CSCI 3753
Operating Systems

Device Management

Lecture Notes By

Shivakant Mishra

Computer Science, CU-Boulder

Last Update: 01/22/14

Bootstrapping the OS

- Multi-stage procedure:
  1. Power On Self Test (POST) from ROM
     - Check hardware, e.g. CPU and memory, to make sure it's OK
  2. BIOS (Basic Input/Output System) looks for a device to boot from…
     - May be prioritized to look for a USB flash drive or a CD/DVD-ROM drive before a hard disk drive
     - Can also boot from network

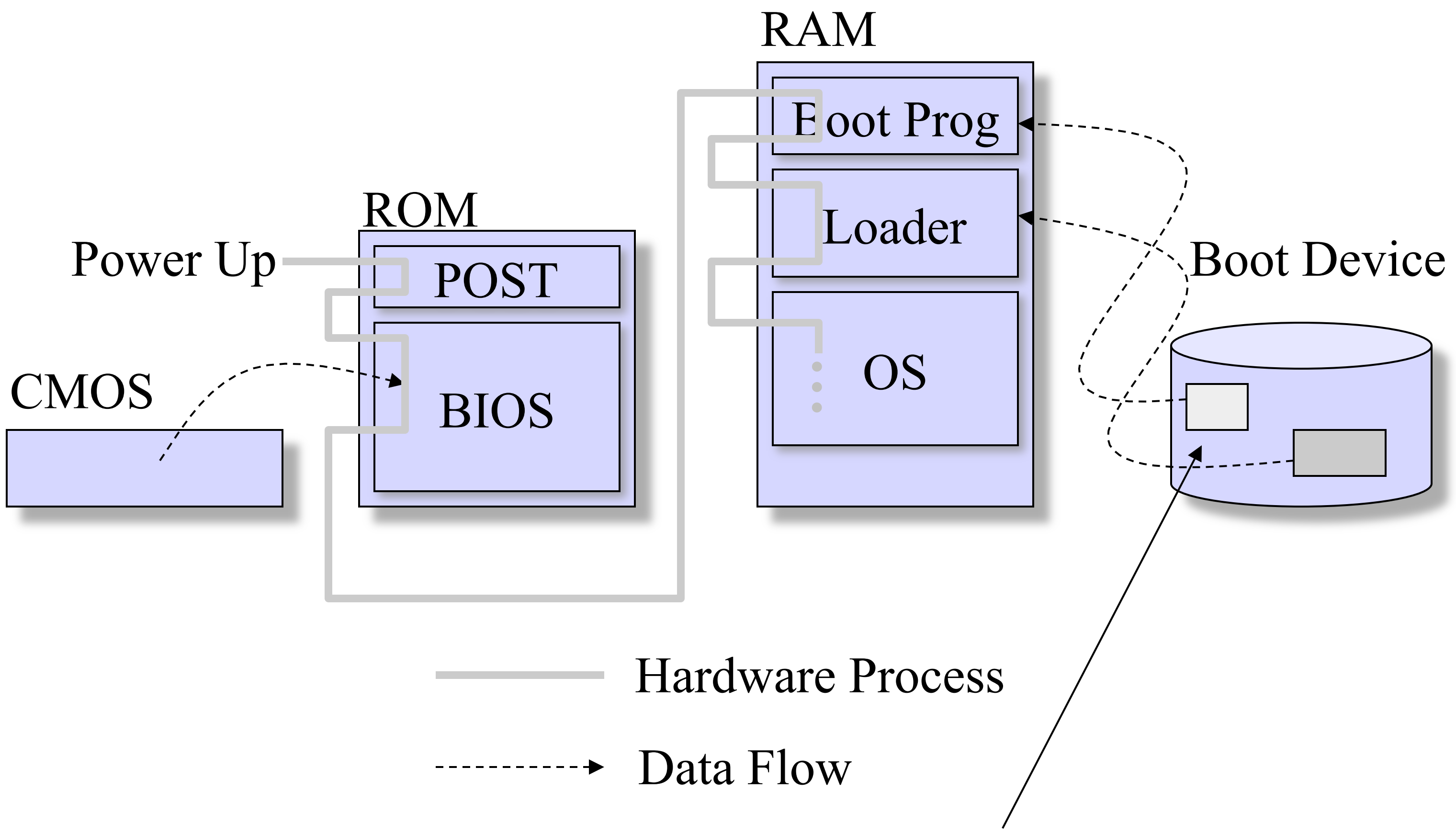
Bootstrapping the OS (2)

- Multi-stage procedure: (continued)
  3. BIOS finds a hard disk drive to boot from
     - Looks at Master Boot Record (MBR) in sector 0 of disk
     - Only 512 bytes long (Intel systems), contains primitive code for later stage loading and a partition table listing an active partition, or the location of the bootloader

Bootstrapping the OS (3)

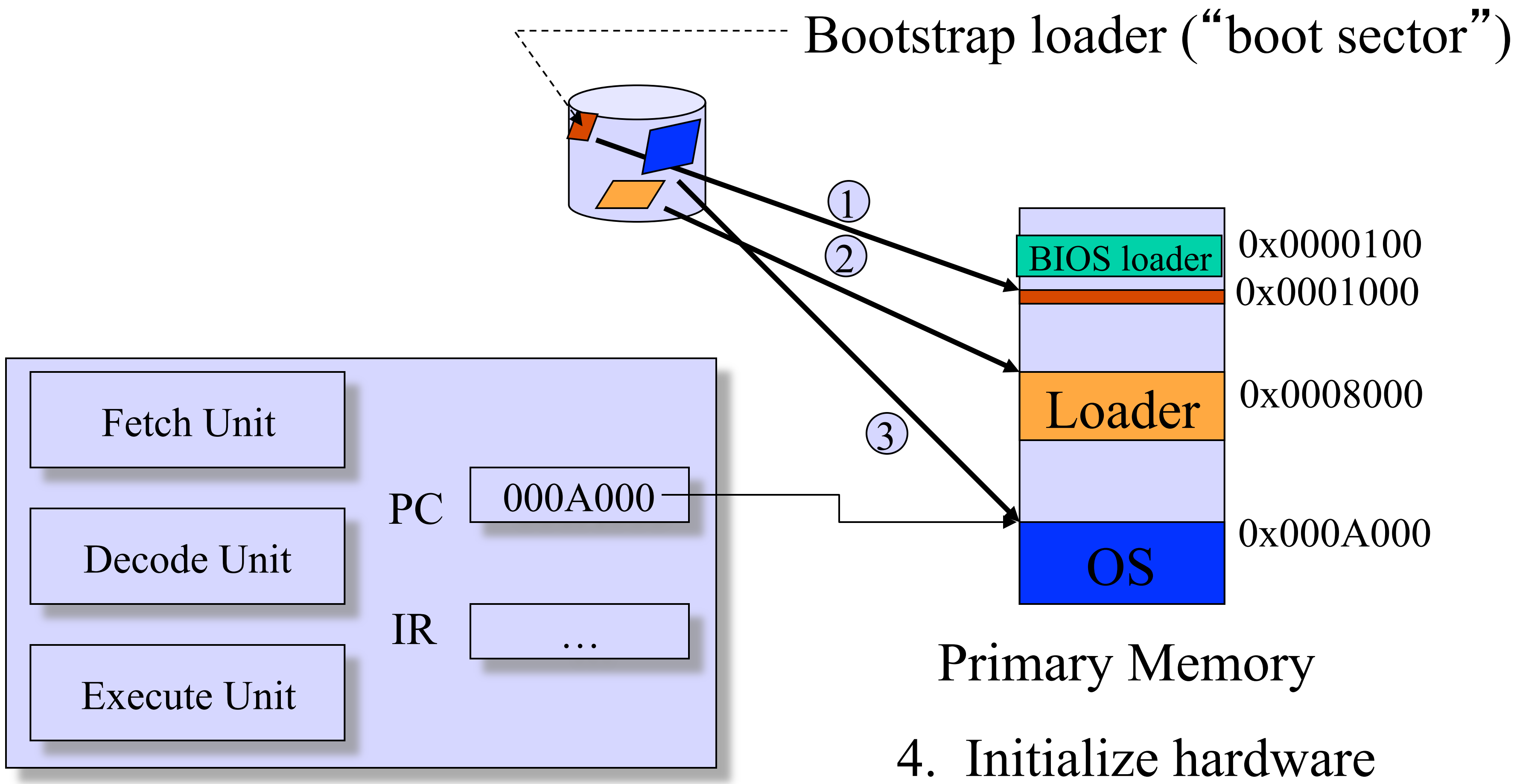
- Multi-stage procedure: (continued)
  4. Primitive loader then loads the secondary stage bootloader
     - Examples of this bootloader include LILO (Linux Loader), and GRUB (Grand Unified Bootloader)
     - Can select among multiple OS's (on different partitions) – i.e. dual booting
     - Once OS is selected, the bootloader goes to that OS's partition, finds the boot sector, and starts loading the OS's kernel

Intel System Initialization



•

Bootstrapping Example



4. Initialize hardware
5. Create user environment
6. …

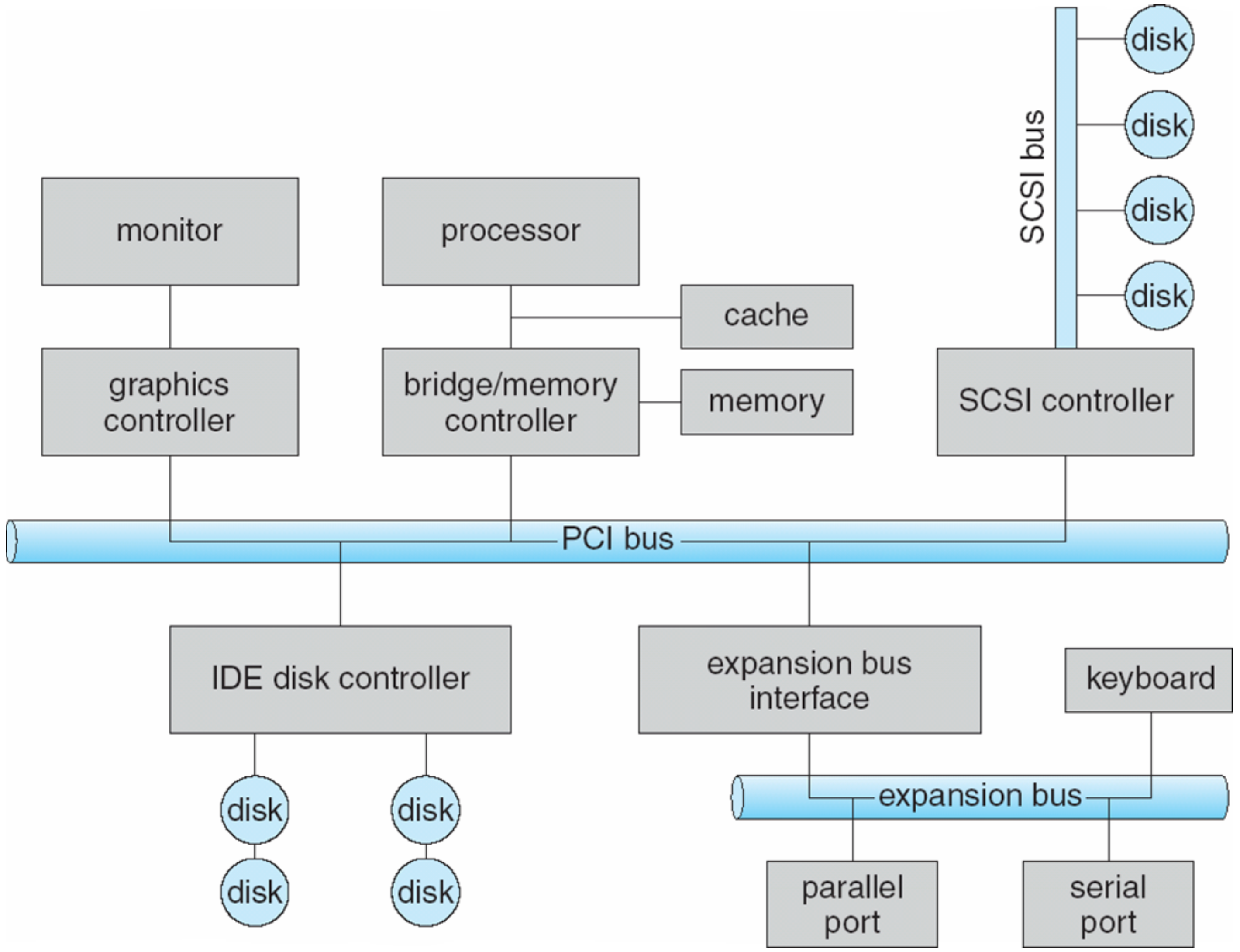
Device Manager

- Controls the 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.

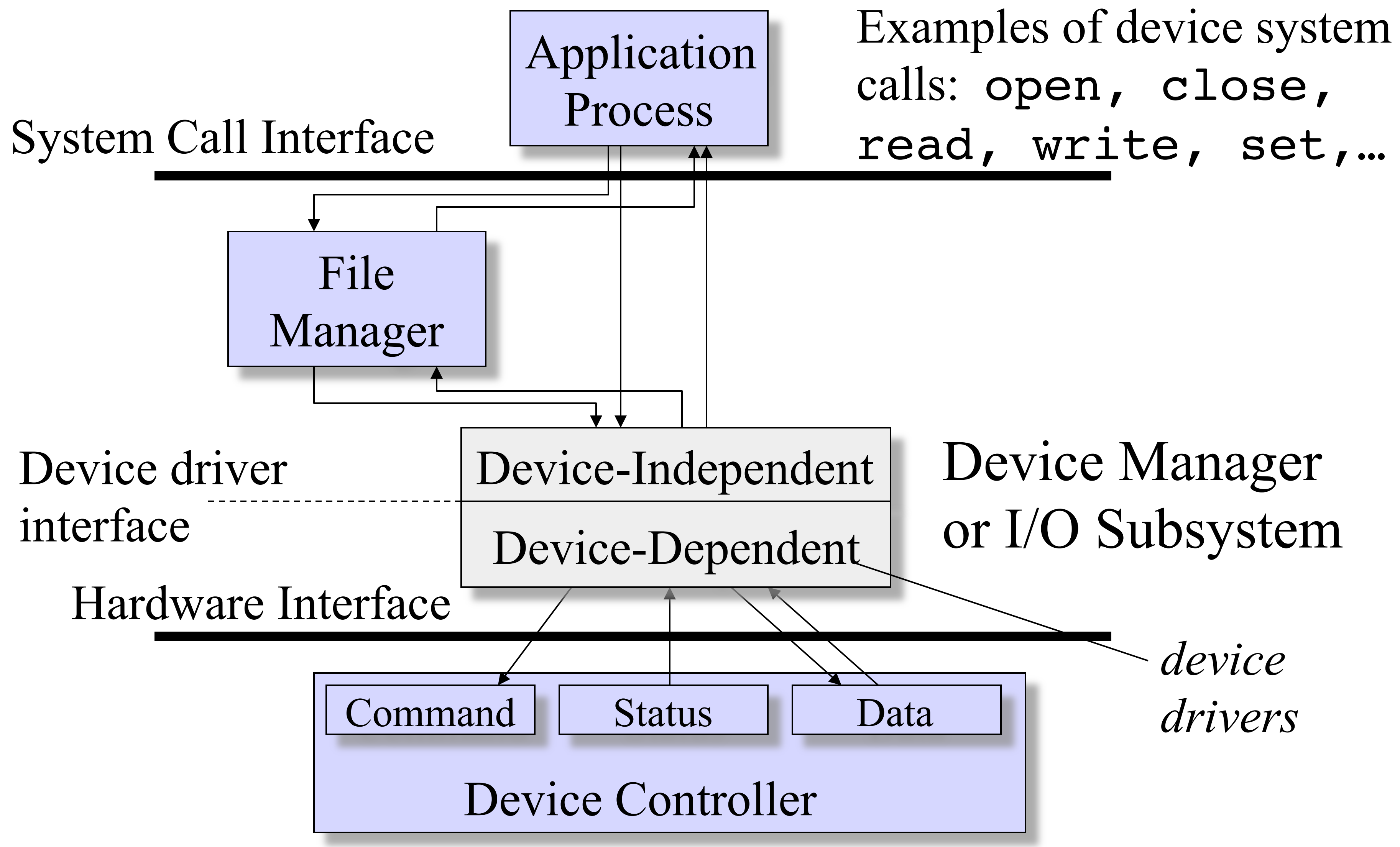
Device Characteristics

- I/O devices consist of two high-level components
  - Mechanical component
  - Electronic component: device controllers
- OS deals with device controllers

A Typical PC Bus Structure



Device Management Organization



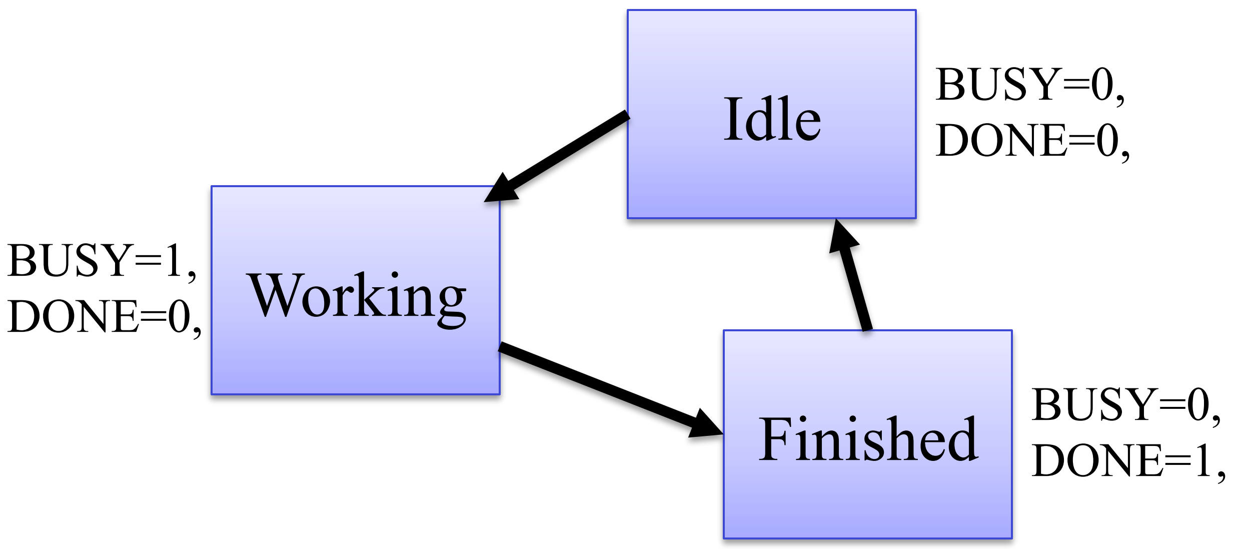
Device System Call Interface

- 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.
  - Block vs character
  - Sequential vs direct/random access
  - Blocking 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 sockets
  - Synchronous versus asynchronous
    - asynchronous returns immediately, but at some later time, the full number of bytes requested is transferred

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

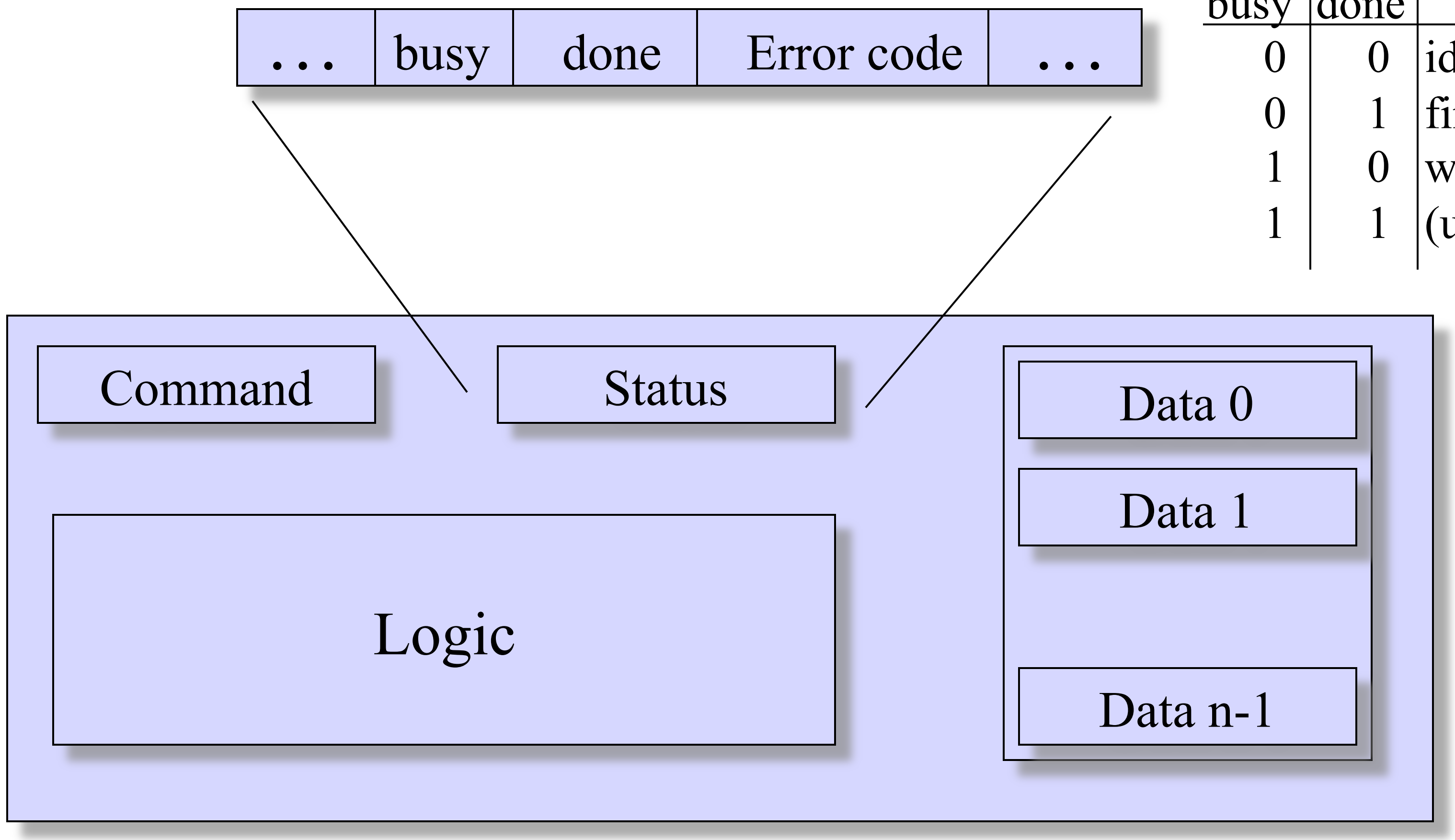
Device Controller States



- 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

Device Controller Interface



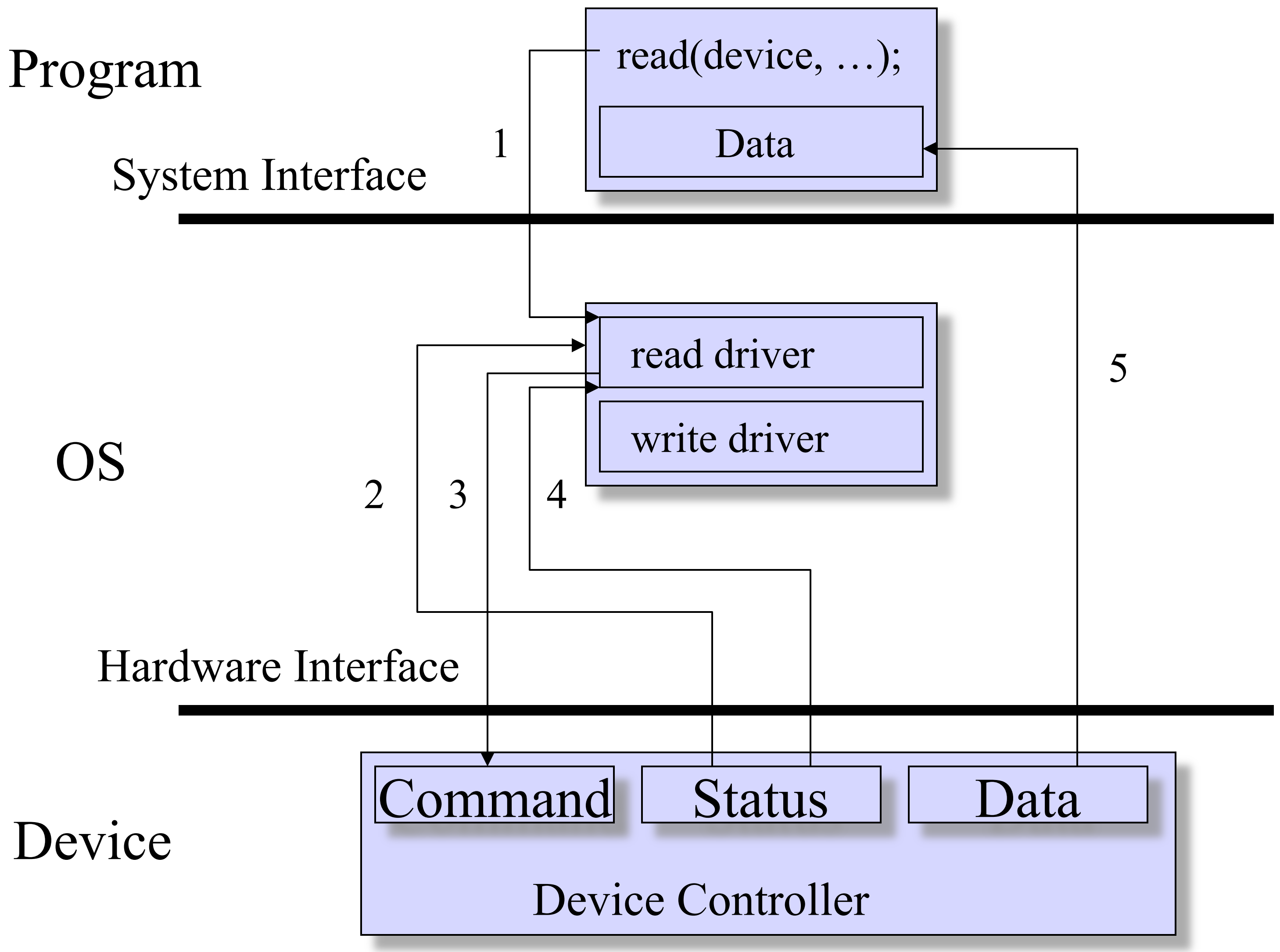
busy	done	
0	0	idle
0	1	finished
1	0	working
1	1	(undefined)

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 read some status bits… */
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

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
  - 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

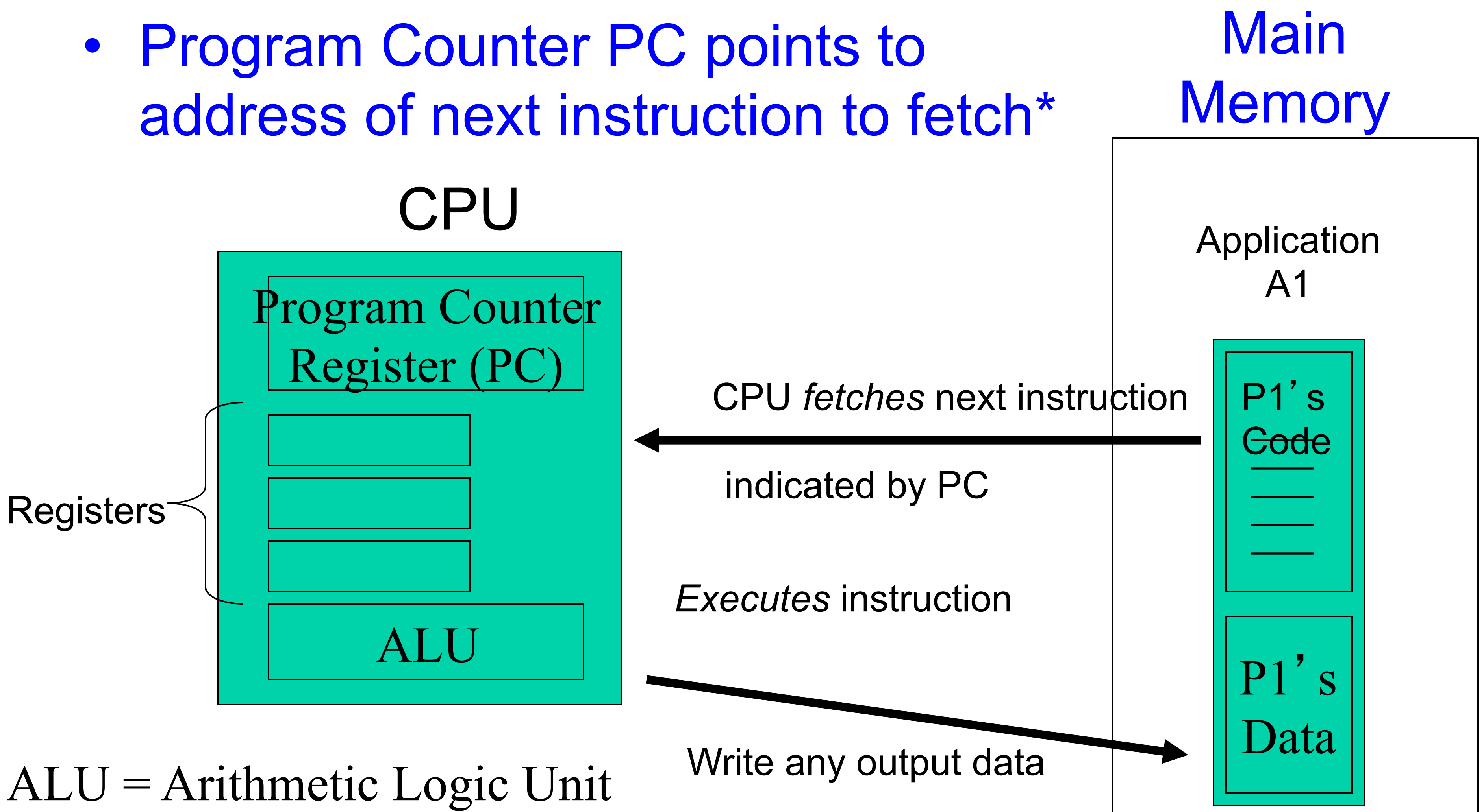
- Underneath the blocking/non-blocking synchronous/asynchronous system call API, OS can implement several strategies for I/O with devices
  - direct I/O with polling
    - the OS device manager busy-waits, we've already seen this
  - direct I/O with *interrupts*
    - More efficient than busy waiting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

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

CPU Execution of a Program

- 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 Checks Interrupt Flag Every Fetch/Execute Cycle

CPU Pseudocode

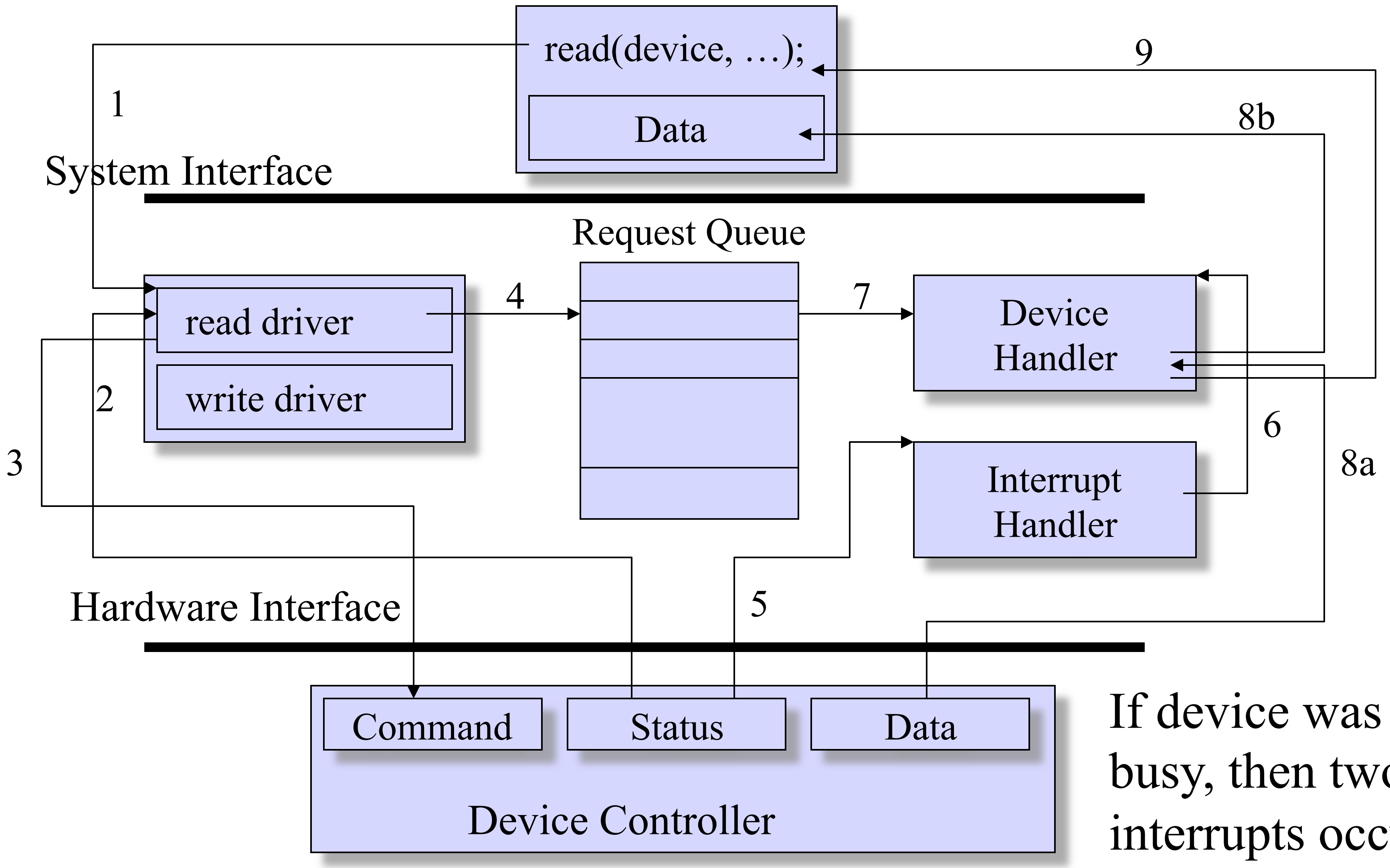
- While (no hardware failure)
  - Fetch next instruction, put in instruction register
  - Execute instruction
  - Check for interrupt: If interrupt flag enabled,
    - Save PC*
    - Jump to interrupt handler

* insight from Nutt's text

Interrupt Handler

- First, save the processor state
  - Save the executing app's program counter (PC) and CPU register data
- Next, find the device causing the interrupt
  - Consult interrupt controller to find the interrupt offset, or poll the devices
- 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
- Finally, reenable interrupts

Interrupt-Driven I/O Operation



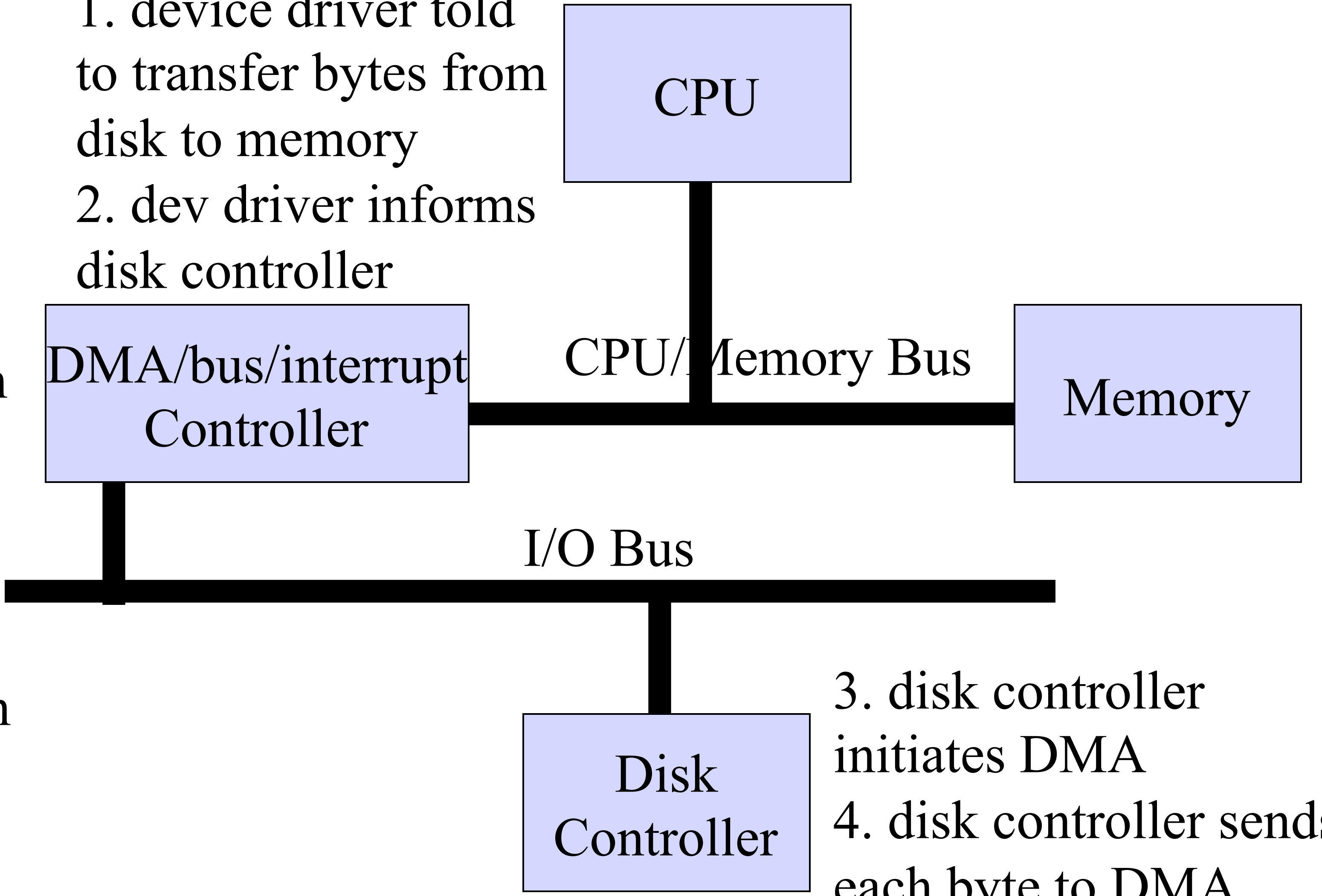
If device was busy, then two interrupts occur

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. So every time a 1 KB block is ready to be copied, an interrupt is raised, interrupting the CPU. This will slow down execution of normal programs and the OS.
  - Worst cases: CPU could be interrupted after the transfer of every byte/character, or every packet from the network card
- DMA solution: Bypass the CPU for large data copies, and only raise an interrupt at the very end of the data transfer, instead of at every intermediate block

DMA with Interrupts Example

1. device driver told to transfer bytes from disk to memory
2. dev driver informs disk controller



3. disk controller initiates DMA
4. disk controller sends each byte to DMA controller
5. DMA sends each byte to memory
6. DMA interrupts CPU when done

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

Memory-Mapped I/O

- Non-memory mapped (port or port-mapped) I/O typically requires special I/O machine instructions to read/write from/to device controller registers
  - e.g. on Intel x86 CPUs, have IN, OUT
    - Example: OUT dest, src (using Intel syntax, not Gnu syntax)
      - Writes to a device port dest from CPU register src
    - Example: IN dest, src
      - Reads from a device port src to CPU register src
    - Only OS in kernel mode can execute these instructions
    - Later Intel introduced INS, OUTS (for strings), and INSB/INSW/INSD (different word widths), etc.

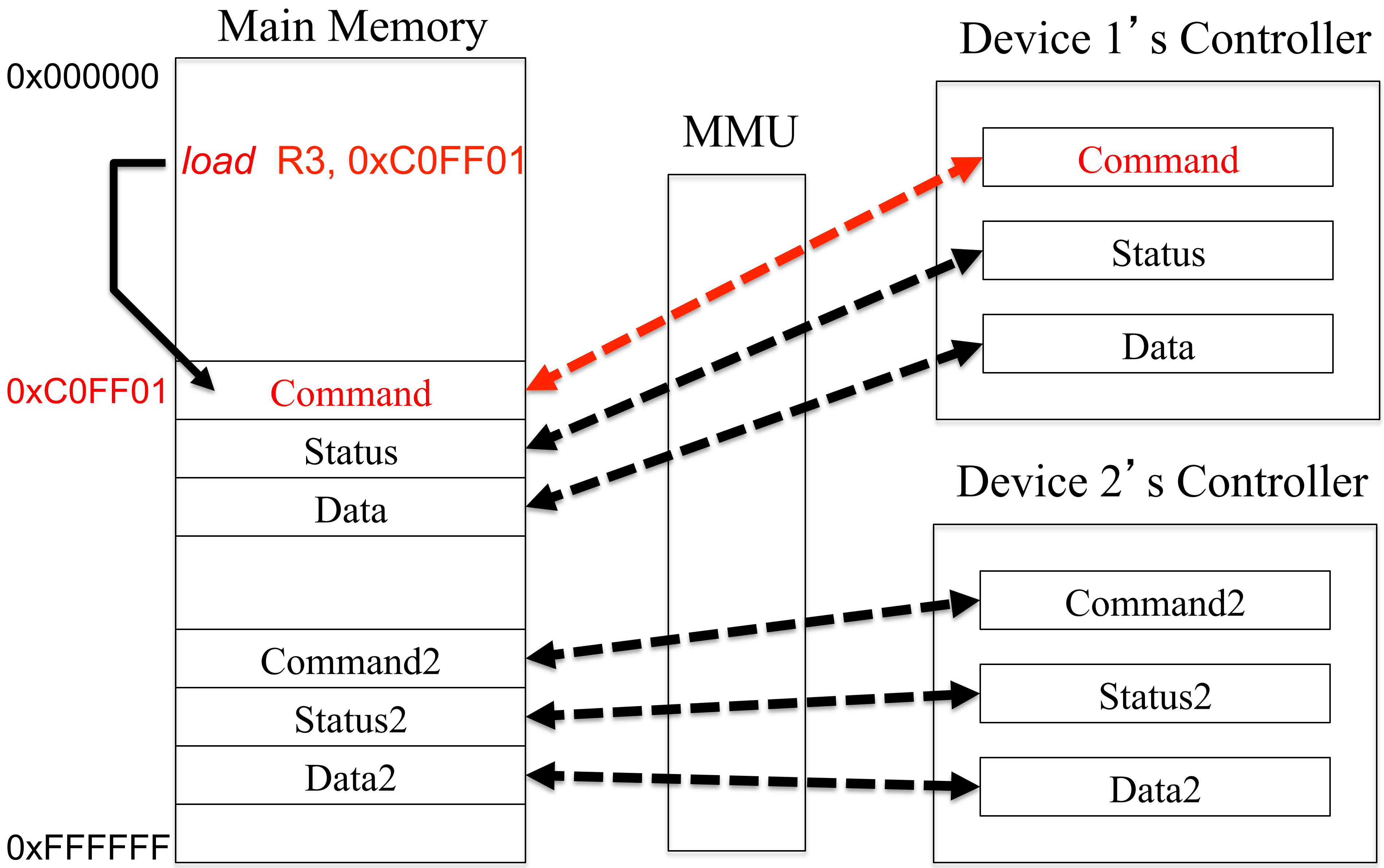
Memory-Mapped I/O (2)

- port-mapped I/O is quite limited
  - IN and OUT can only store and load
  - don’t have full range of memory operations for normal CPU instructions
    - Example: to increment the value in say a device’s data register, have to copy register value into memory, add one, and copy it back to device register.
      - With memory-mapped I/O, can increment value in memory directly, and it gets reflected into the device controller’s data register automatically
    - AMD did not extend the port I/O instructions when defining the x86-64

Memory-Mapped I/O (3)

- With memory-mapped I/O, just address memory directly using normal instructions to speak to an I/O address
  - e.g. load R3, 0xC0FF01
  - the memory address 0xC0FF01 is mapped to the I/O device's registers
  - Memory Management Unit (MMU) maps memory values and data to/from device registers
    - Device registers are assigned to a block of memory
    - When a value is written into that I/O-mapped memory, the device sees the value, loads the appropriate value and executes the appropriate command

Memory-Mapped I/O (4)



Memory-Mapped I/O (5)

- Typically, devices are mapped into lower memory
  - frame buffers for displays take the most memory, since most other devices have smaller buffers
  - Even a large display might take only 10 MB of memory, which in modern address spaces of tens-hundreds of GBs is quite modest – so memory-mapped I/O is a small penalty

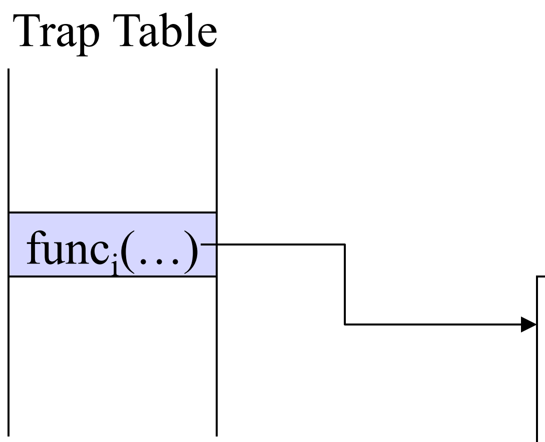
Device I/O Port Locations on PCs (partial)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Device Independent Part

- A set of system calls that an application program can use to invoke I/O operations
- A particular device will respond to only a subset of these system calls
  - A keyboard does not respond to *write( )* system call
- POSIX set: *open*( ), *close*( ), *read*( ), *write*( ), *lseek*( ) and *ioctl*( ).

Device Independent Function Call



```
dev_func_i(devID, …) {
// Processing common to all devices
   …
   switch(devID) {
   case dev0:  dev0_func_i(…);
               break;
   case dev1:  dev1_func_i(…);
               break;
   …
   case devM:  devM_func_i(…);
               break;
   };
// Processing common to all devices
   …
}
```

Adding a New Device

- Write device-specific functions for each I/O system call
- For each system call, add a new *case* clause to the *switch* statement in device independent function call
- Compile the kernel and new drivers

Problem: Need to compile the kernel, every time a new device or a new driver is added

Loadable Kernel Modules

- A loadable kernel module (LKM) is an object file that contains code to extend a running kernel
- Windows, Linux, OS X …
- Without loadable kernel modules, an OS would have to include all possible anticipated functionality already compiled directly into the base kernel
- Linux (kernel object: *.ko* extension)
  - *modprobe ( )* high level handling of LKMs (add or remove)
  - *lsmod* to list all loaded LKMs
  - *Insmod ( )* to insert and LKM
  - *rmmod ( )* to remove an LKM
  - See */lib/modules* for the all the LKMs

insmod ( ) command

- *insmod* makes an *init_module* system call to load the LKM into kernel memory
  - *sudo insmod helloworld.ko*
- *init_module* system call invokes the LKM's initialization routine (also called *init_module*) right after it loads the LKM
- The LKM author sets up the initialization routine to call a kernel function that registers the subroutines that the LKM contains

LKM example: Reconfigurable Device Drivers

- Allows system administrators to add a device driver to the OS without recompiling the OS
- The new driver is first stored as a *.ko* file
  - Contains an initialization routine
- The initialization routine calls a kernel function to register the device
  - e.g. *register_chrdev*, *register_blkdev*

- An entry table stores the actual function pointers for each device specific funcion call

  `dev_func_i[N]`

- Replace *switch* statement with

  `dev_func_i[j] (…);`

- Device registration: Fill appropriate function pointers in the entry table

Universal Serial Bus (USB)

- USB is an industry standard that defines the cables, connectors and communication protocols used in a bus for connection, communication and power supply between computers and electronic devices
  - Keyboards, mouse, printers, digital cameras, etc.
- USB has effectively replaced a variety of earlier interfaces such as serial and parallel ports as well as chargers for portable devices

Firewire (IEEE 1394)

- A serial bus interface for high speed communication and real-time data transfer
- Comparable to USB