Chapters 13 and 2: Memory-Mapped I/O, Bootloader

CSCI 3753

Instructor: Chris Womack

CSCI 3753 Announcements

- Problem set #1 will be released Monday and due the following Monday
- Teams not allowed
- Finish reading for the week

Recap…

- Exceptions: Traps, Faults, Interrupts, and Aborts
- OS Management of Input/Output (I/O) Devices
  - Device system call interface:
    - Open, close, read, write, …
  - Device Drivers
    - Interact with Device Controller’s registers: Command, Data, and status/busy bits
  - Polling I/O vs Interrupt-driven I/O
    - Hardware Interrupt flag is checked every instruction!
    - If interrupt, save state & jump to interrupt handler for device
    - Disabling/masking interrupts of different priorities
  - Direct Memory Access (DMA)

Port-Mapped vs. Memory-Mapped I/O

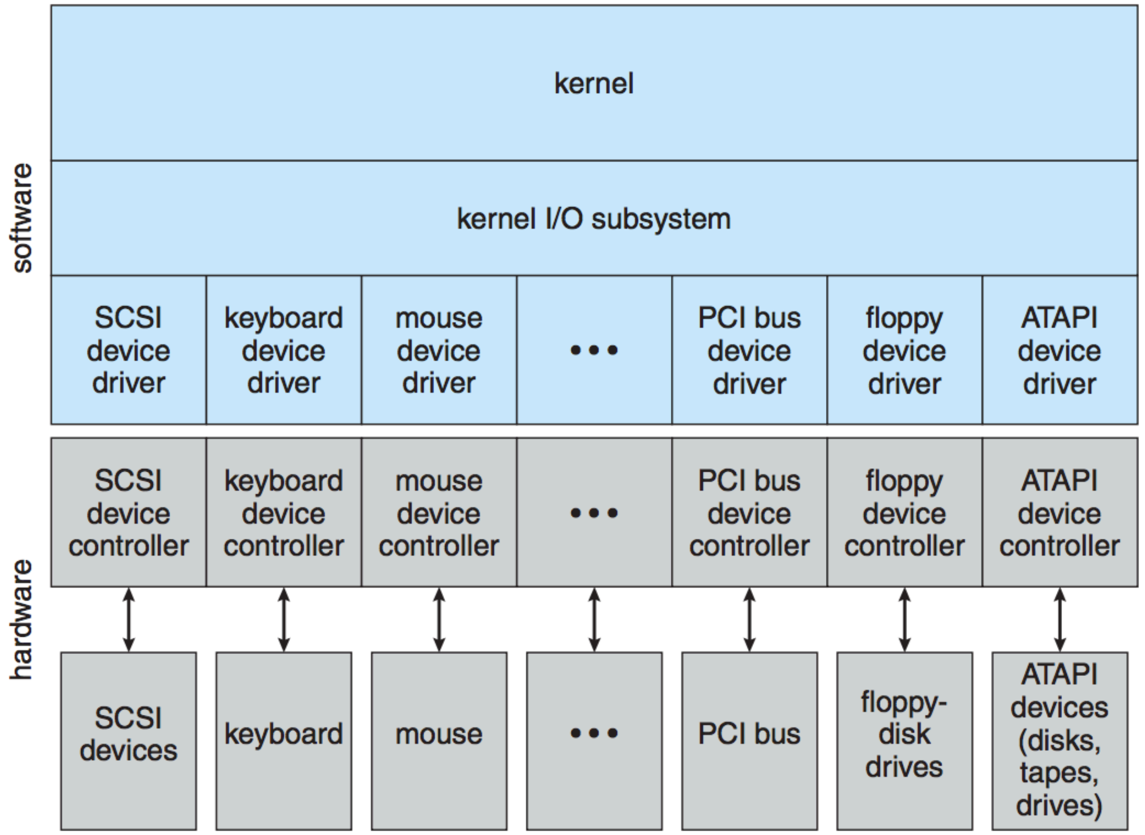
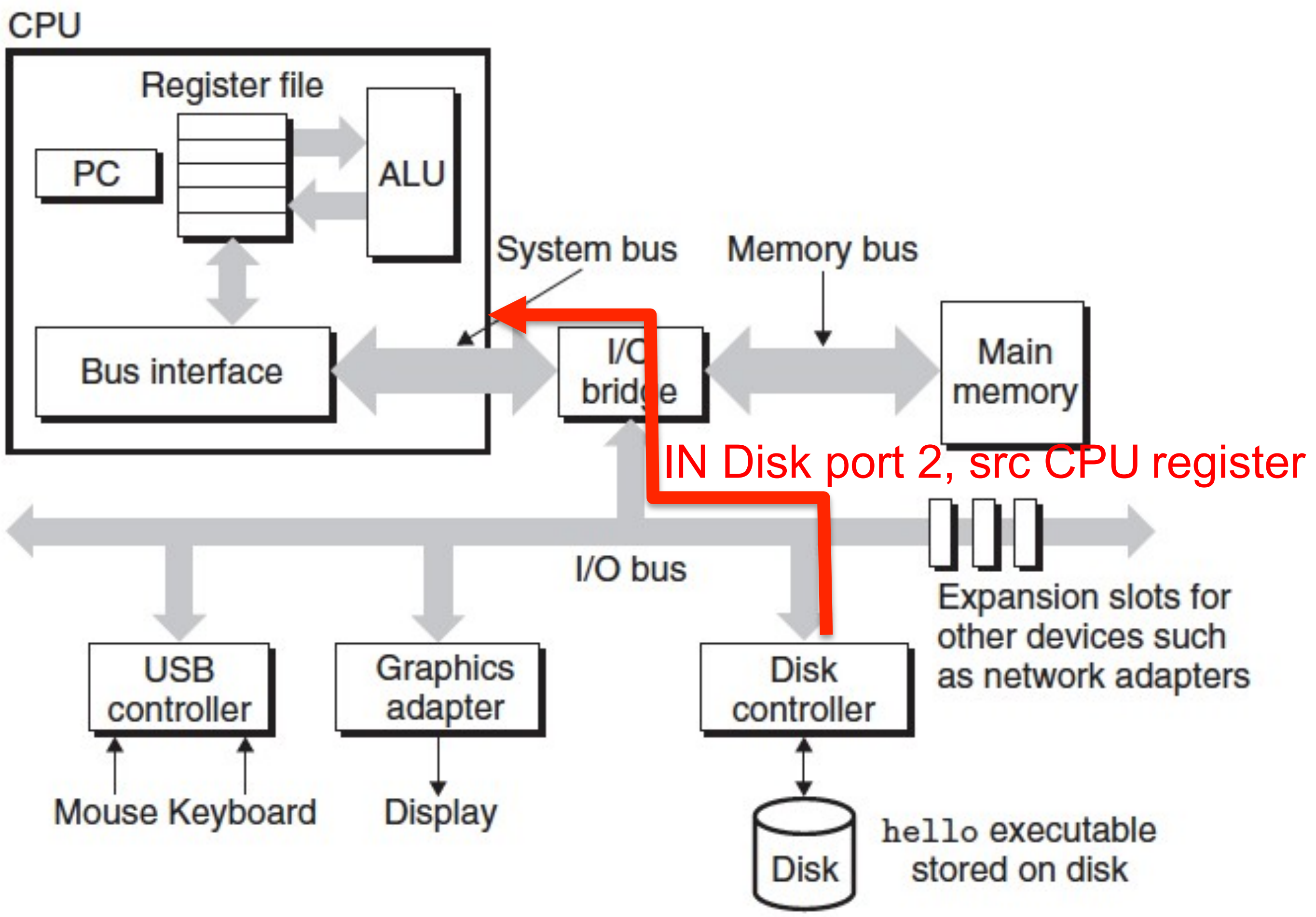


Figure 13.6 A kernel I/O structure.

Port-Mapped I/O

- One way the OS communicates with devices is through *port-mapped I/O*,
  - OS's device driver issues a special low-level assembly language instruction (IN or OUT on x86 corresponding to read or write)
    - Example: OUT dest, src
      - Writes to a device port dest from CPU register src
    - Example: IN dest, src
      - Reads from a device port src to CPU register src
    - Later Intel introduced INS, OUTS (for strings), and INSB/INSW/INSD (different word widths), etc.
  - This sets I/O bus lines (control, address, data) to read/write data from/to a device controller's registers (i.e. device "port")

Port-Mapped I/O



Port-Mapped I/O Implications

- I/O address space is different/separate from main memory's address space
- Need special hardware instructions to support port-mapped I/O
- Only OS (device driver) in kernel mode can execute these instructions

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)

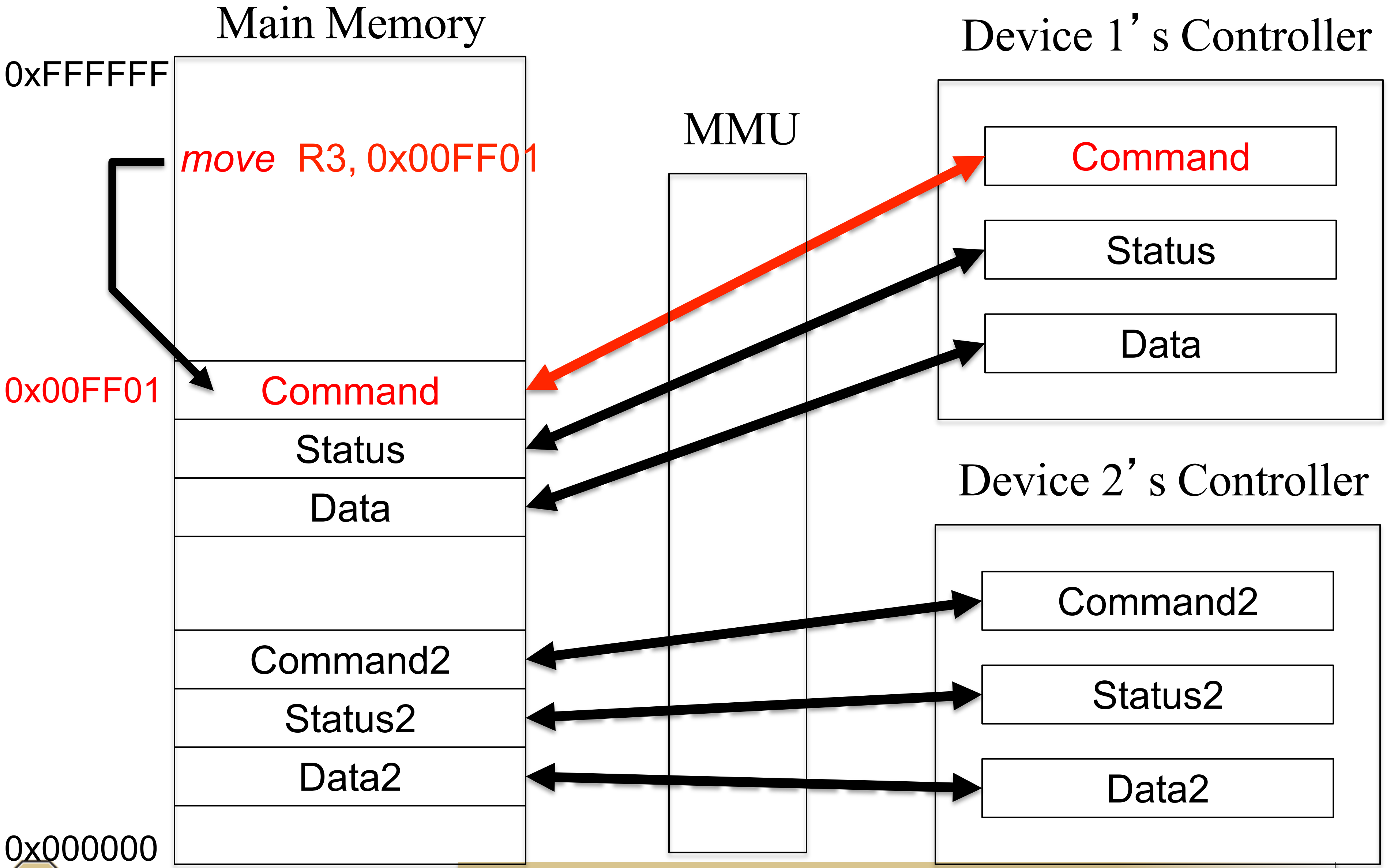
Port-Mapped I/O Limitations

- 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
- Difficult for application developers
  - Application has to save data to be written, then call write()
- Typically transfer very small amounts of data

Memory-Mapped I/O

- Map a device’s registers and memory to portions of main memory
  - Reserve a block of memory M1 to map to the device D1’s registers and memory, memory block M2 maps to D2, …
- To read from/write to a device, just reference the memory location, and the OS+hardware will fulfill the request automatically
  - For example, writing to a mapped memory location will automatically write the data into the I/O device
  - e.g. `move` R3, 0x00FF01
  - the memory address 0x00FF01 is mapped to the I/O device’s registers

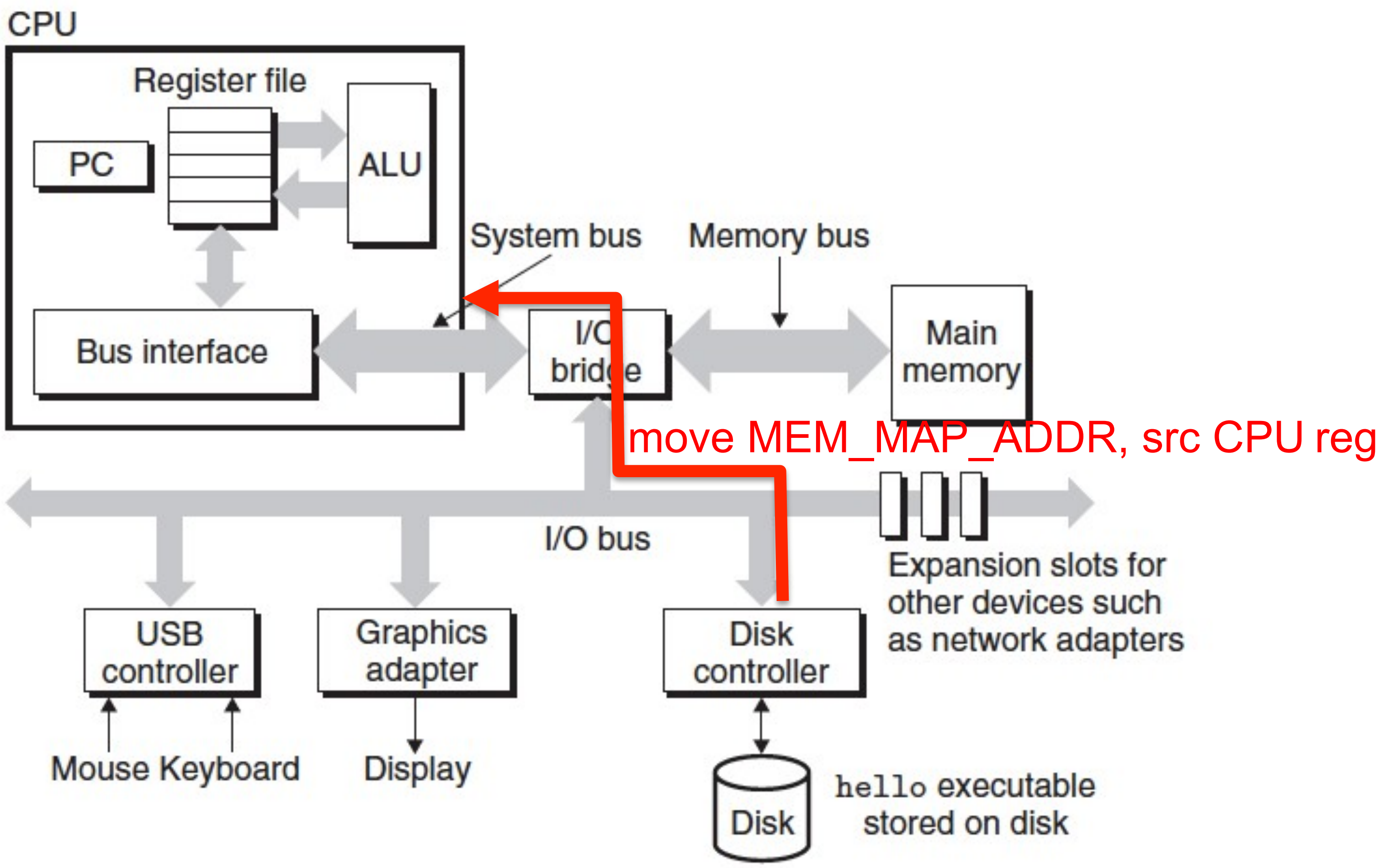
Memory-Mapped I/O



Memory-Mapped I/O (2)

- 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
  - i.e. I/O devices are listening on the shared address lines, and when they see their memory-mapped address, and say a write, they will intercept that and write to their device's memory or registers

Memory-Mapped I/O



Memory-Mapped I/O (3)

- How does a device learn its memory-mapped address?
  - OS will tell the device its assigned memory-mapped address range by writing into a special set of device registers called the base address registers
  - These addresses will be non-conflicting with other addresses mapped to other devices

Memory-Mapped I/O (4)

- Typically, devices are mapped into lower memory in x86
  - frame buffers for displays take the most memory, since most other devices have smaller buffers
  - Even a large display might take only tens of MB of memory
  - memory-mapped I/O is a small penalty, since tens of MB of space is small vs many GBs of memory available in RAM

Memory-Mapped I/O Advantages

- Example: rendering data into the graphics card's frame buffer. Old approach:
  1. A program firsts renders image to be displayed into main memory
  2. Then the program calls high-level API to write image into frame buffer
  3. This invokes OS to write each byte to device (read a byte into CPU register and call OUT for each byte to the device).
- Advantage: Faster data transfer
  – With memory-mapped I/O, step 3 is eliminated because writes to addresses in step 1 are memory-mapped and are immediately written into the frame buffer at the correct location

Memory-Mapped I/O Advantages (2)

- Programming simplified
  - Step #2 is eliminated (previous slide), because writes to memory-mapped addresses corresponding to the frame buffer are immediately written into the frame buffer
- No extra hardware instructions are needed
  - Normal `move` instructions suffice
- I/O and memory share the same address space, i.e. same control, data and address buses
- For these reasons, memory-mapped I/O is more popular than port-mapped I/O

Summarize MM I/O VS PI/O

- Port Mapped I/O
  - Very hardware dependent – needs a lot of hardware support.
  - Can be faster due to own memory space
  - No operations on data in device port (IN/OUT)
- Memory Mapped I/O
  - Reduces hardware complexity
  - Arithmetic instructions for main mem valid on device
  - Can cause slow down with a lot of data due to shared memory space

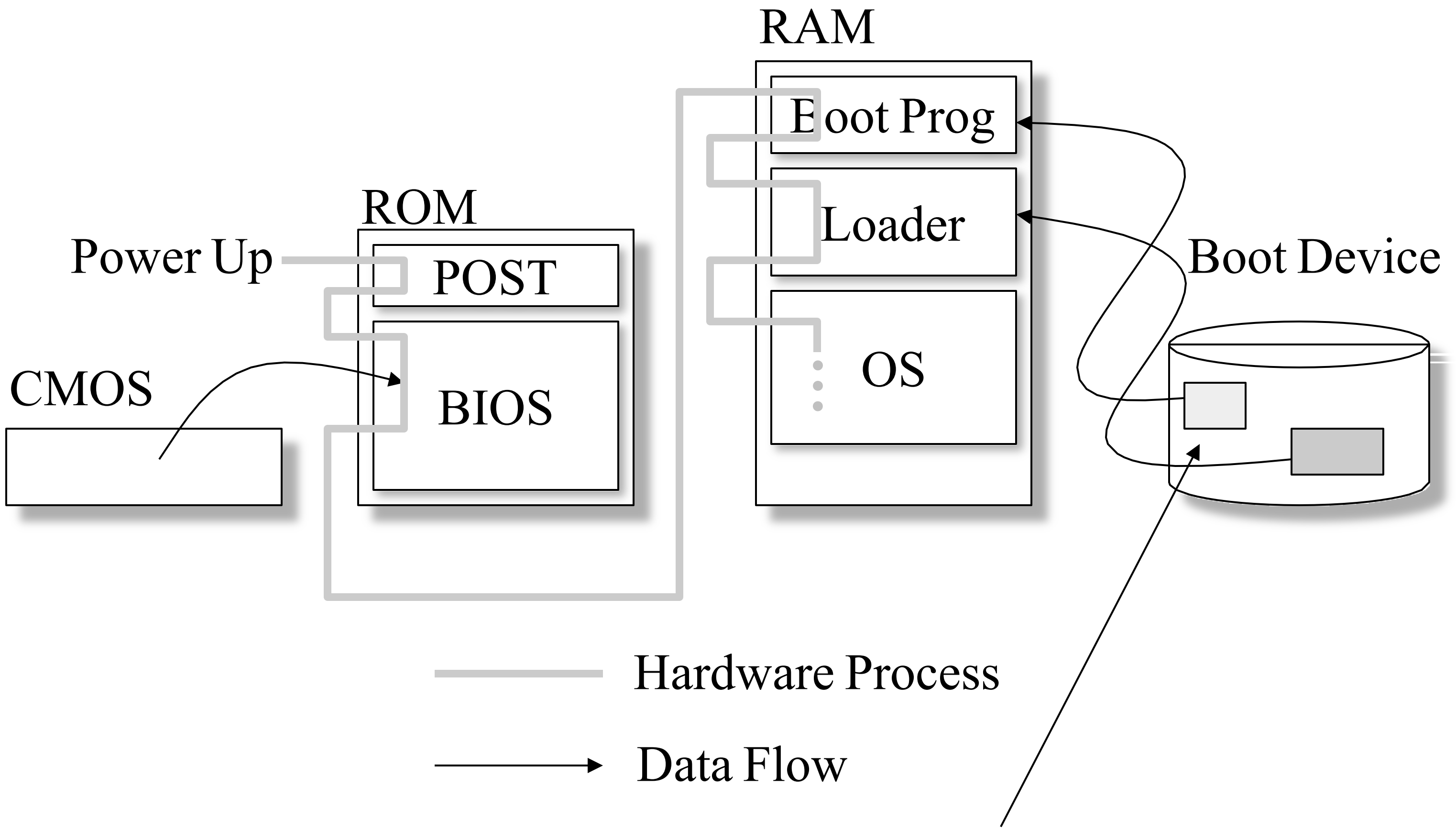
Bootstrapping the OS in PCs

- 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
  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 in PCs

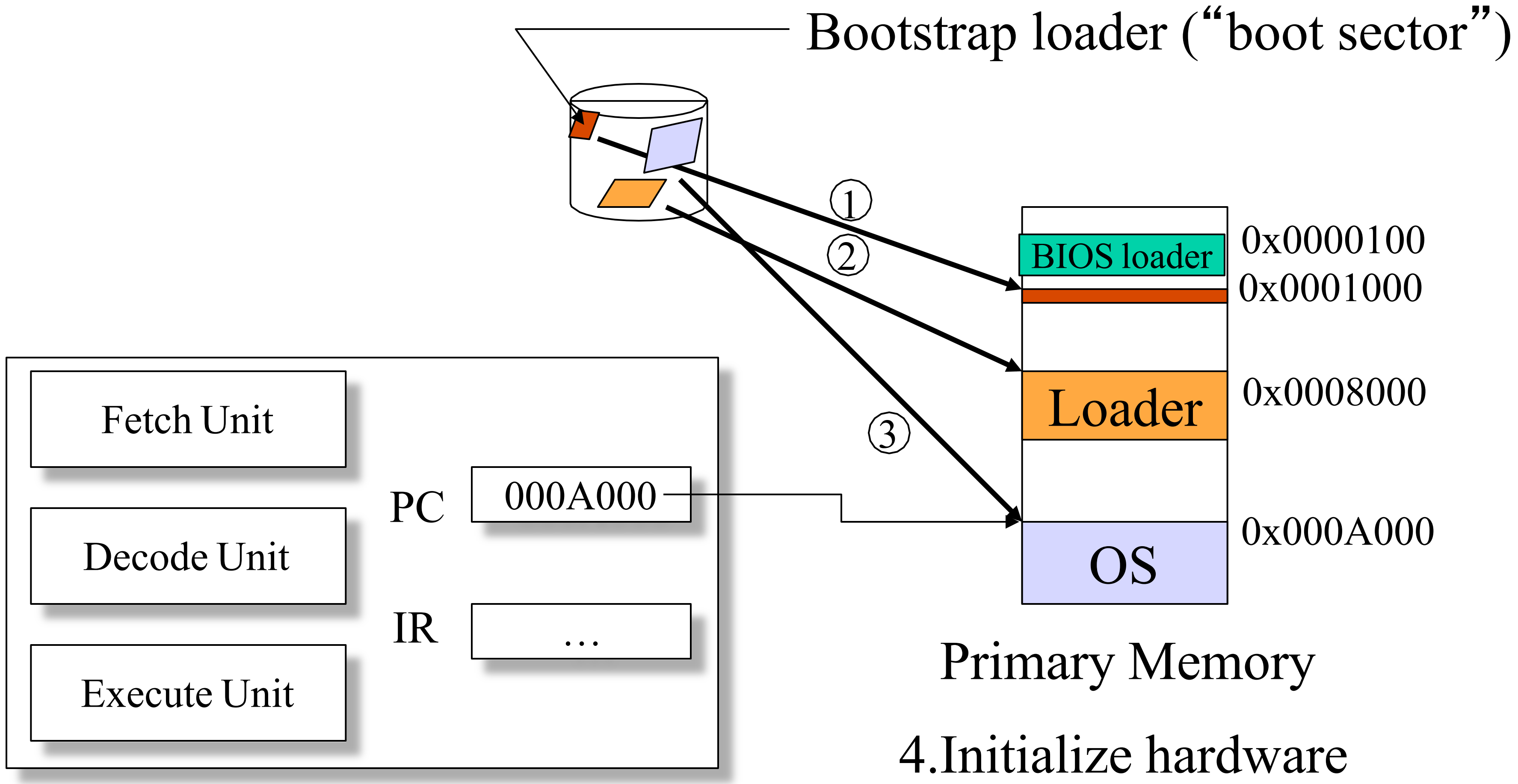
- Multi-stage procedure: (continued)
  4. Primitive loader then loads the secondary stage bootloader
     - Examples of this bootloader include LILO (Linux Loader), and GRUB (Grand Unified Bootloader)
     - Can select among multiple OS's (on different partitions) – i.e. dual booting
     - Once OS is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



Primary Memory

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

Adding Device Drivers

- Unsatisfactory approach: device drivers are statically linked into the OS kernel
  - Requires kernel to be recompiled each time a new device was added
- Want OS to support new devices (which have their own device drivers) without recompiling the kernel each time
- Solution: kernel modules
  - Essentially these are dynamically linked software objects
  - Most modern OSs support this way to add functionality to the kernel