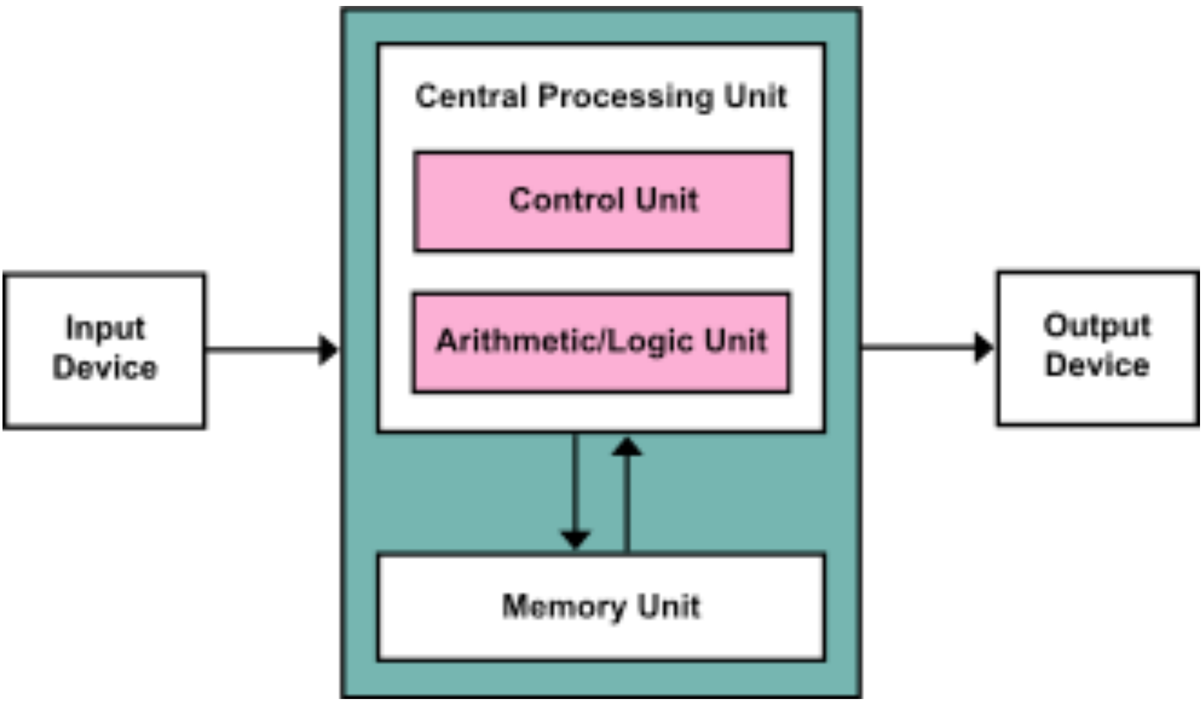
CSCI 3753
Operating Systems

Device Management

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 09/07/2017

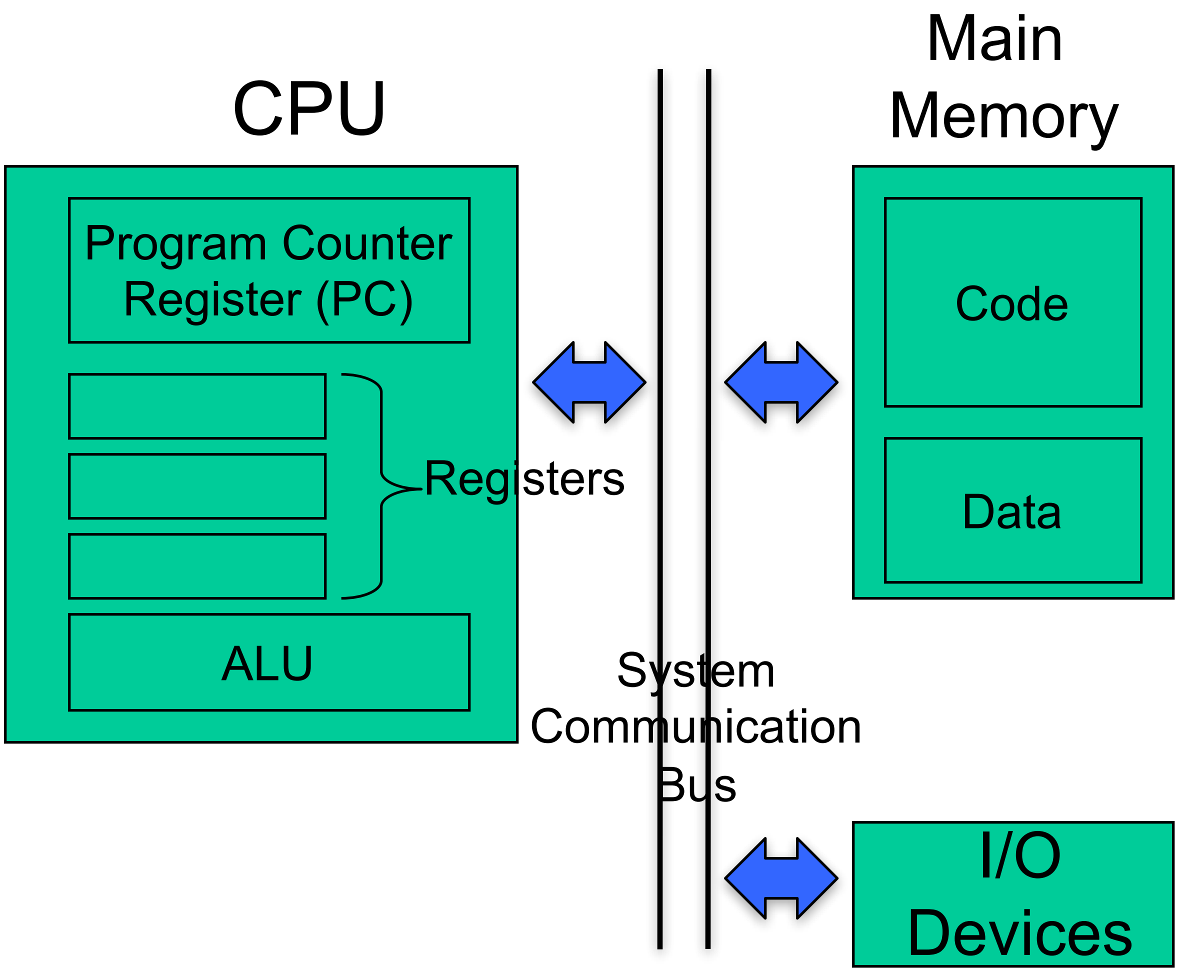
Von Neumann Computer Architecture



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

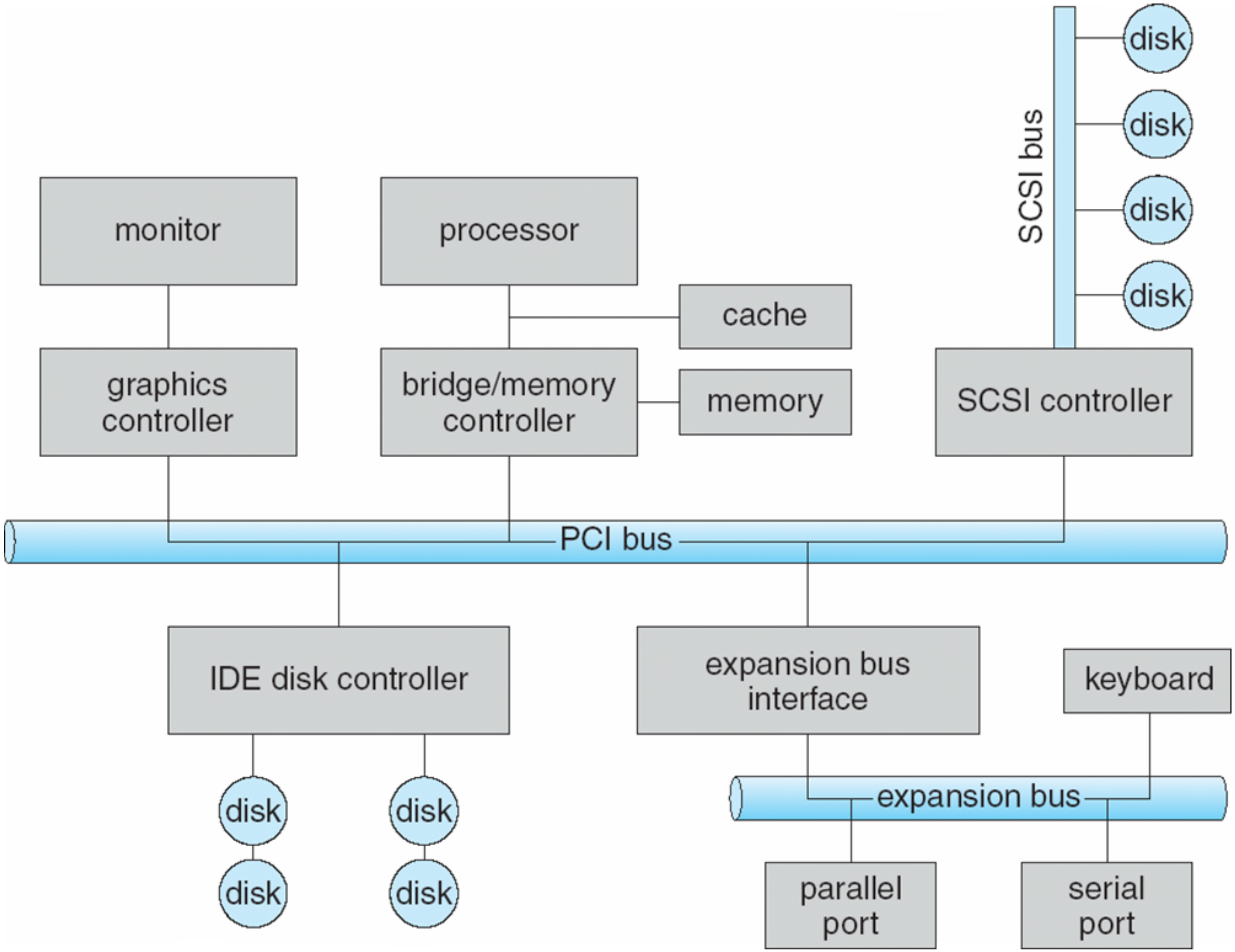


Want to support more devices: card reader, magnetic tape reader, printer, display, disk storage, etc.

System bus evolved to handle multiple I/O devices.

Includes control, address and data buses

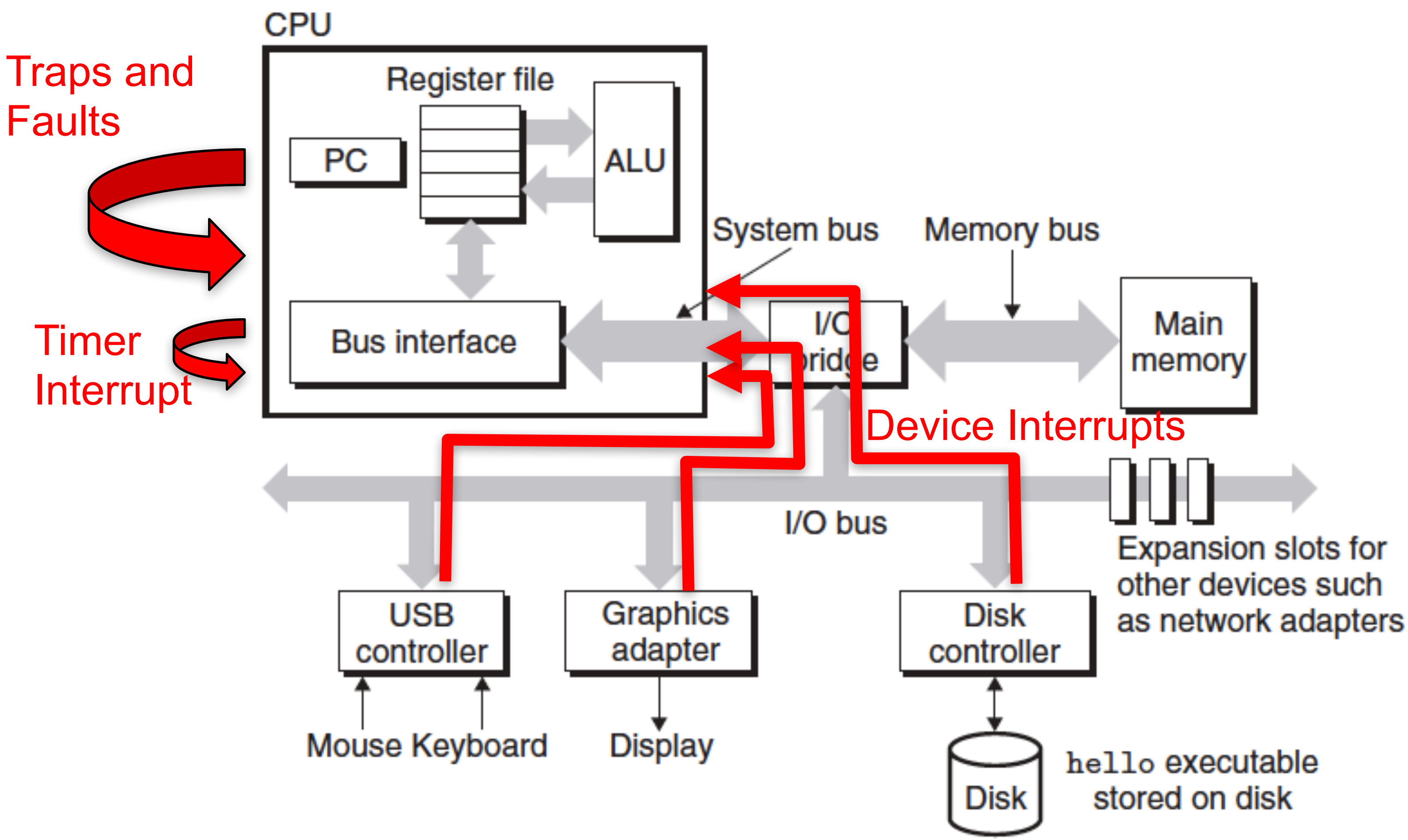
A Typical PC Bus Structure



Recap …

- Three design issues
  1. System Boot: 4 stages
     - Power On Self Test (POST)
     - BIOS
     - Master Boot Record (MBR) – primitive loader
     - Secondary stage boot loader
  2. Protecting OS from applications
     - Processor mode bit – supervisor mode and user mode
     - trap instruction
  3. System call API
     - Trap table
- Reading: Chapters 1, 2 and 13

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

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

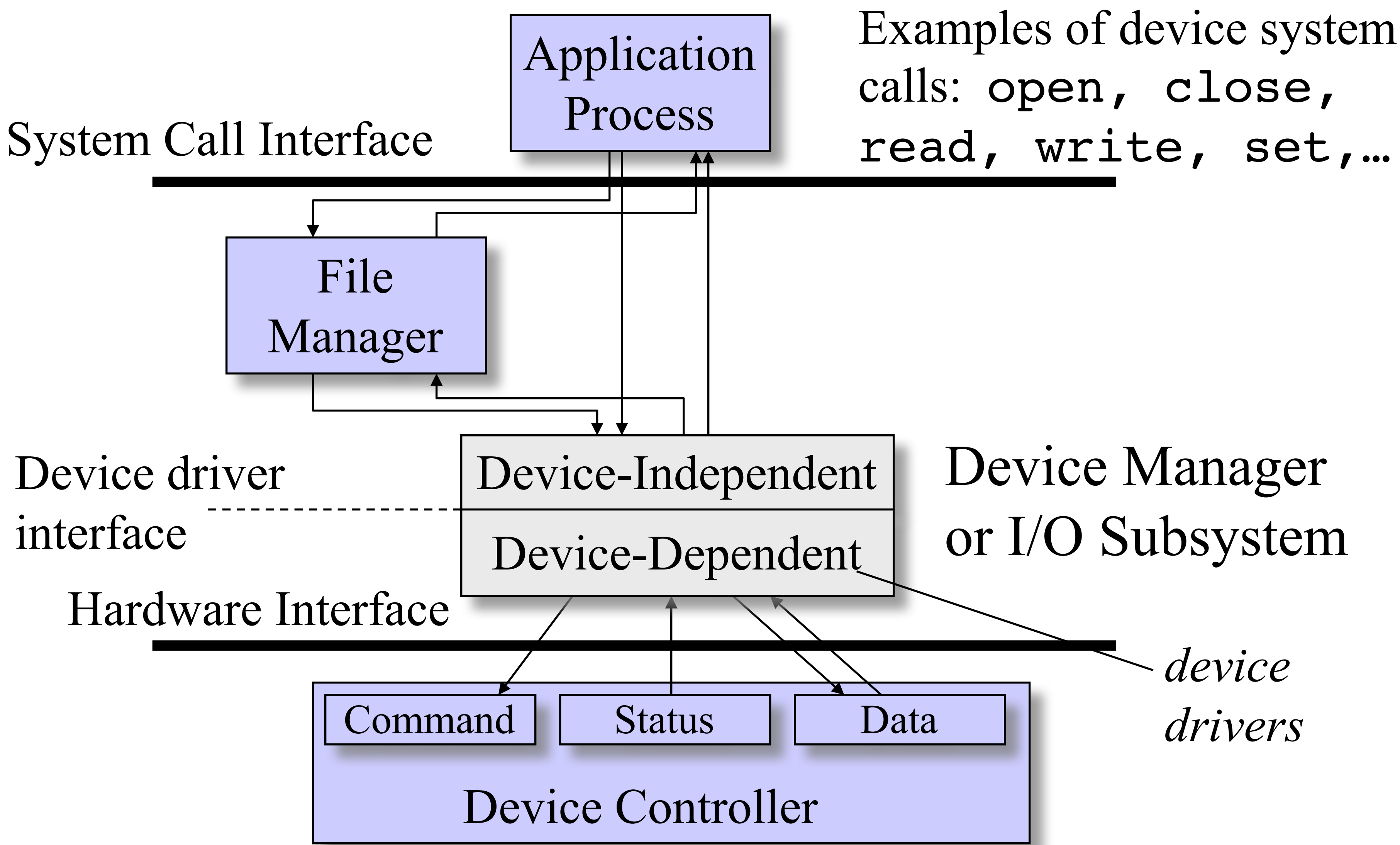
OS assigns

offsets form *interrupt vector*

Device 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.

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
  - 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

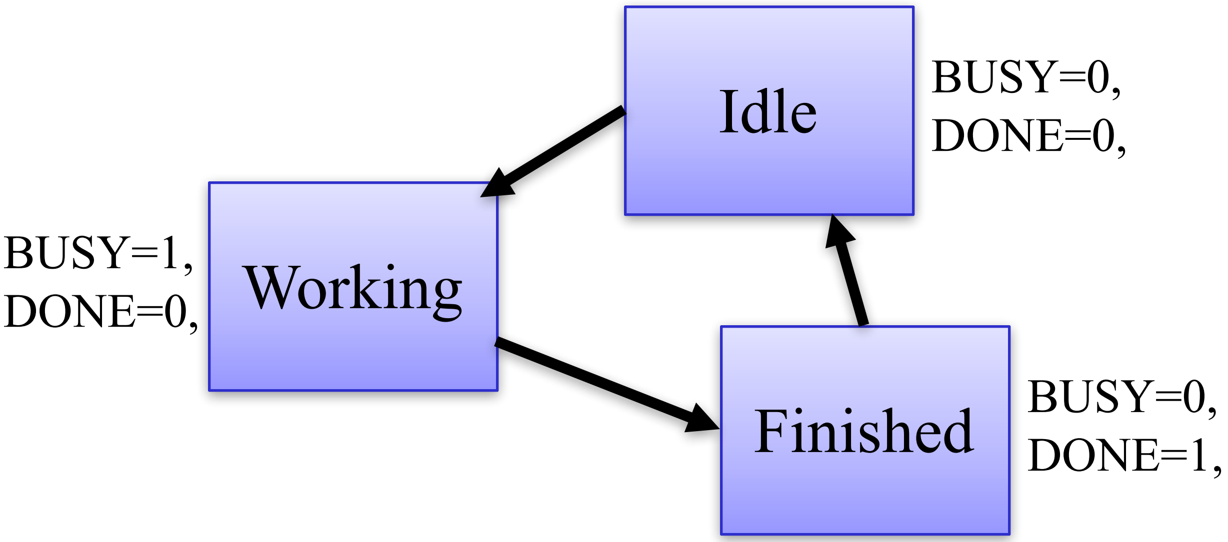
Device Characteristics

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

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

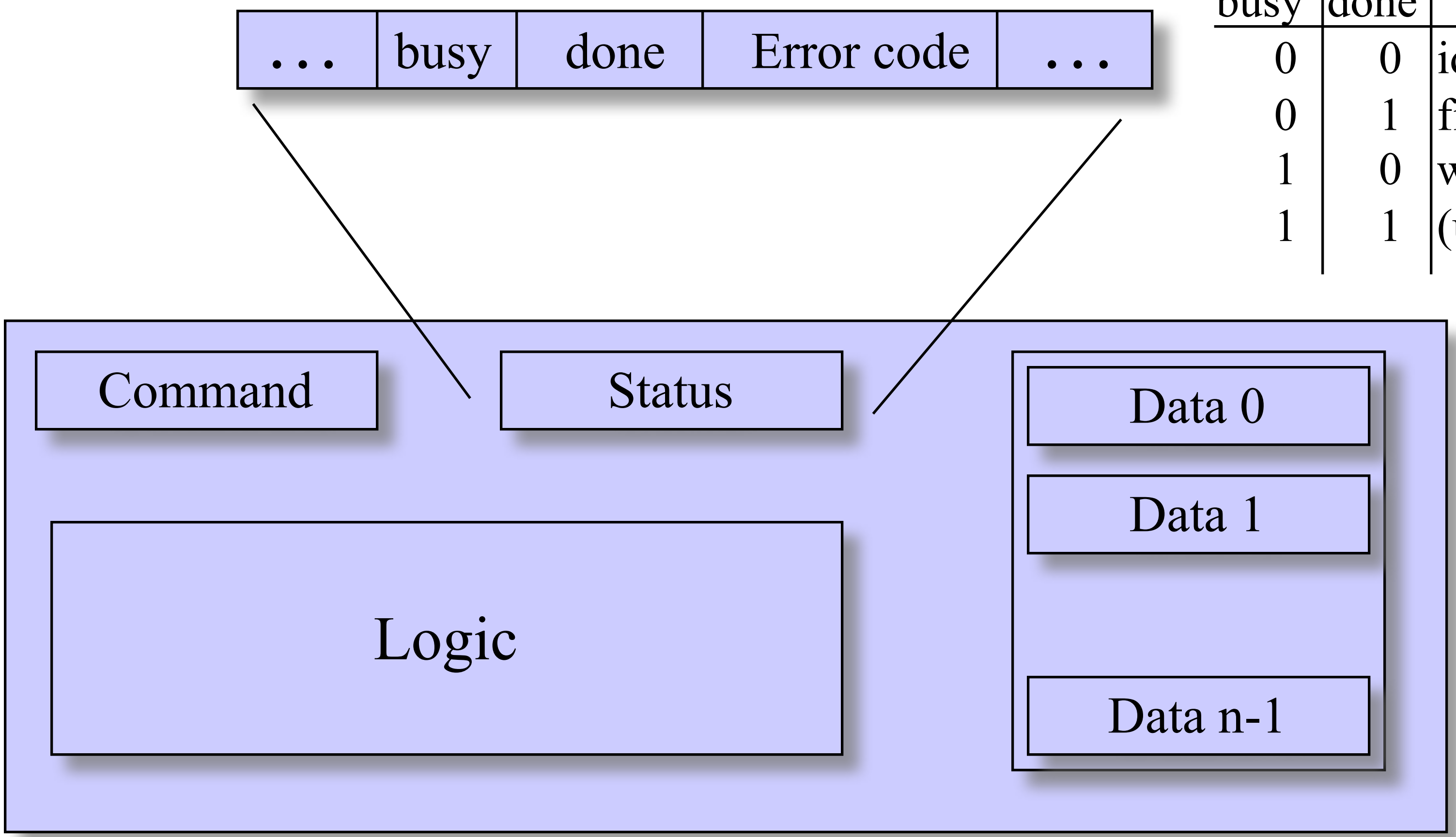
- Therefore, need 2 bits for 3 states:
  - A BUSY flag and a DONE flag
  - BUSY=0, DONE=0 => Idle
  - BUSY=1, DONE=0 => Working
  - BUSY=0, DONE=1 => Finished
  - BUSY=1, DONE=1 => Undefined

Polling I/O: A Write Example

```
while(deviceN.busy || deviceN.done) <waiting>;
deviceN.data[0] = <value to write>
deviceN.command = WRITE;
while(deviceN.busy) <waiting>;
/* finished, so read some status bits… */
deviceN.done = FALSE;
```

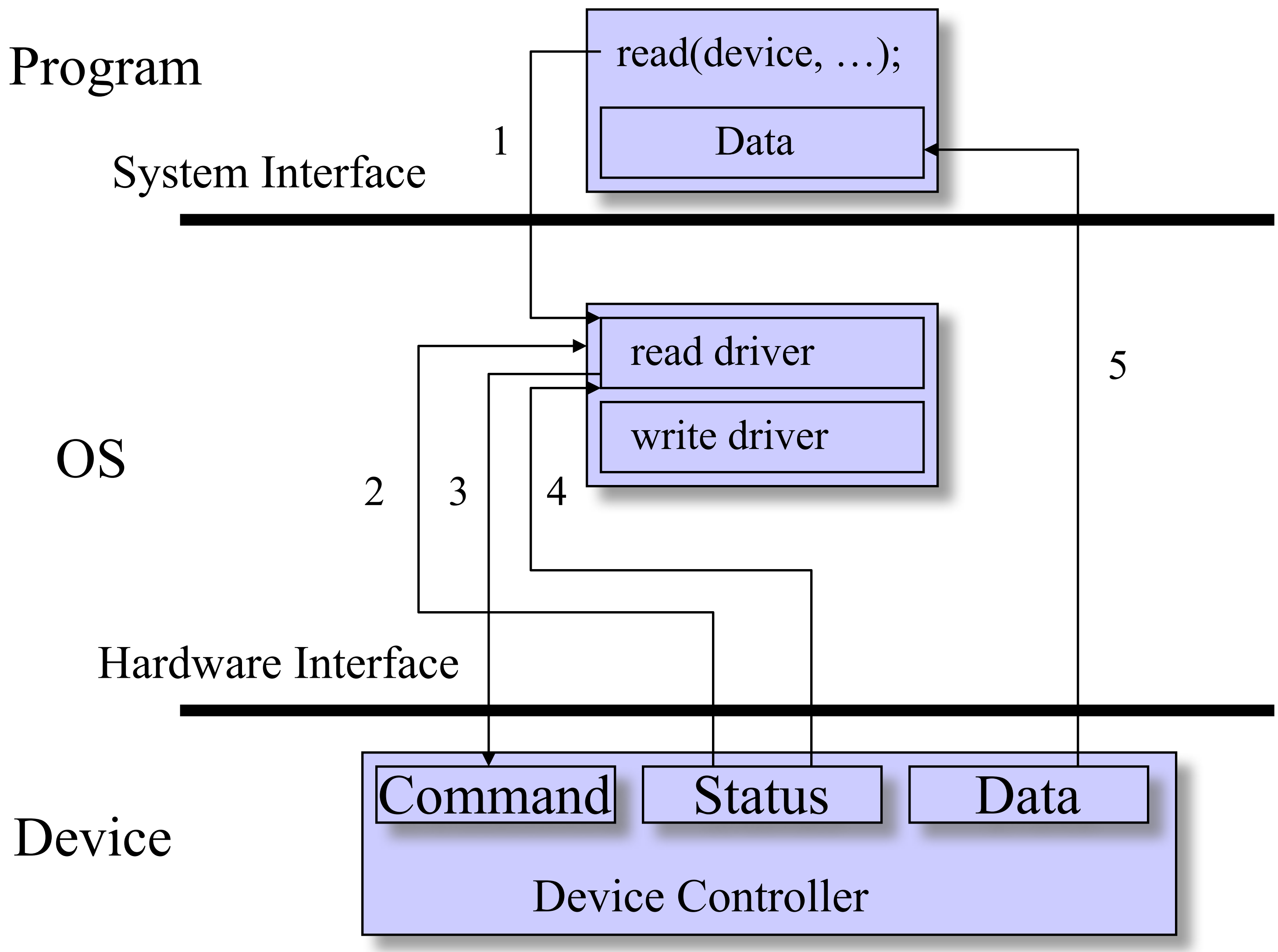
BUSY	DONE
*	*
0	0
1	0
0	1
0	0

Device Controller Interface



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

Polling I/O Read Operation



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

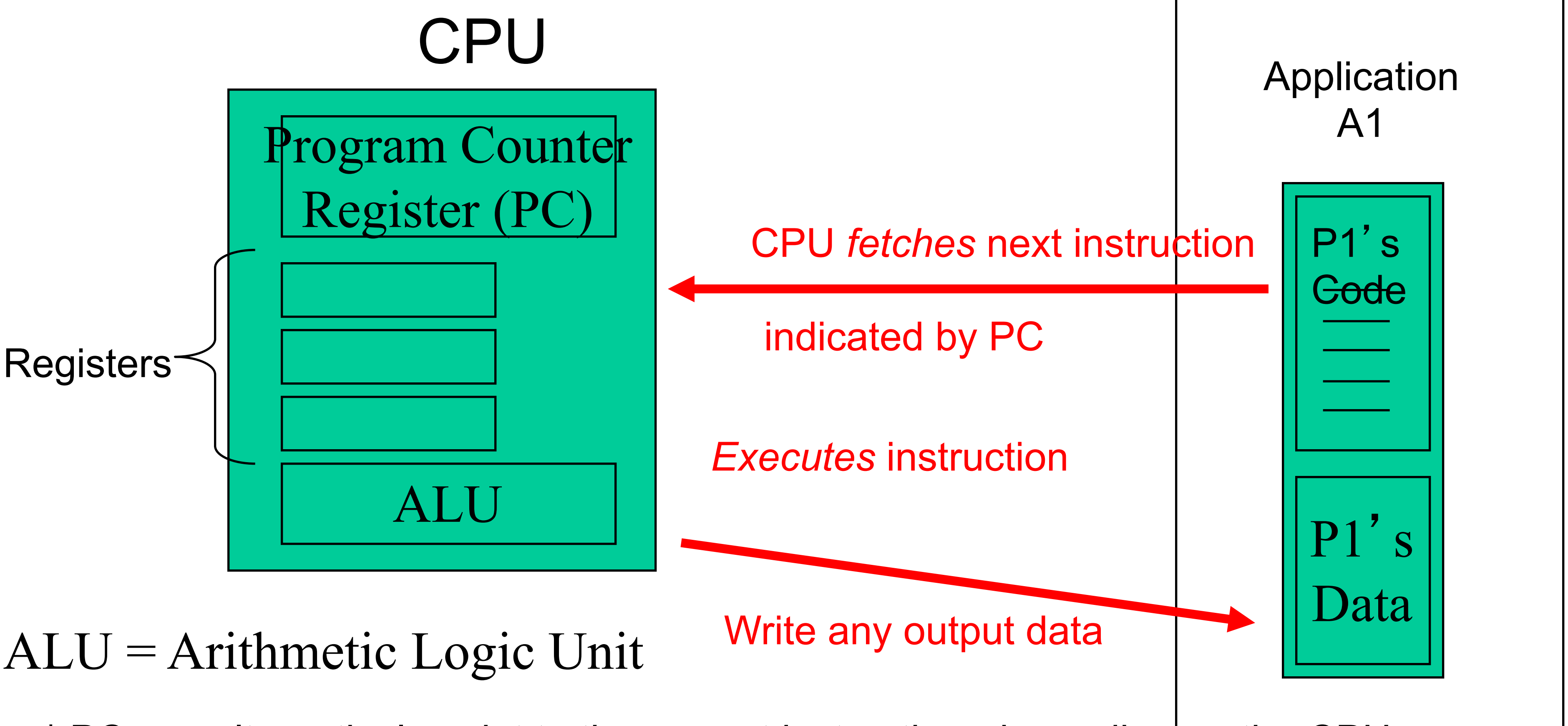
- Underneath the blocking/non-blocking synchronous/asynchronous system call API, OS can implement several strategies for I/O with devices
  - direct I/O with polling
    - the OS device manager busy-waits, we've already seen this
  - direct I/O with *interrupts*
    - More efficient than busy waiting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

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*



* 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

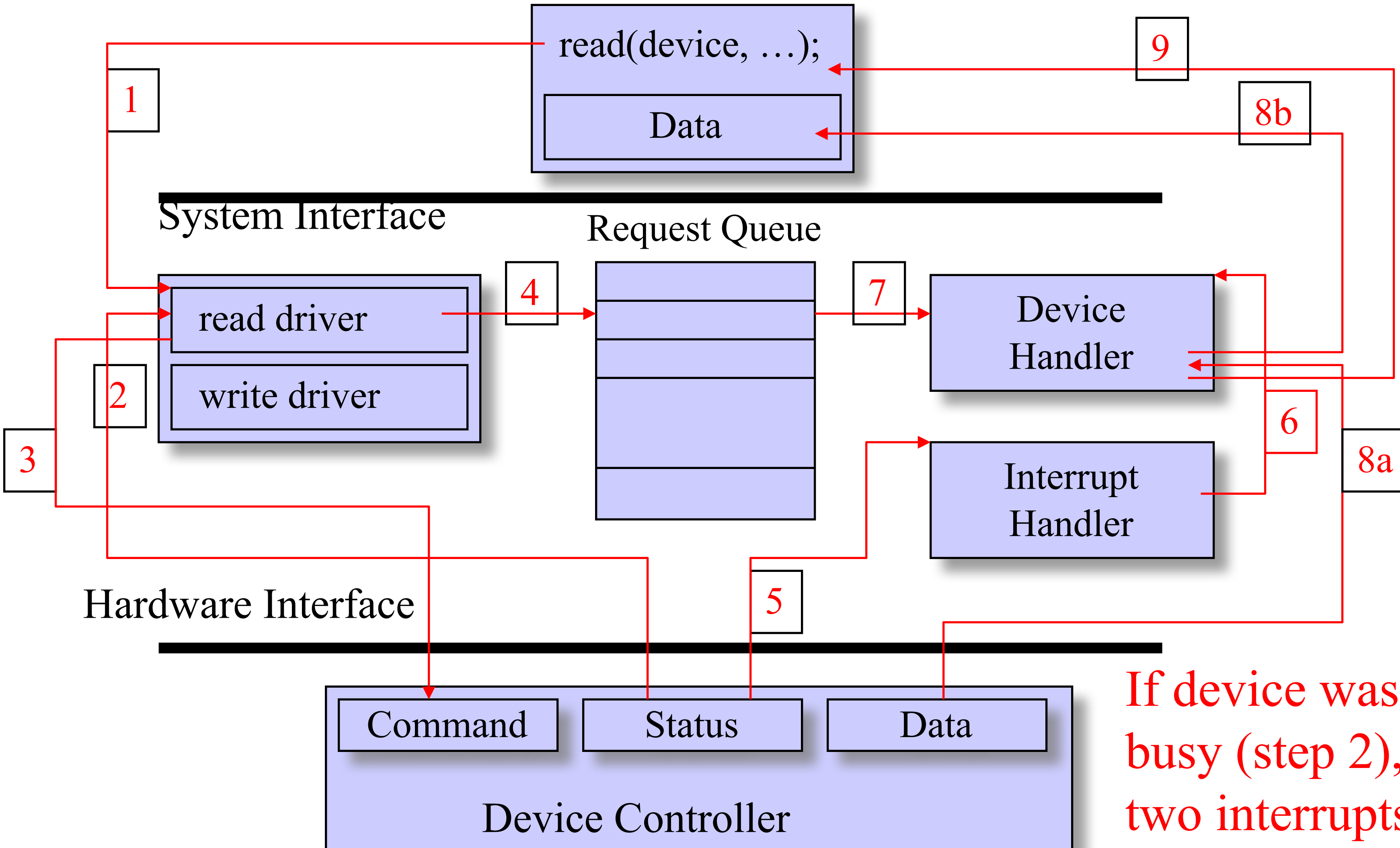
Interrupt Handler

- 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

Recap …

- Device manager: Controls operations of I/O devices
- I/O devices consist of two high-level components
  - Mechanical component and device controllers
- Device controller states: Idle, Working, Busy
- Three I/O strategies
  - Direct I/O with polling
    - CPU first waits for device to become idle
    - CPU issue I/O command
    - CPU waits for device to complete
  - Direct I/O with interrupts
    - No busy waiting
  - DMA with interrupts

Interrupt-Driven I/O Operation



If device was busy (step 2), two interrupts occur

Problem with Interrupt driven I/O

- Data transfer from disk can become a bottleneck if there is a lot of I/O copying data back and forth between memory and devices
  - Example: read a 1 MB file from disk into memory

    The disk is only capable of delivering 1 KB blocks

    So every time a 1 KB block is ready to be copied, an interrupt is raised, interrupting the CPU

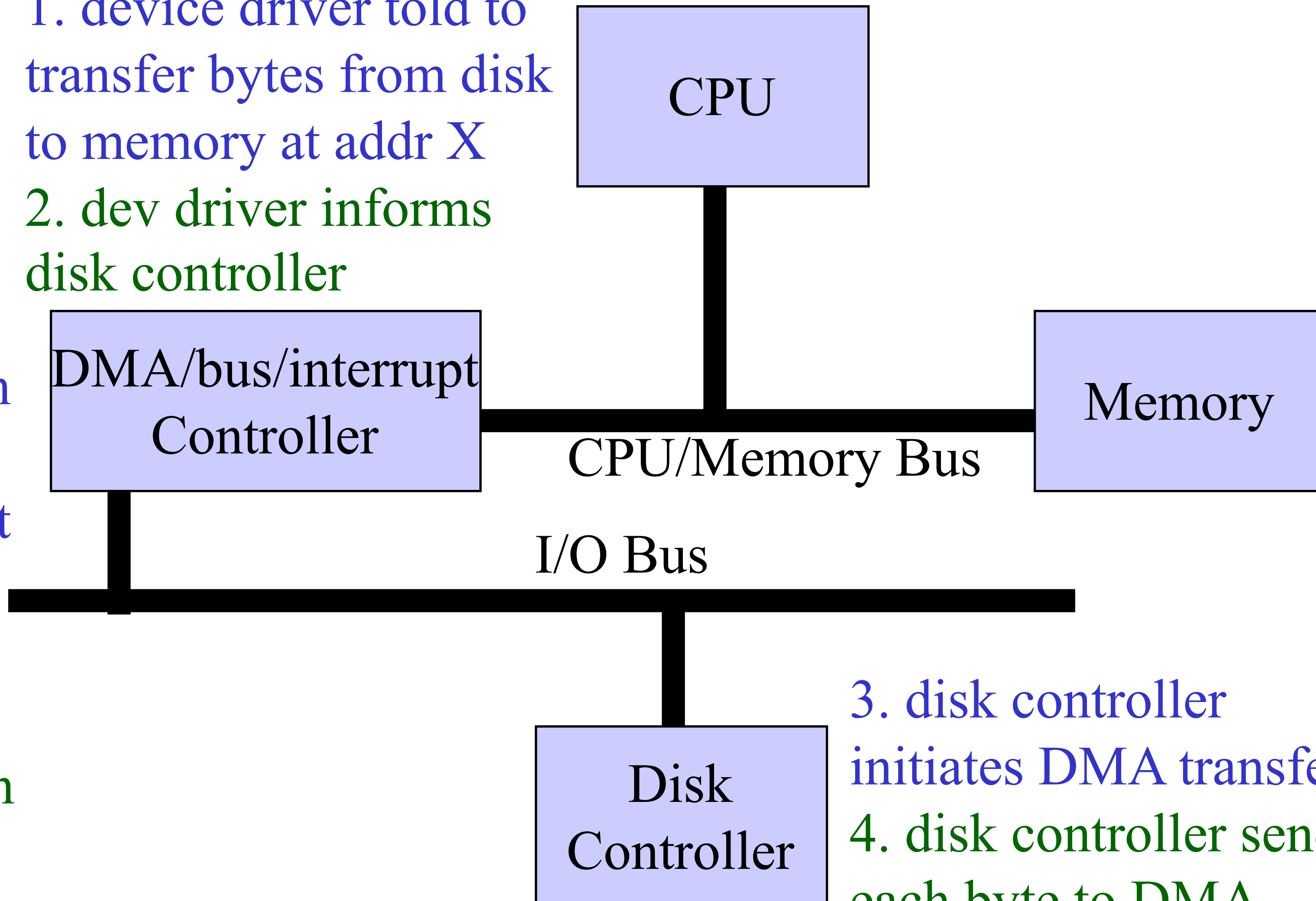
    This slows down execution of normal programs and the OS
  - Worst case: CPU could be interrupted after the transfer of every byte/character, or every packet from the network card

Direct Memory Access (DMA)

- 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
- Modern systems offload some of this work to a special-purpose processor, Direct-Memory-Access (DMA) controller
- The DMA controller operates the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU

DMA with Interrupts Example

1. device driver told to transfer bytes from disk to memory at addr X
2. dev driver informs disk controller
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA sends each byte to memory at addr X
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.

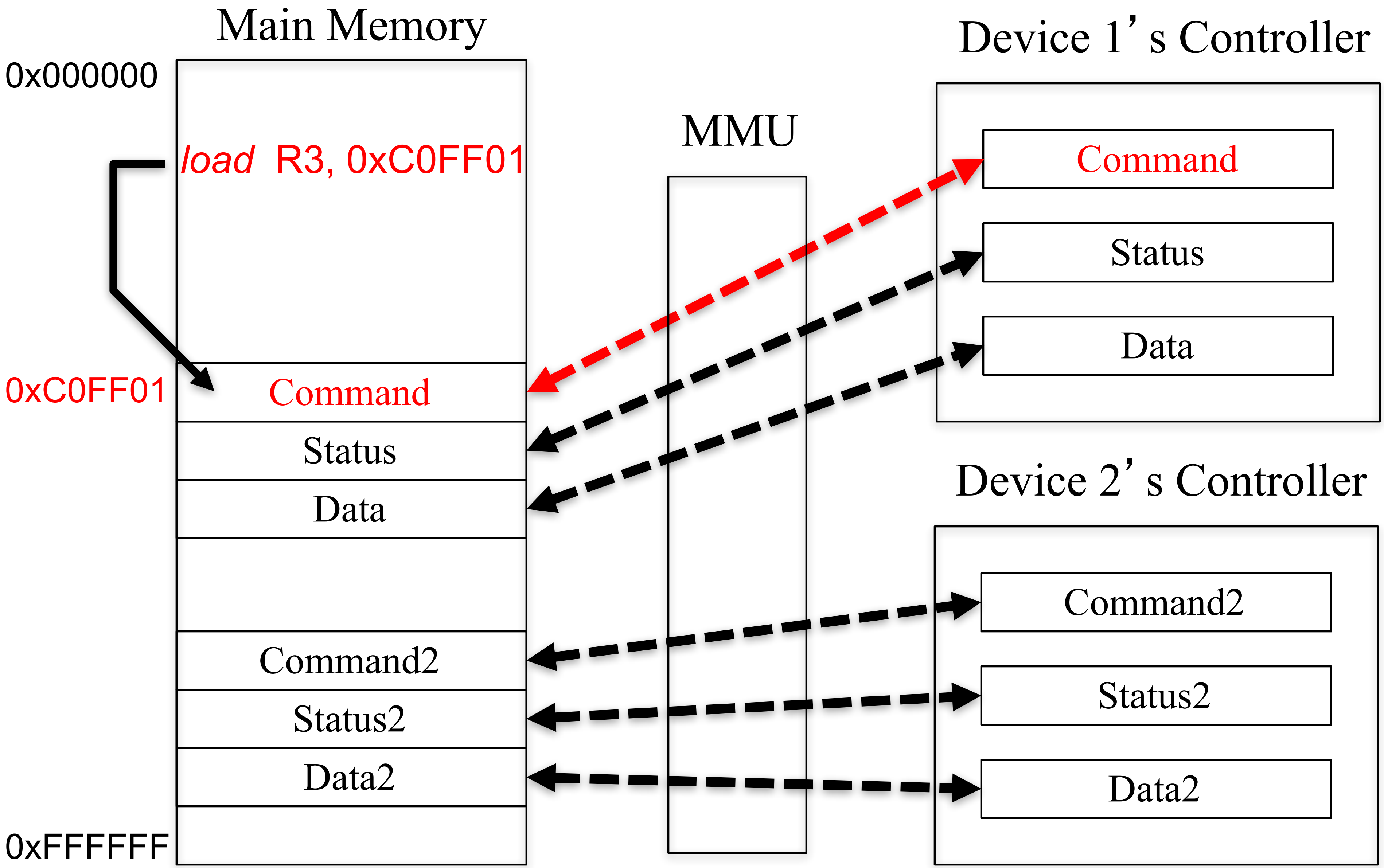
Memory-Mapped I/O (2)

- 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.
    - AMD did not extend the port I/O instructions when defining the x86-64

Memory-Mapped I/O (3)

- Memory-mapped I/O: device registers and device memory are mapped to the system address space
- 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 an I/O device’s register
- 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)

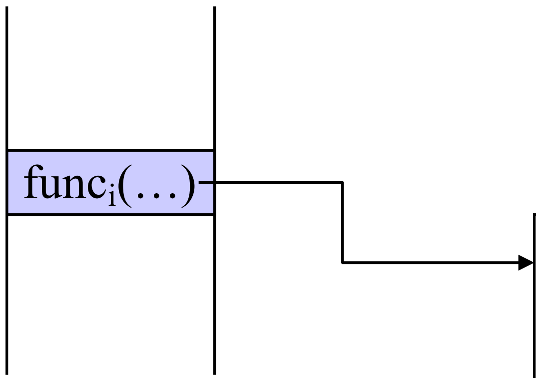
I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

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

Trap Table

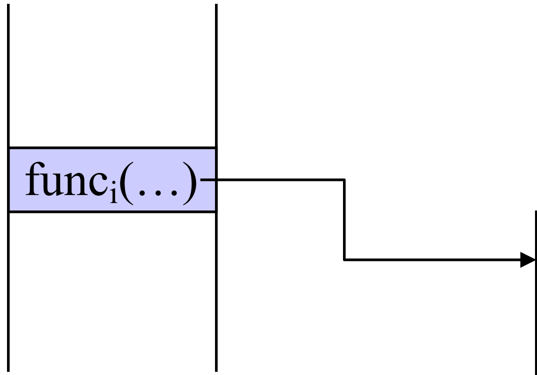



```
dev_func_i(devID, …) {
// Processing common to all devices
   …
   switch(devID) {
   case dev0:  dev0_func_i(…);
               break;
   case dev1:  dev1_func_i(…);
               break;
   …
   case devM:  devM_func_i(…);
               break;
   };
// Processing common to all devices
   …
}
```

Adding a New Device

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

Trap Table




```
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;
   case devNew: devNew_func_i(…);
               break;
   };
// Processing common to all devices
   …
}
```

Adding a New Device

- After updating all dev_func_*(…) in the kernel, compile the kernel

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