Chapter 13:
Device I/O, Interrupts, DMA

CSCI 3753 Operating Systems

Instructor: Chris Womack

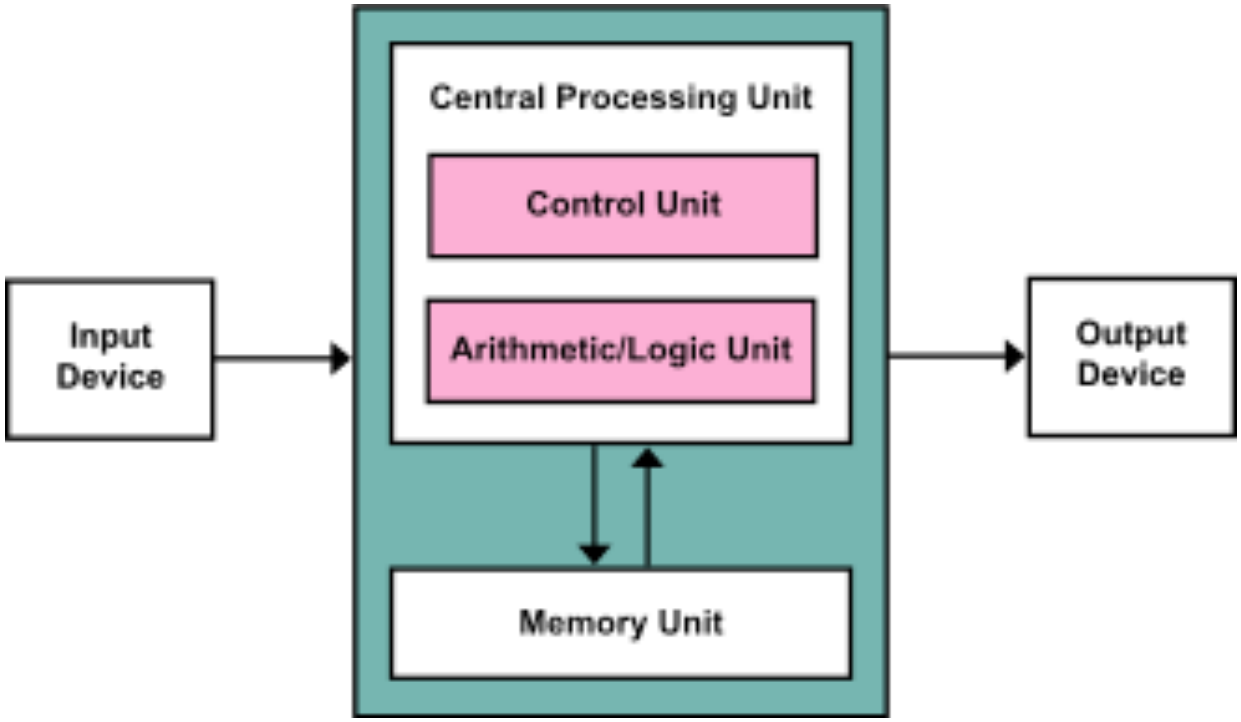
CSCI 3753 Announcements

- PA #1 is up so check Moodle and Github
  - Add a system call to Linux
- Read Chapter 13 (I/O Systems) and Chapter 2 in the textbook

Recap

1. Mode bit for kernel mode vs. user mode
2. System calls
   - Special *trap* assembly instruction invokes OS, passing the system call #
   - Trap table
   - Different classes
3. Efficient CPU usage
   - Batch mode multiprogramming
   - Multitasking
     - Cooperative
     - Preemptive via *periodic timer interrupts*
     - Context switch overhead

Von Neumann Computer Architecture

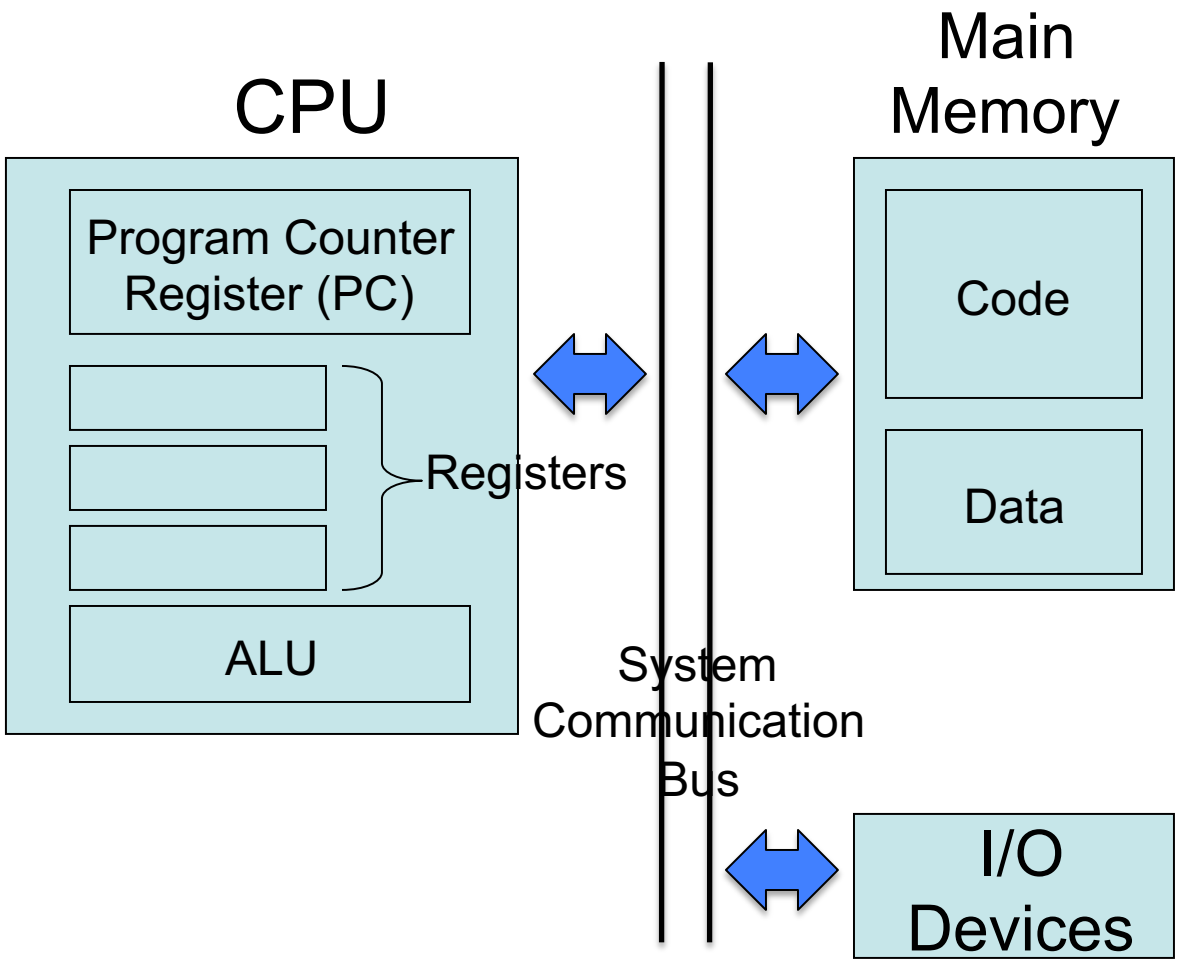


In 1945, von Neumann described a “stored-program” digital computer in which memory stored both instructions *and* data

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

Turing had described this concept earlier(1936), as did Zuse (1936) and Eckert and Mauchly (1943/44).

Von Neumann Computer Architecture

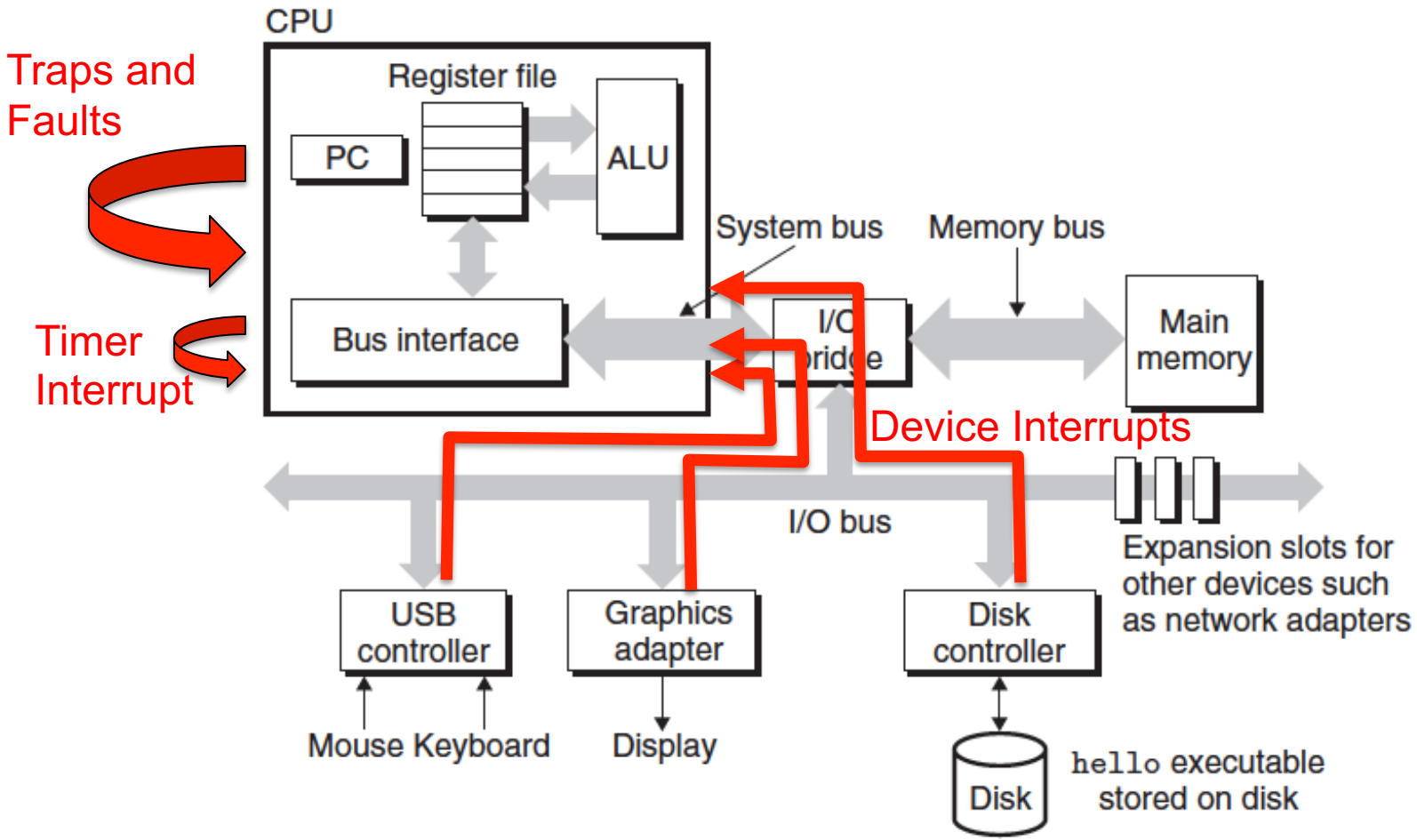


Want to support more devices: card reader, magnetic tape reader, printer, display, disk storage, etc.

System bus evolved to handle multiple I/O devices.

Includes control, address and data buses

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	nonrecover-able error	Hardware bus failure	never returns, synchronous

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

Intel Pentium Processor Event-Vector Table

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Examples of x86 Exceptions

Exception Table

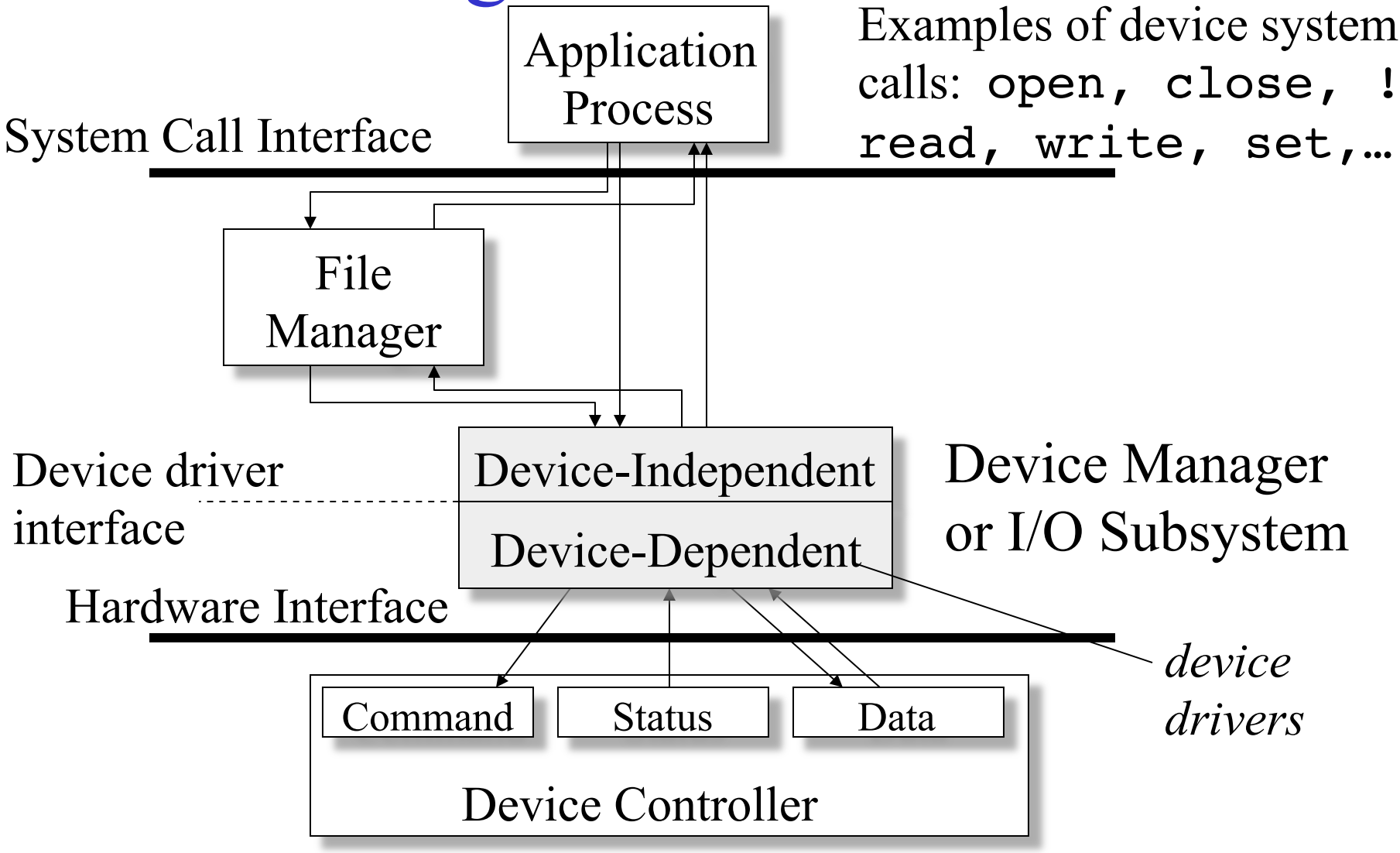
Exception Number	Description	Exception Class	Pointer to Handler
0	Divide error	fault	---
13	General protection fault	fault	---
14	Page fault	fault	---
18	machine check	abort	---
32-127	OS-defined	Interrupt or trap	---
128	System call	Trap	---
129-255	OS-defined	Interrupt or trap	---

0-31 reserved for hardware

OS assigns

offsets form *interrupt vector*

Device Management Organization



Device System Call Interface

- Creates 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!`
  - Other functions as needed, e.g. `seek` for random access devices...
- Not every function will be fully realized for certain I/O devices
  - e.g. devices that are exclusively input would not need to be written to

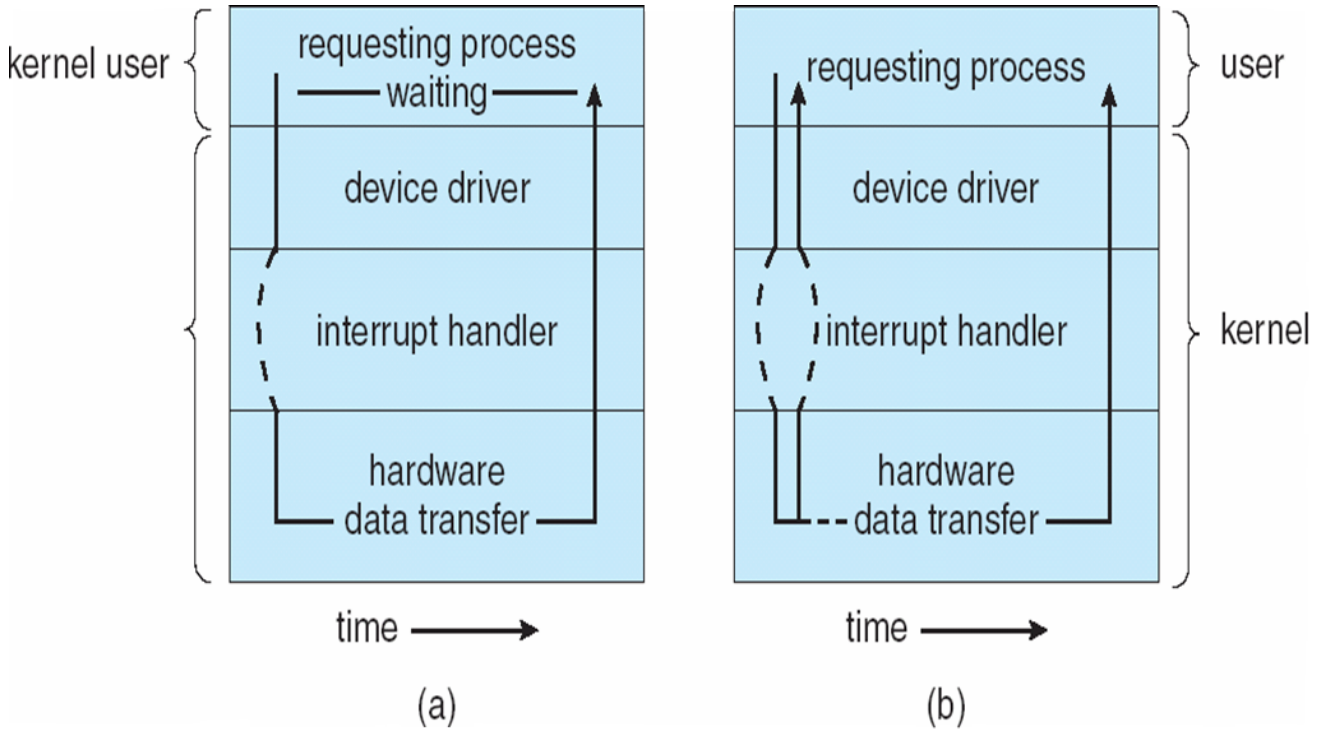
Device System Call Interface

- Block vs character devices
  - Character devices generate or process data as a linear stream of bytes
    - e.g. keyboards, mice, audio, modems, printers
  - Block devices read/write data in discrete blocks
    - e.g. most storage devices like disks
- Sequential vs direct/random access
  - Some devices like magnetic tapes are best suited for sequentially reading or writing the medium
  - Other devices like magnetic/solid-state disks support random access, i.e. the read and writes can access any part of disk in any order

Device System Call Interface

- Blocking versus Non-Blocking I/O
  - blocking system call: process put on wait queue until I/O read or write completes
  - non-blocking system call: a write or read returns immediately with partial number of bytes transferred (possibly zero), e.g. keyboard, mouse, network sockets
    - Makes the application more complex, because not all the data may have been read or written – have to add additional code to handle this, like a loop
- Synchronous versus asynchronous
  - asynchronous returns immediately, *but at some later time, the full number of bytes requested is transferred* – subtle difference with non-blocking definition
  - Many use synchronous and blocking interchangeably, and asynchronous and non-blocking interchangeably

Two I/O Methods



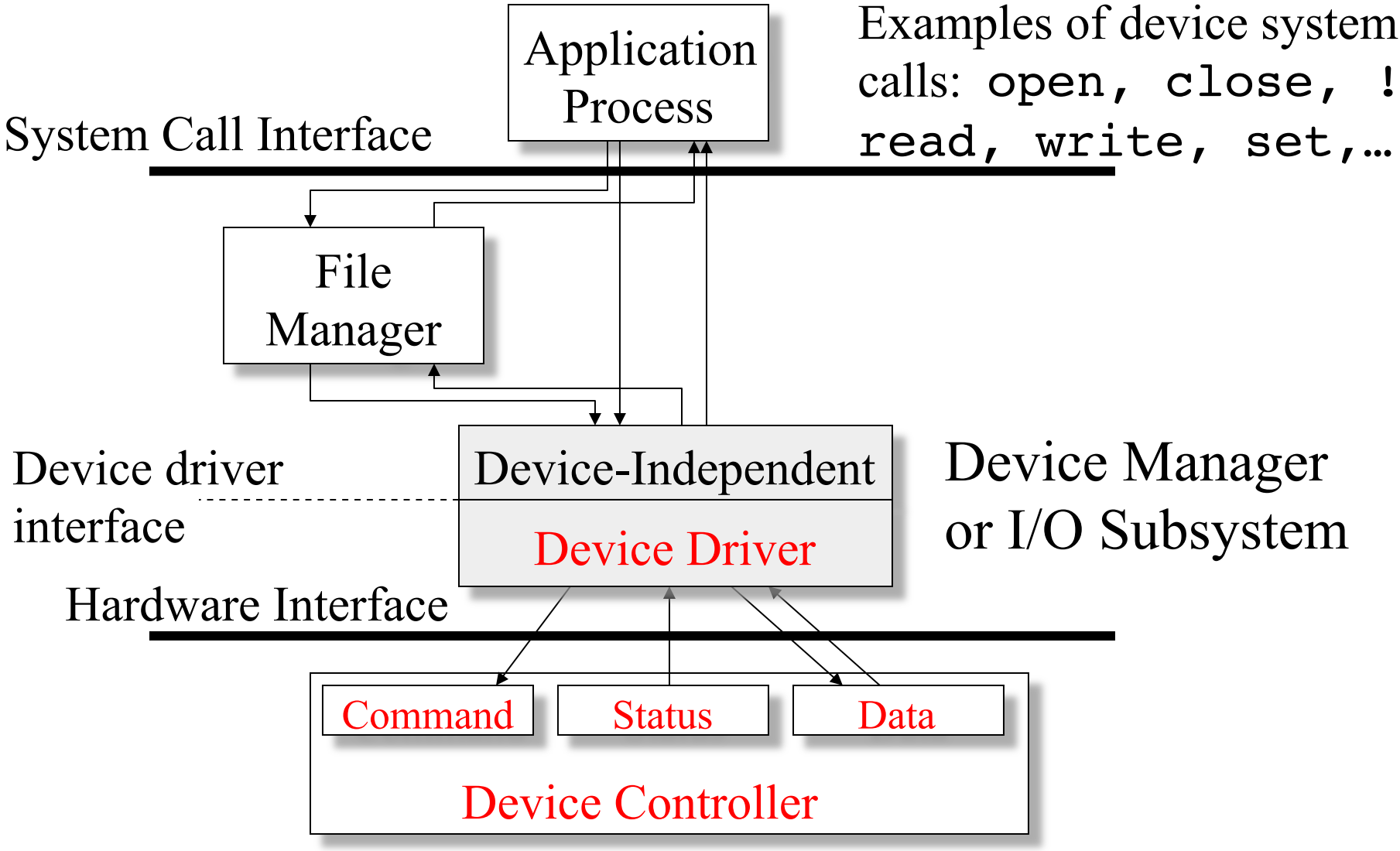
Synchronous"

Asynchronous"

Queuing of Device Requests

- More than one application may want access to the same device
  - OS keeps a queue for each device.
  - Read/write requests for this device are placed in this queue
  - Queue is often FIFO
  - For printers, queuing has the special term *spooling*
- Device requests may be reordered in some cases
  - Reorder requests to satisfy a performance criterion
    - Speed of access to a mechanical device, e.g. disk, may be improved by grouping reads/writes to the same location on the device
  - Priority of an application

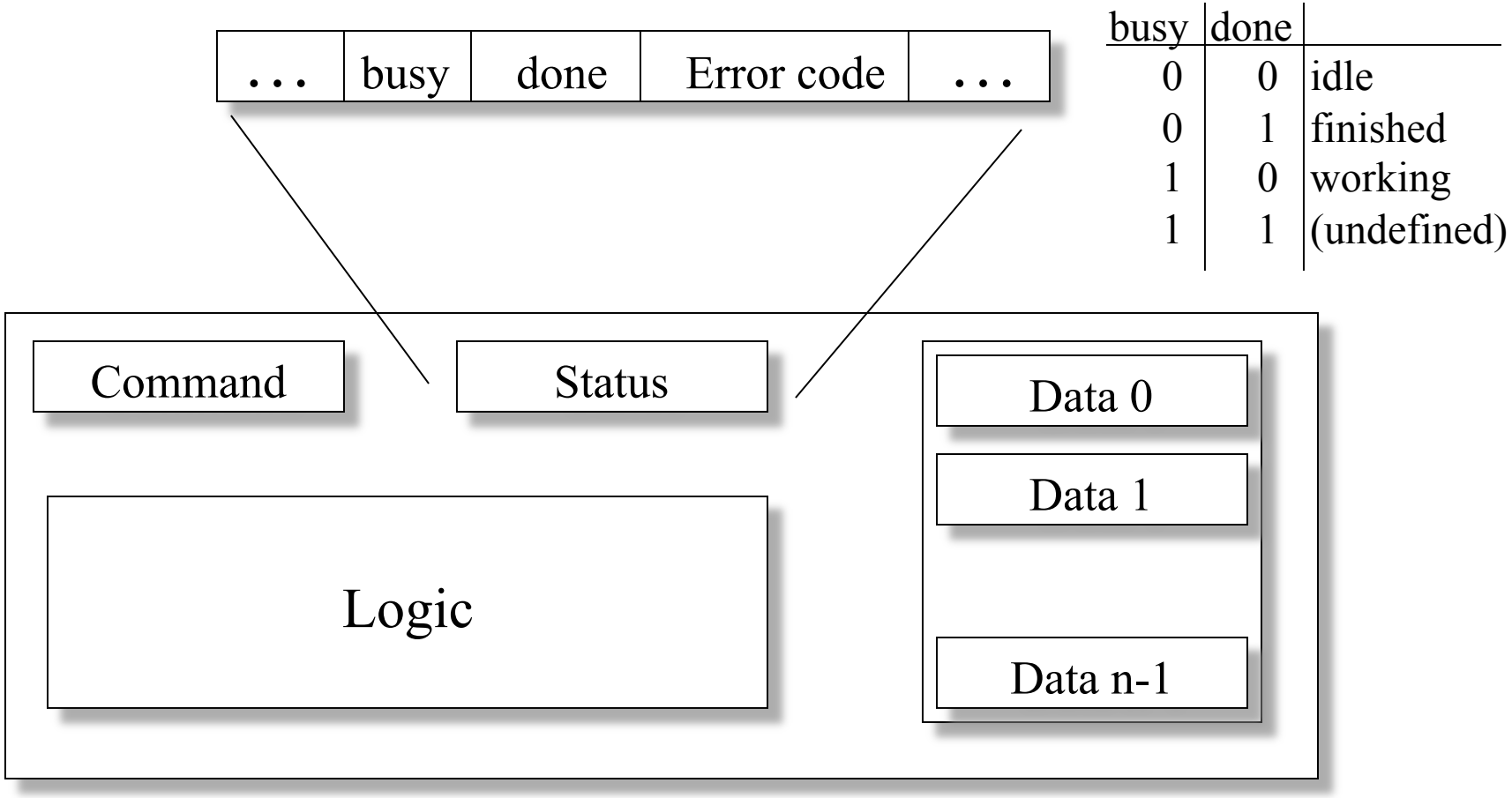
Device Drivers



Device Drivers

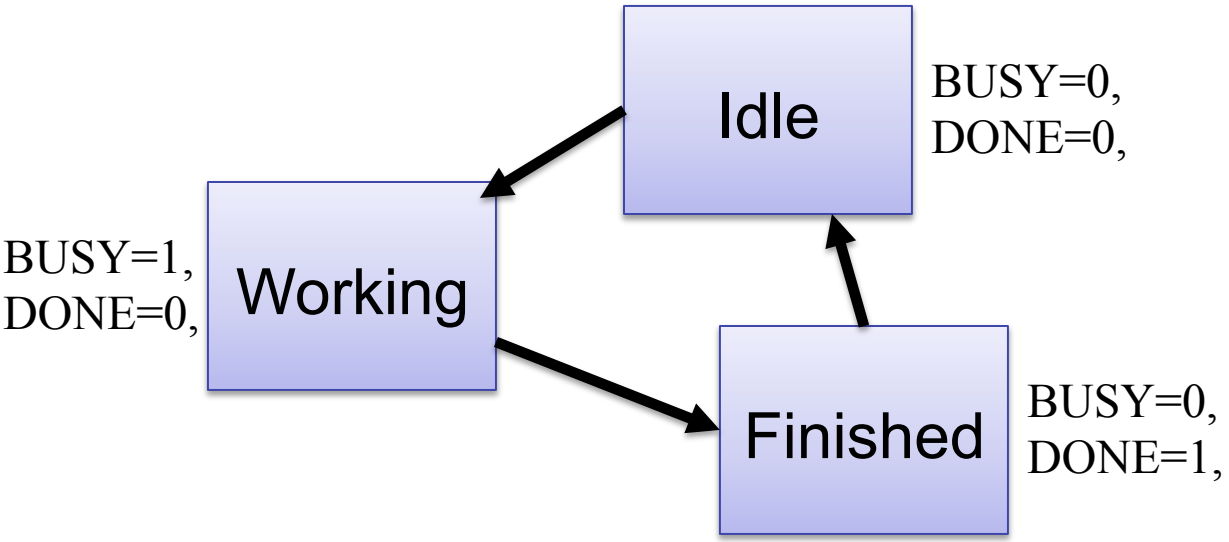
- Support the device system call interface functions `open, read, write`, etc. for that device
- Interact directly with the device controllers
  - Know the details of what commands the device can handle, how to set/get bits in device controller registers, etc.
  - Are part of the device-dependent component 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



- 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

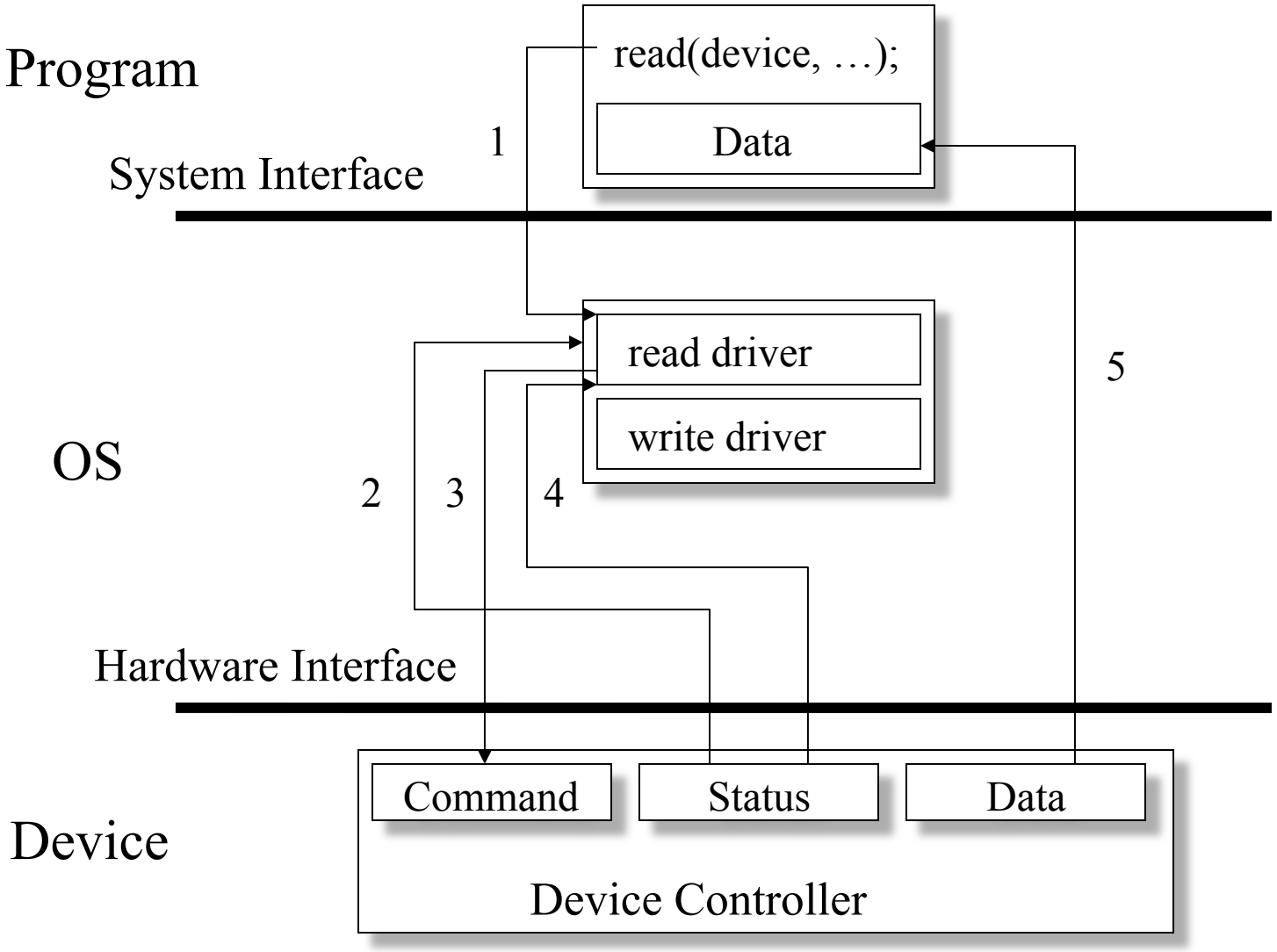
- Need three states to distinguish the following:
  - Idle: no app is accessing the device
  - Working: one app only is accessing the device
  - Finished: the results are ready for that one app

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

- Devices *much* slower than CPU
- CPU waits while device operates
- Would like to multiplex CPU to a different process while I/O is in process
  - Actual OS loop would alternate between checking and doing other useful work if flags not right

Polling I/O Read Operation



Polling I/O – Busy Waiting

- 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
  - If N devices with requests pending, have to poll all N devices
  - 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

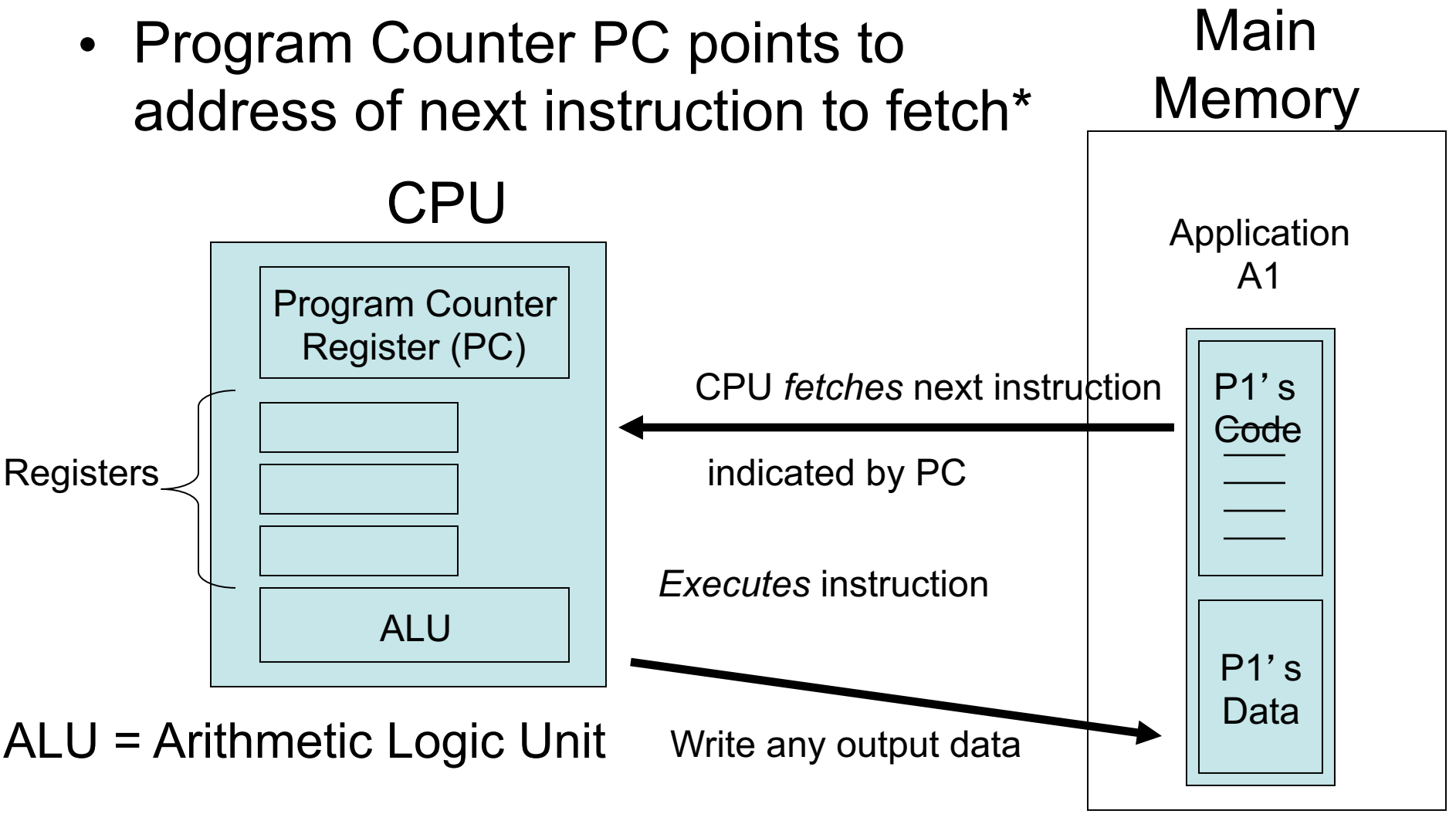
- Beneath the system call API (blocking/non-blocking or synchronous/asynchronous), OS can implement several strategies for I/O with devices
  1. direct I/O with polling
     - the OS device manager busy-waits, we've already seen this
  2. direct I/O with *interrupts*
     - More efficient than busy waiting
  3. DMA with interrupts

Hardware Interrupts

- CPU incorporates a *hardware interrupt flag*
- Whenever a device is finished with a read/write, it communicates to the CPU and raises the flag
  - Frees up CPU to execute other tasks without having to keep polling devices
- Upon an interrupt, the CPU interrupts normal execution, and invokes the OS' s *interrupt handler*
  - Eventually, after the interrupt is handled and the I/O results processed, the OS resumes normal execution

Standard CPU Execution

- Program Counter PC points to address of next instruction to fetch*



ALU = Arithmetic Logic Unit

* PC can alternatively point to the current instruction, depending on the CPU

CPU Execution with Interrupts

- *CPU checks Interrupt Flag every instruction*

CPU Pseudocode

- While (no hardware failure)
  - Fetch next instruction, put in instruction register
  - Execute instruction
  - Check for interrupt: If interrupt flag enabled,
    - Save PC*
    - Jump to interrupt handler

* from Nutt's text

Interrupt Handler

1. First, save the processor state
   - Save the executing app' s program counter (PC) and CPU register data
2. Next, find the device causing the interrupt
   - Consult interrupt controller to find the interrupt offset, or poll the devices
3. Then, jump to the appropriate device handler
   - Index into the Interrupt Vector using the interrupt offset
   - An Interrupt Service Routine (ISR) either refers to the interrupt handler, or the device handler
4. Finally, reenable interrupts (see later slides)

Examples of Exceptions in x86 Pentium Systems

OS consults Exception or Interrupt Vector Table to find address of appropriate exception/interrupt/device handlers

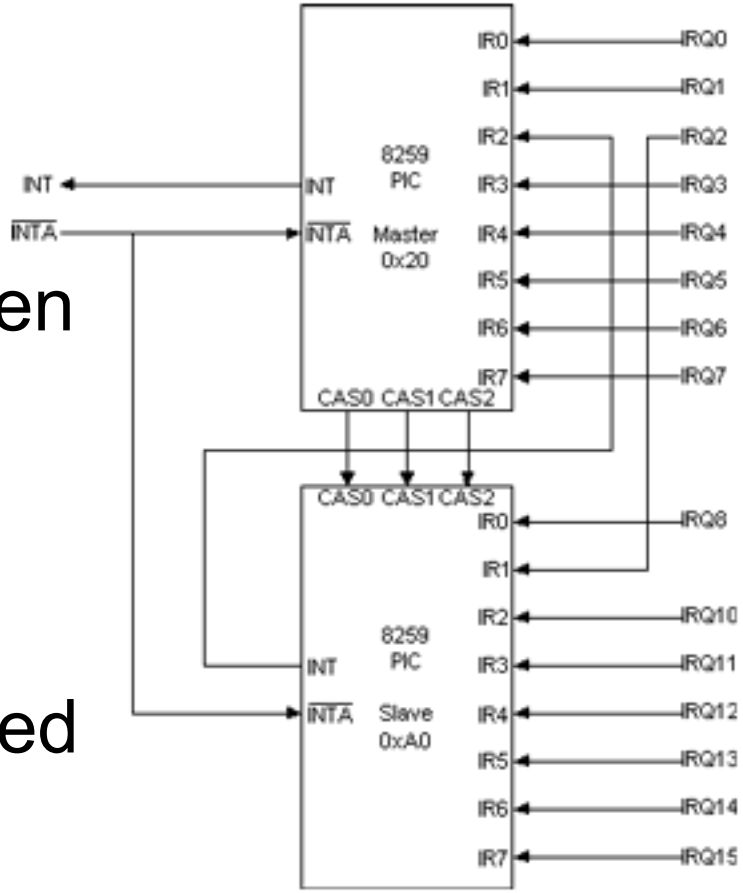
Exception Number	Description	Exception Class	Address of handler
0	Divide error	fault	---
13	General protection fault	fault	---
14	Page fault	fault	---
18	machine check	abort	---
32-127	OS-defined	Interrupt or trap	---
128	System call	Trap	---
129-255	OS-defined	Interrupt or trap	---

OS assigns

"interrupt vector" either refers to the handler's address or the entire table, depending on definition

Interrupts From Multiple Devices

- Early x86 interrupt architecture cascades two 8259 PICs
- If any IRQ line is raised, then the INT line is raised, signaling the CPU that an interrupt has occurred
- Which device (IRQ line) caused the interrupt is stored in the data lines



Maskable Interrupts

- Maskable interrupts can be turned off by CPU before execution of critical instruction sequences
  - For example, don’ t want the interrupt handler to be interrupted while it has not yet saved the processor state
  - are used by device controllers to talk with CPU
- Nonmaskable interrupts/exceptions are reserved for events such as unrecoverable memory errors and cannot be turned off by the CPU
- Can have multiple interrupt priority levels
  - high-priority interrupts can preempt execution of a low-priority interrupt

Examples of Exceptions in x86 Pentium Systems

Exception Table

Exception Number	Description	Exception Class
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

non-maskable

OS assigns

maskable interrupts

Disabling/Enabling Interrupts

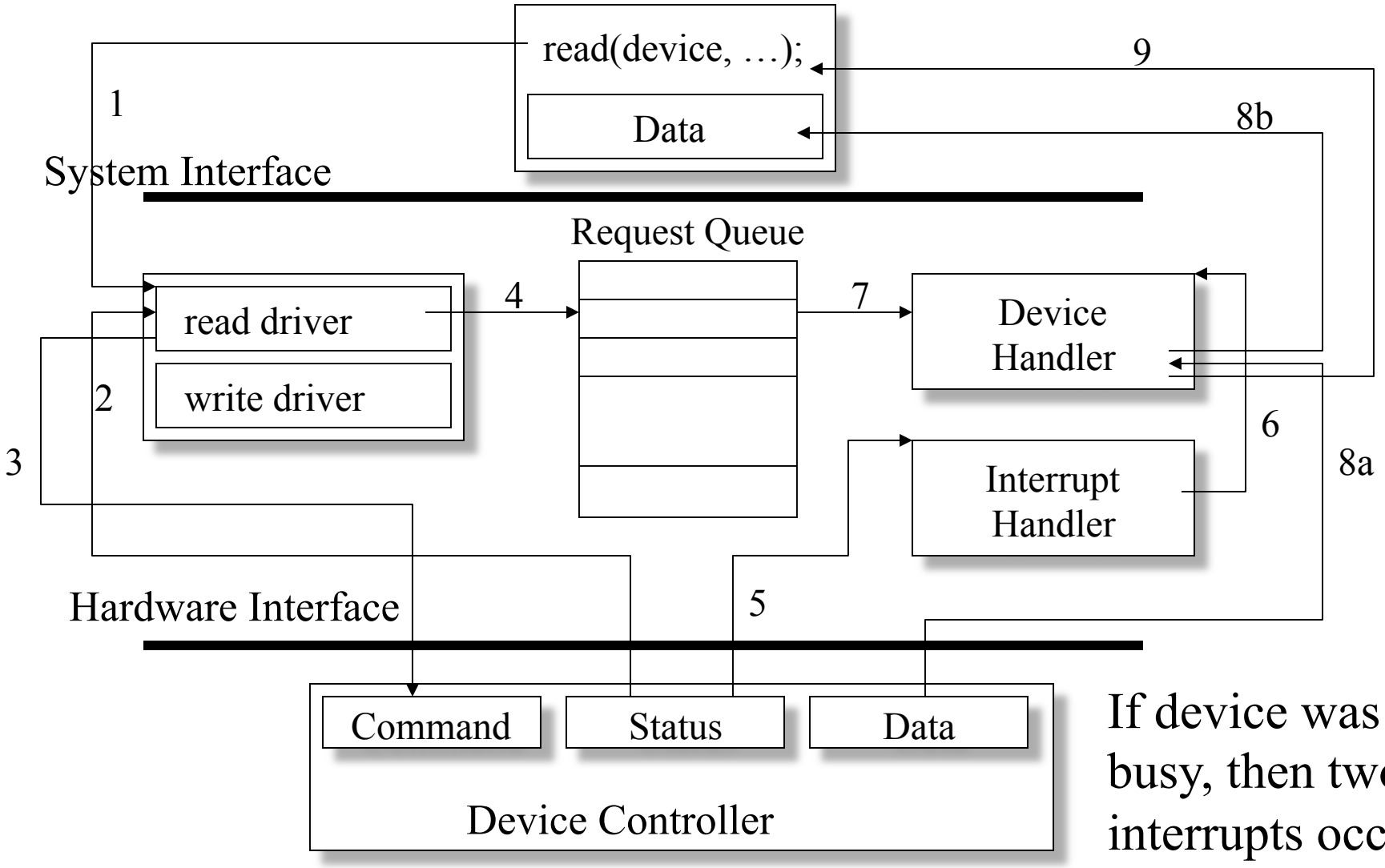
CPU Pseudocode

- While (no hardware failure)
  - Fetch next instruction, put in instruction register
  - Execute instruction
  - *If interrupts are enabled (check a 2nd flag)*
    - Check for interrupt: If interrupt flag enabled,
      - *Disable interrupts**
      - Save PC
      - Jump to interrupt handler

Note interrupt handler should reenable interrupts when done.

* See Nutt's text

Interrupt-Driven I/O Operation



If device was busy, then two interrupts occur

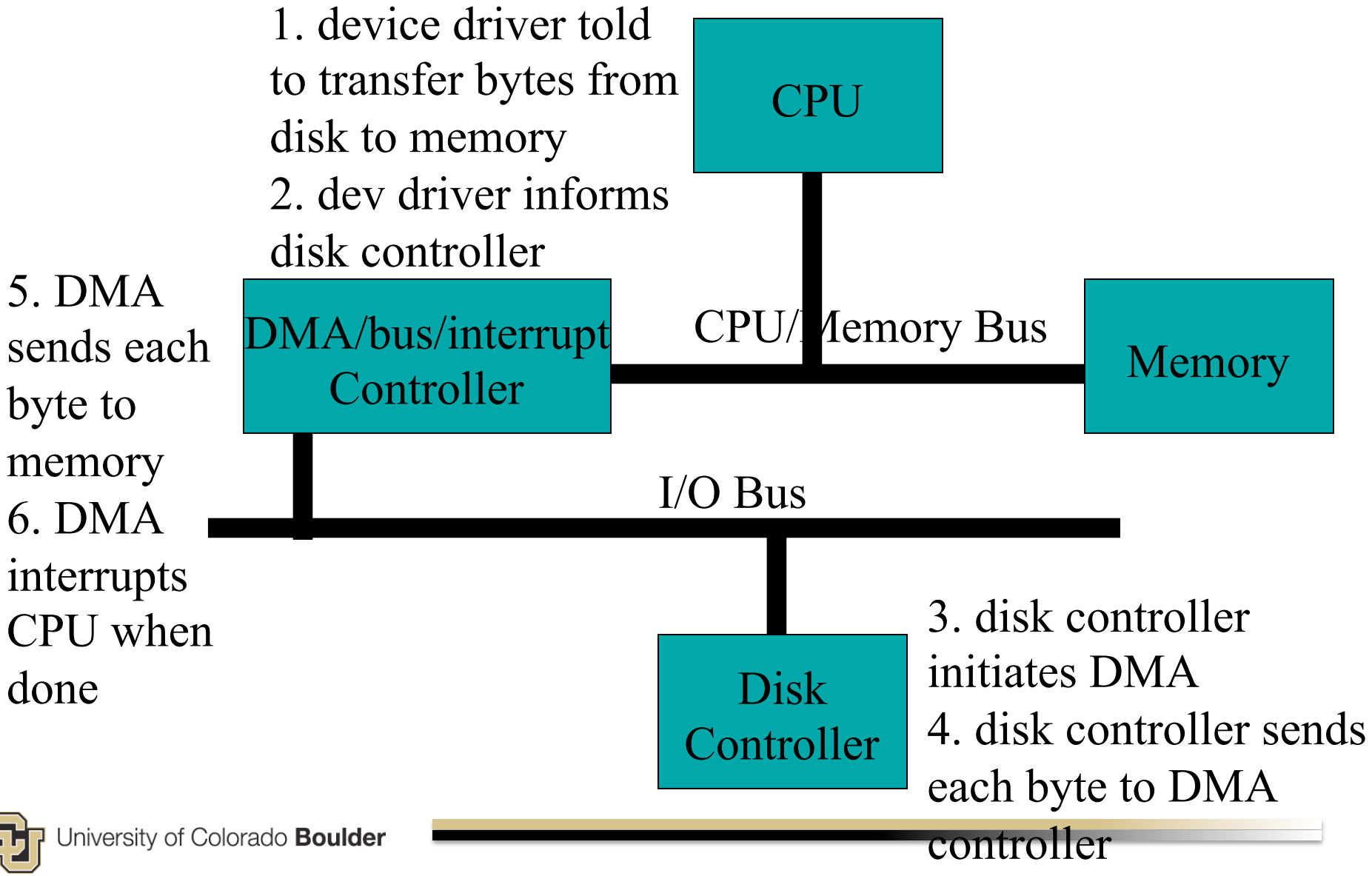
Direct Memory Access (DMA)

- The CPU can become a bottleneck if there is a lot of I/O copying data back and forth between memory and devices
- Example: want to copy a 1 MB file from disk into memory.
  - The disk is only capable of delivering memory in say 1 KB blocks through the small data register in its device controller
  - So every time a 1 KB block is ready to be copied, an interrupt is raised, interrupting the CPU a thousand times for a 1 MB file!
  - This will slow down execution of normal programs and the OS.
- Worst case: CPU could be interrupted after the transfer of every byte/character, or every packet from the network card

Direct Memory Access (DMA)

- DMA solution:
  - Bypass the CPU for large data copies
  - Only raise an interrupt at the very end of the data transfer, instead of at every intermediate block
  - Example: for a 1 MB file, only one interrupt is raised at the end of file transfer to memory, rather than 1000
- Substantially improves performance of large I/O transfers

DMA with Interrupts Example



Direct Memory Access (DMA)

- Since both CPU and the DMA controller have to move data to/from main memory, how do they share main memory?
  - Burst mode
    - While DMA is transferring, CPU is blocked from accessing memory
  - Interleaved mode or “cycle stealing”
    - DMA transfers one word to/from memory, then CPU accesses memory, then DMA, then CPU, etc… - interleaved
  - Transparent mode – DMA only transfers when CPU is not using the system bus
    - Most efficient but difficult to detect