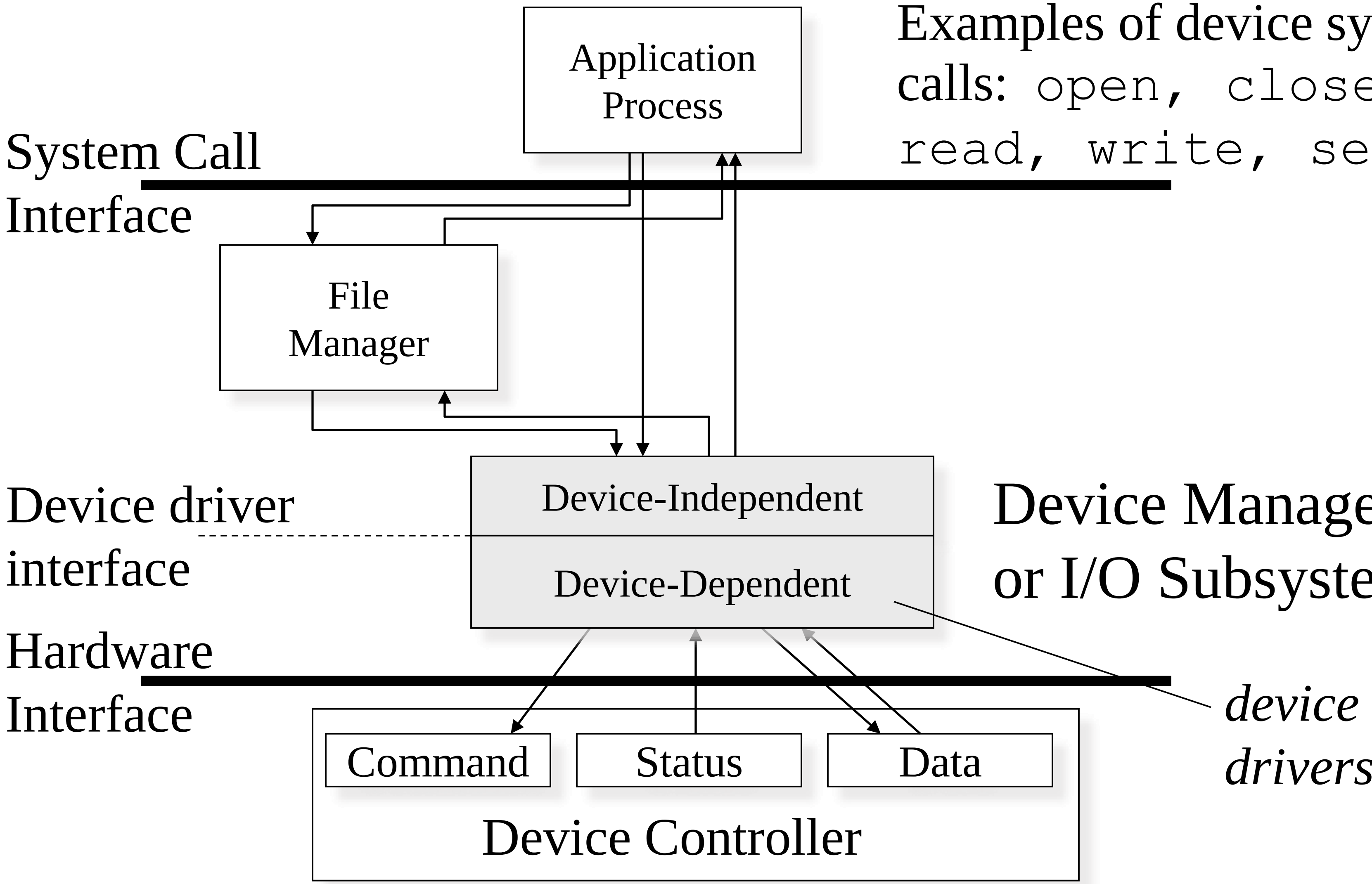

File System
OS “Kernel”
Mem Mgr
Sched- uler
Device Mgr
Disk
Memory
CPU
Display
Mouse
more I/O …

Device Manager

- Controls operation of I/O devices
  - Issue I/O commands to the devices
  - Catch interrupts
  - Handle errors
  - Provide a simple and easy-to-use interface
    - Device independence: same interface for all devices.

POSIX

Device Management Organization



Examples of device system calls: `open`, `close`, `read`, `write`, `set`,...

Device Manager or I/O Subsystem

Device System Call Interface

- Create a simple standard interface to access most devices
  - Every I/O device driver should support the following:
    open, close, read, write, set (ioctl in UNIX), stop, etc.

    POSIX
  - Block vs character
    - Specify how we talk to the device
  - Sequential vs direct/random access
    - Old tape vs. Disk
  - Blocking I/O versus Non-Blocking I/O
    - blocking system call: process put on wait queue until I/O completes
    - non-blocking system call: returns immediately with partial number of bytes transferred, e.g. keyboard, mouse, network
  - Synchronous versus asynchronous
    - asynchronous returns immediately, but at some later time, the full number of bytes requested is transferred

ioctl and fcntl (input/output control)

- Want a richer interface for managing I/O devices than just open, close, read, write, …
- ioctl allows a user-space application to configure parameters and/or actions of an I/O device
  - e.g set the speed of a device, or eject a disk
- Usage: *int ioctl(int fd, int cmd, ...)*;
  - Invokes a system call to execute **device-specific *cmd*** on I/O device *fd*
  - Used for I/O operations and other operations which cannot be expressed by regular system calls
  - Requests are directed to the correct device driver

ioctl and fcntl (input/output control)

- Avoids having to create new system calls for each new device and/or unforeseen device function
  - Helps make the OS/kernel extensible
- UNIX, Linux, MacOS X all support ioctl, and Windows has its own version
- In UNIX, each device is modeled as a file
  - *fcntl* for file control is related to ioctl and is used for configuring file parameters, hence in many cases I/O communication
  - e.g. use fcntl to set a network socket to non-blocking
  - part of POSIX API, so portable across platform

Device Characteristics

- I/O devices consist of two high-level components
  - Mechanical components
  - Electronic components:
    The device is operated by Device controllers
- OS deals with device controllers
  - Through device driver

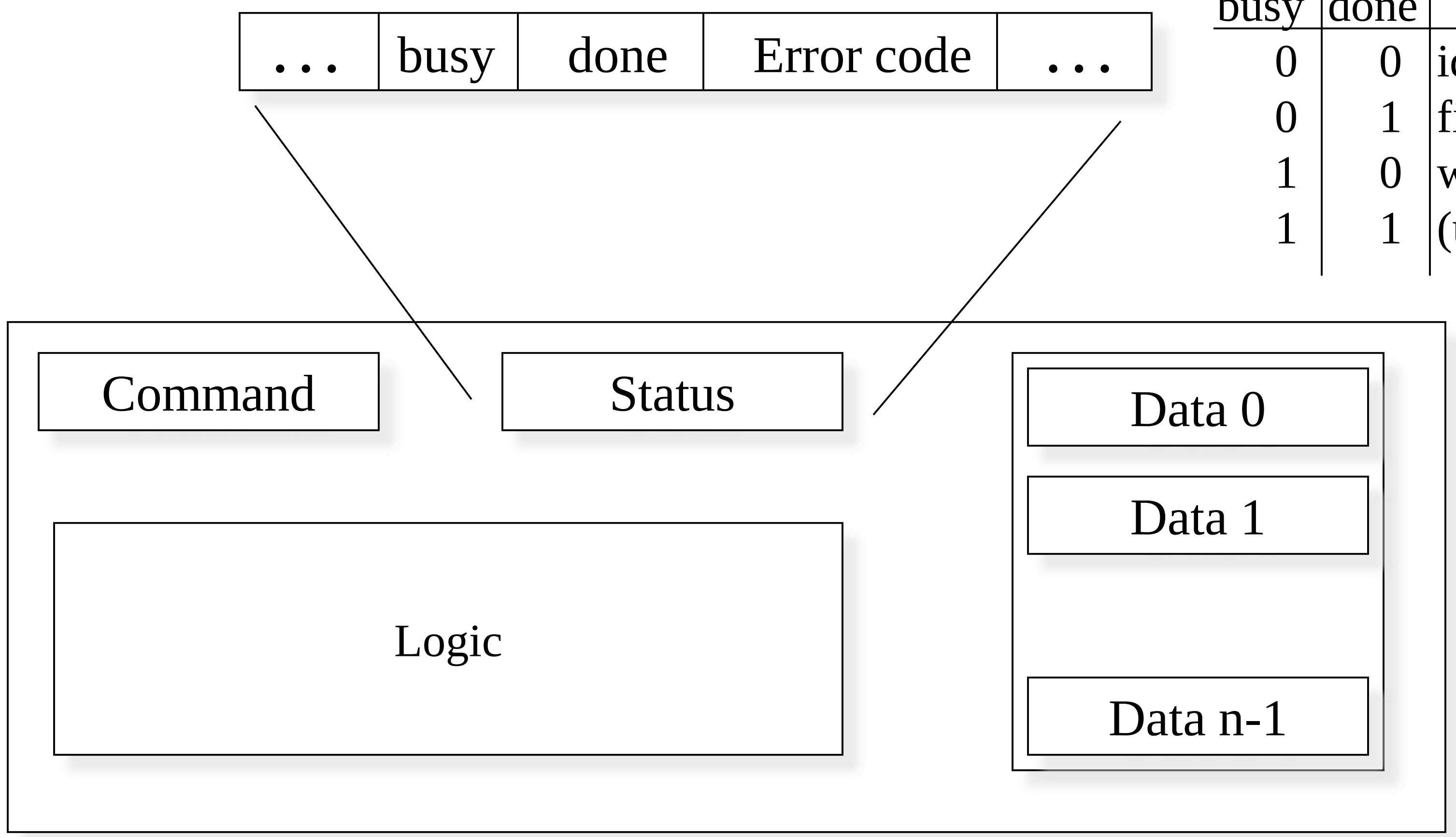
Device Drivers

- Support the device system call interface functions open, read, write, etc. for that device

- Interact directly with the device controllers
  - Know the details of what commands the device can handle, how to set/get bits in device controller registers, etc.
  - Are part of the device-dependent component of the device manager

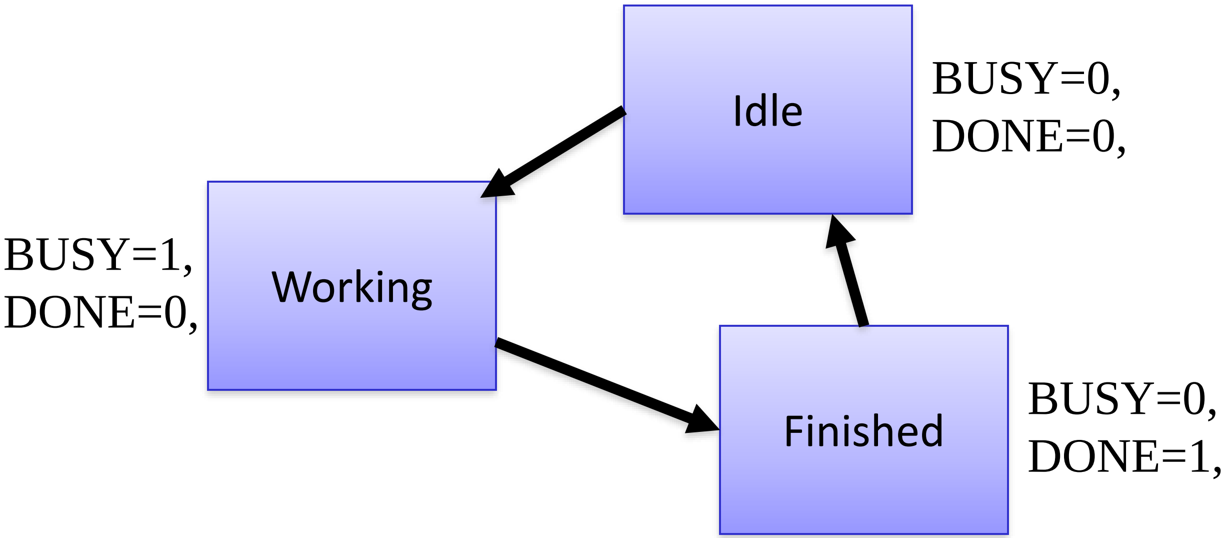
- Control flow:
  - An I/O system call traps to the kernel, invoking the trap handler for I/O (the device manager), which indexes into a table using the arguments provided to run the correct device driver

Device Controller Interface



busy	done	
0	0	idle
0	1	finished
1	0	working
1	1	(undefined)

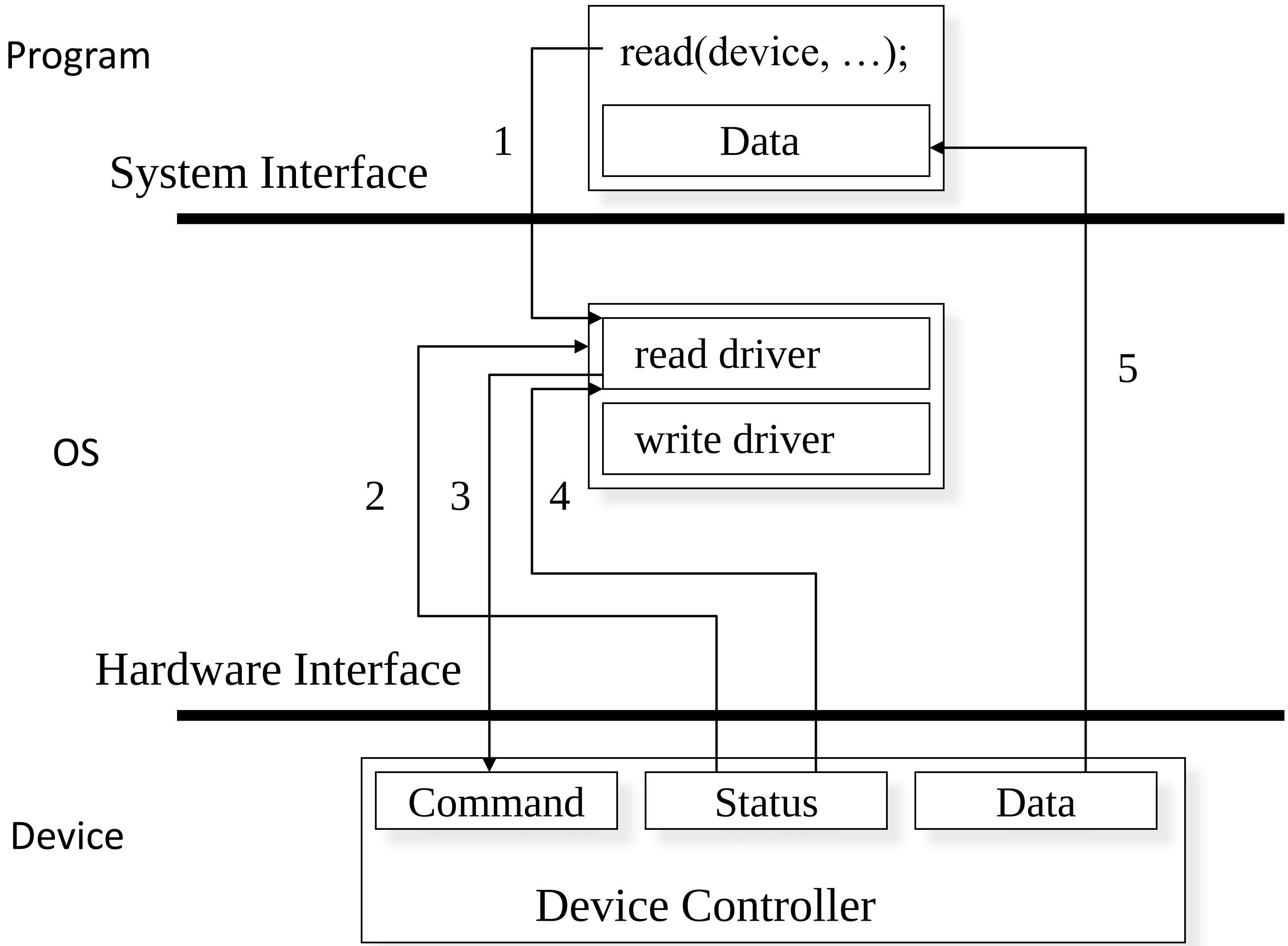
Device Controller States



- Need three states to distinguish the following:
  - Idle: no app is accessing the device
  - Working: one app only is accessing the device
  - Finished: the results are ready for that one app

- Therefore, need 2 bits for 3 states:
  - A BUSY flag and a DONE flag
  - BUSY=0, DONE=0 => Idle
  - BUSY=1, DONE=0 => Working
  - BUSY=0, DONE=1 => Finished
  - BUSY=1, DONE=1 => Undefined

Example: Polling I/O Read Operation



Polling I/O: A Write Example

```
                                                  BUSY  DONE
while(deviceN.busy || deviceN.done) <waiting>;      *     *
                                  _____________________
                                                    0     0
deviceN.data[0] = <value to write>

deviceN.command = WRITE;          _____________________
                                                    1     0
while(deviceN.busy) <waiting>;    _____________________
                                                    0     1
/* finished, so set status bits… */
deviceN.done = FALSE;             _____________________
                                                    0     0
```

Polling I/O – Problem

- Note that the OS is spinning in a loop twice:
  - Checking for the device to become idle
  - Checking for the device to finish the I/O request, so the results can be retrieved
  - Busy waiting: this wastes CPU cycles that could be devoted to executing applications
- Instead, want to *overlap* CPU and I/O
  - Free up the CPU while the I/O device is processing a read/write

Device Manager I/O Strategies

- Underneath the blocking/non-blocking synchronous/asynchronous system call API,
  OS can implement several strategies for I/O with devices

  - direct I/O with polling
    - the OS device manager busy-waits, we've already seen this
  - direct I/O with *interrupts*
    - More efficient than busy waiting
  - DMA with interrupts