CSCI 3753 Operating Systems
Summer 2020

Christopher Godley

PhD Student

Department of Computer Science

University of Colorado Boulder

Lecture 4
Device Strategies

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

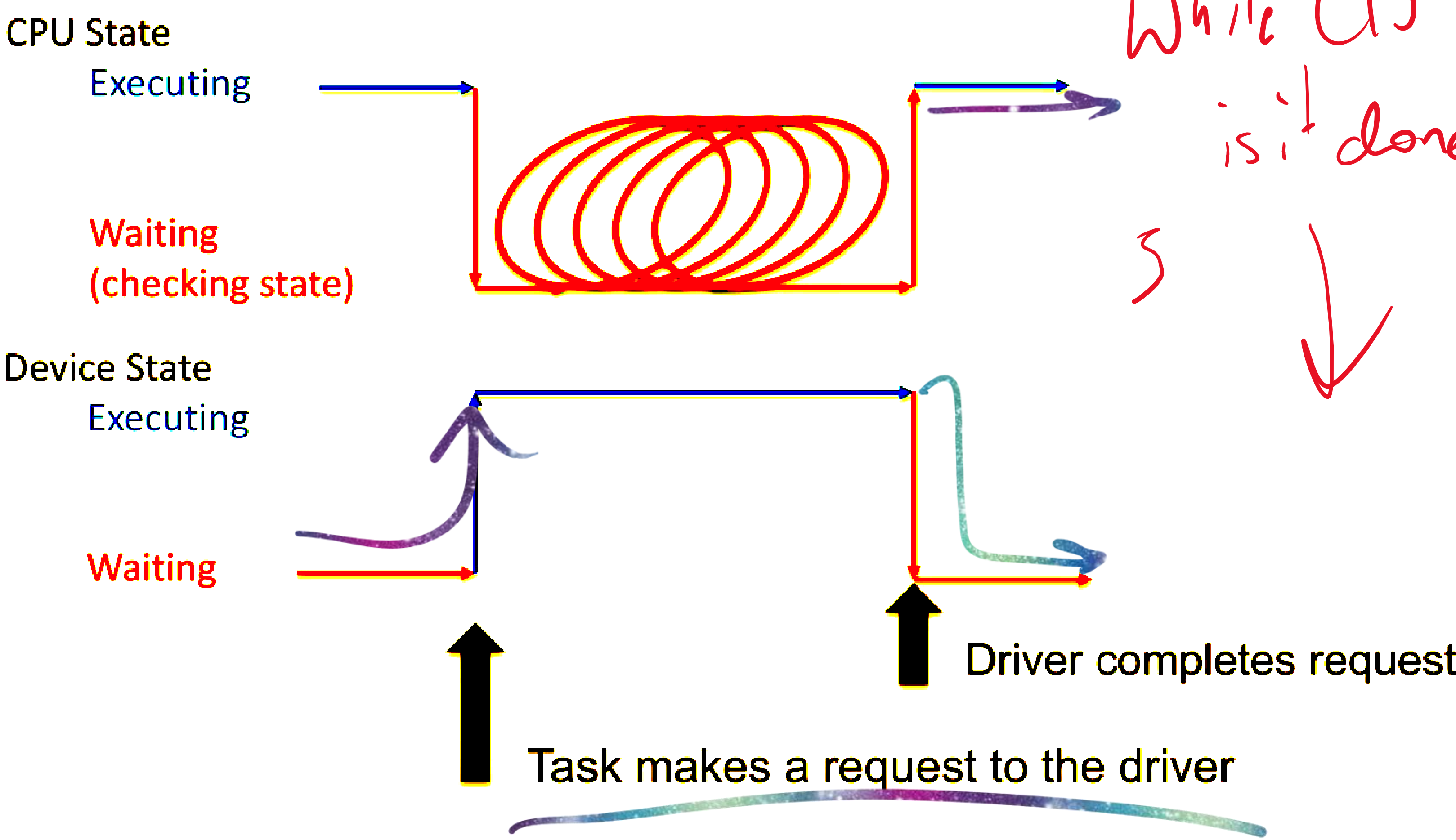
Device Manager I/O Strategies

- 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

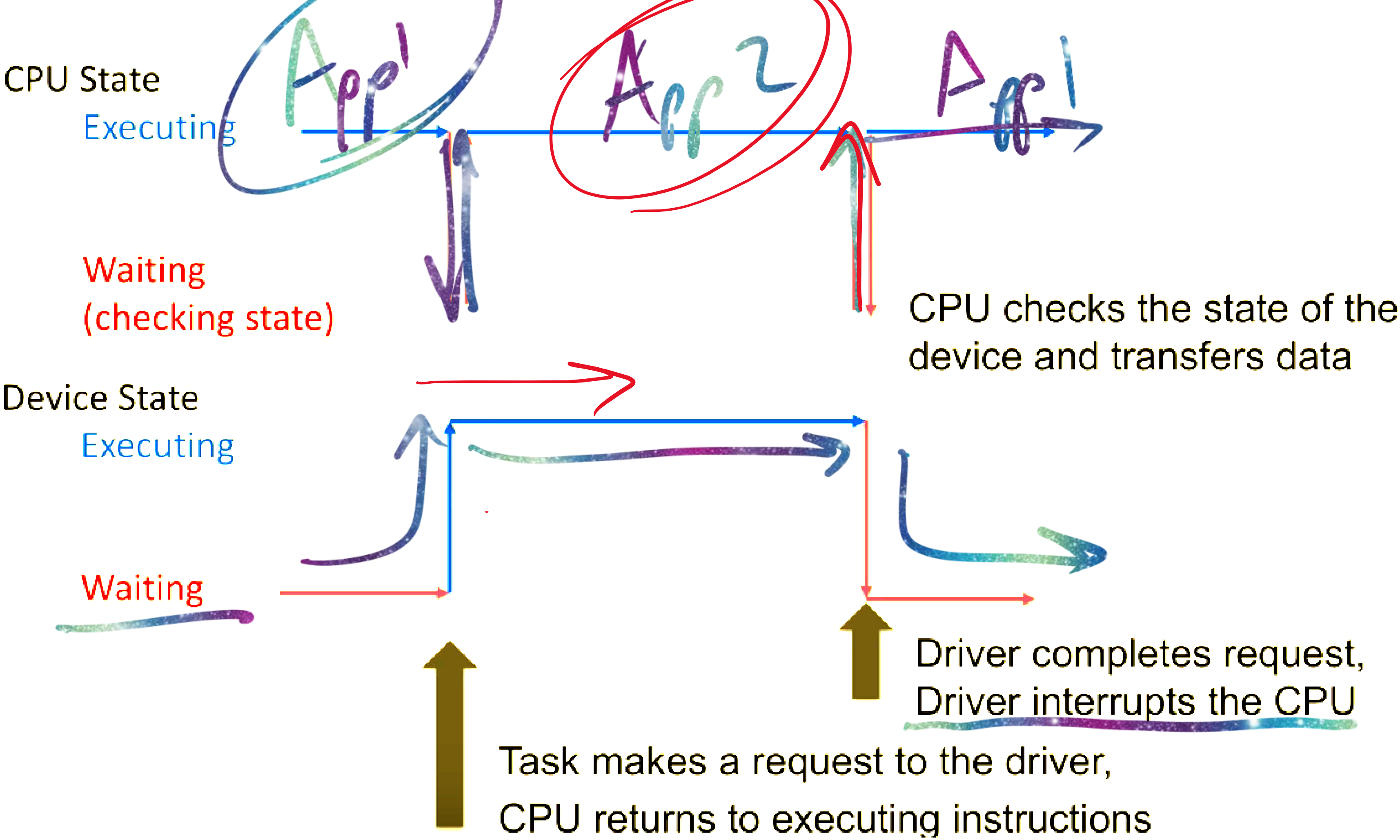
Direct I/O with Polling

while(is it ready?)



while (1) {
is it done?
}

Direct I/O with Interrupt



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

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

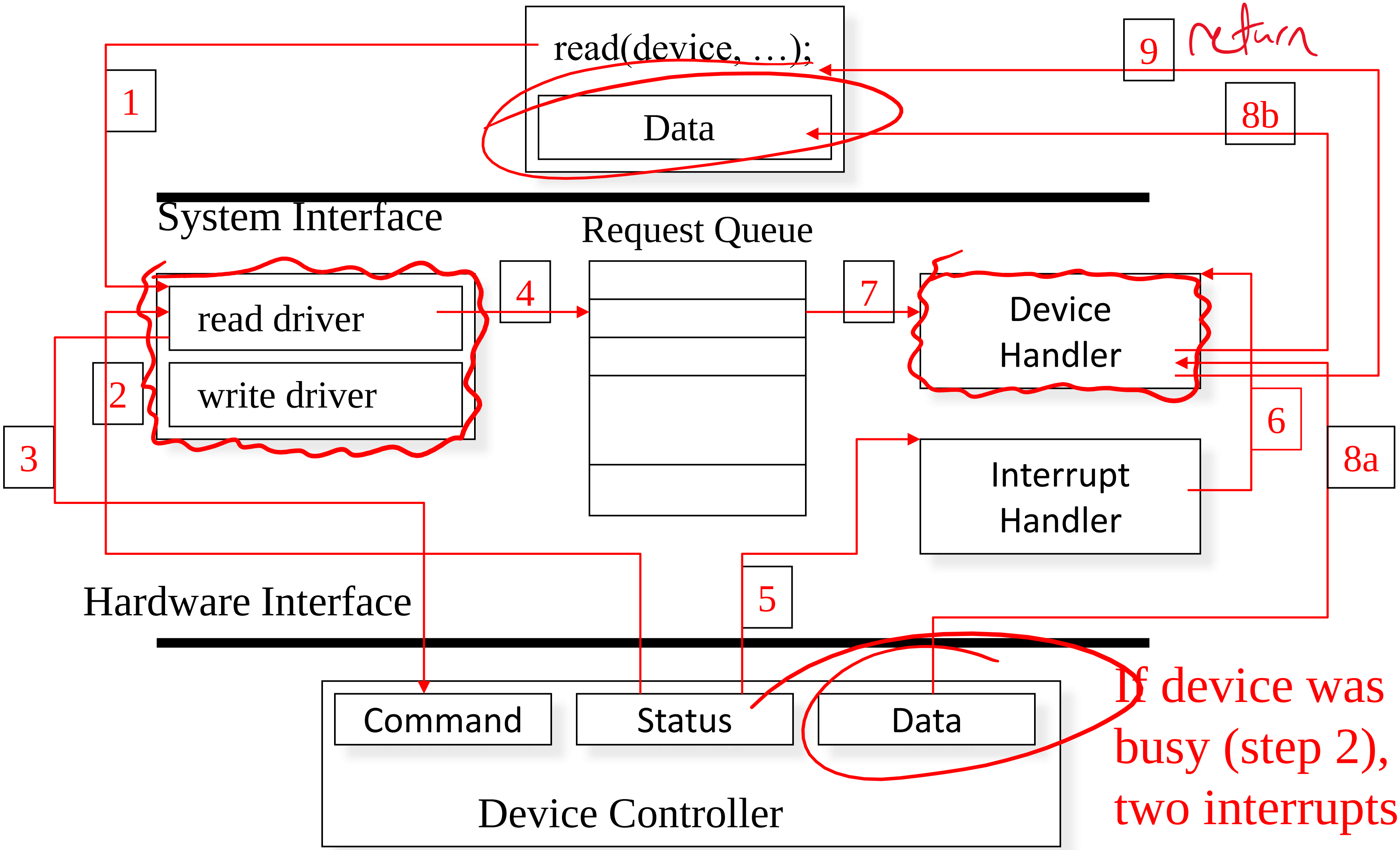
Interrupt Handler

- Prep: Disable interrupts

- 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 (step 2), two interrupts occur

When is Polling BETTER than Interrupt handling?

- Setting up the interrupts takes overhead
- Handling the interrupts takes overhead
- Handling the scheduling of the processes takes overhead

- If it is always a short wait for the IO then Polling is better
- If the wait is predictable then Polling is better

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

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
  - direct I/O with *interrupts*
    - More efficient than busy waiting, but still has overhead for every transfer
  - **DMA with interrupts**

Direct Memory Access (DMA)

- Idea: 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

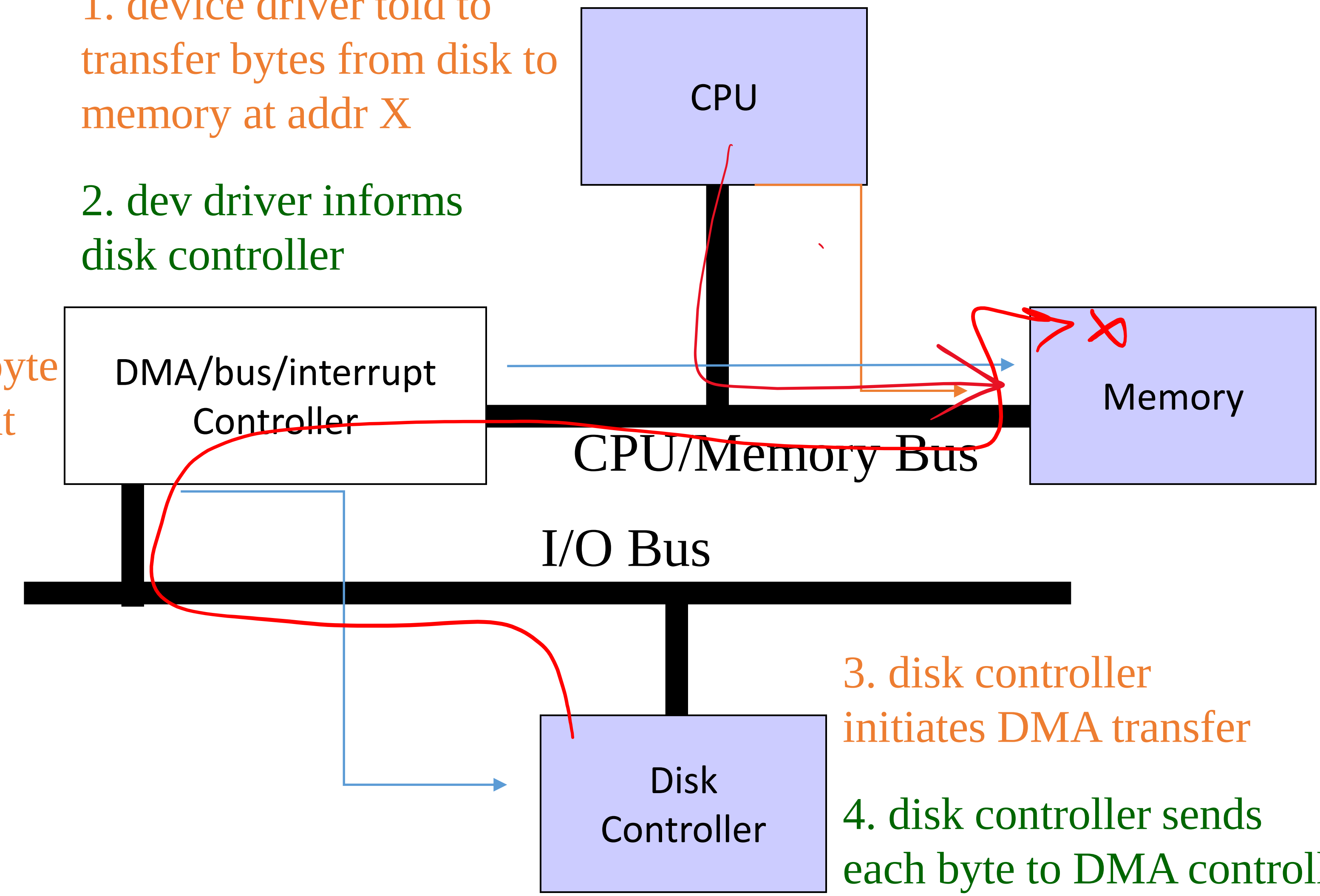
2. dev driver informs disk controller

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,

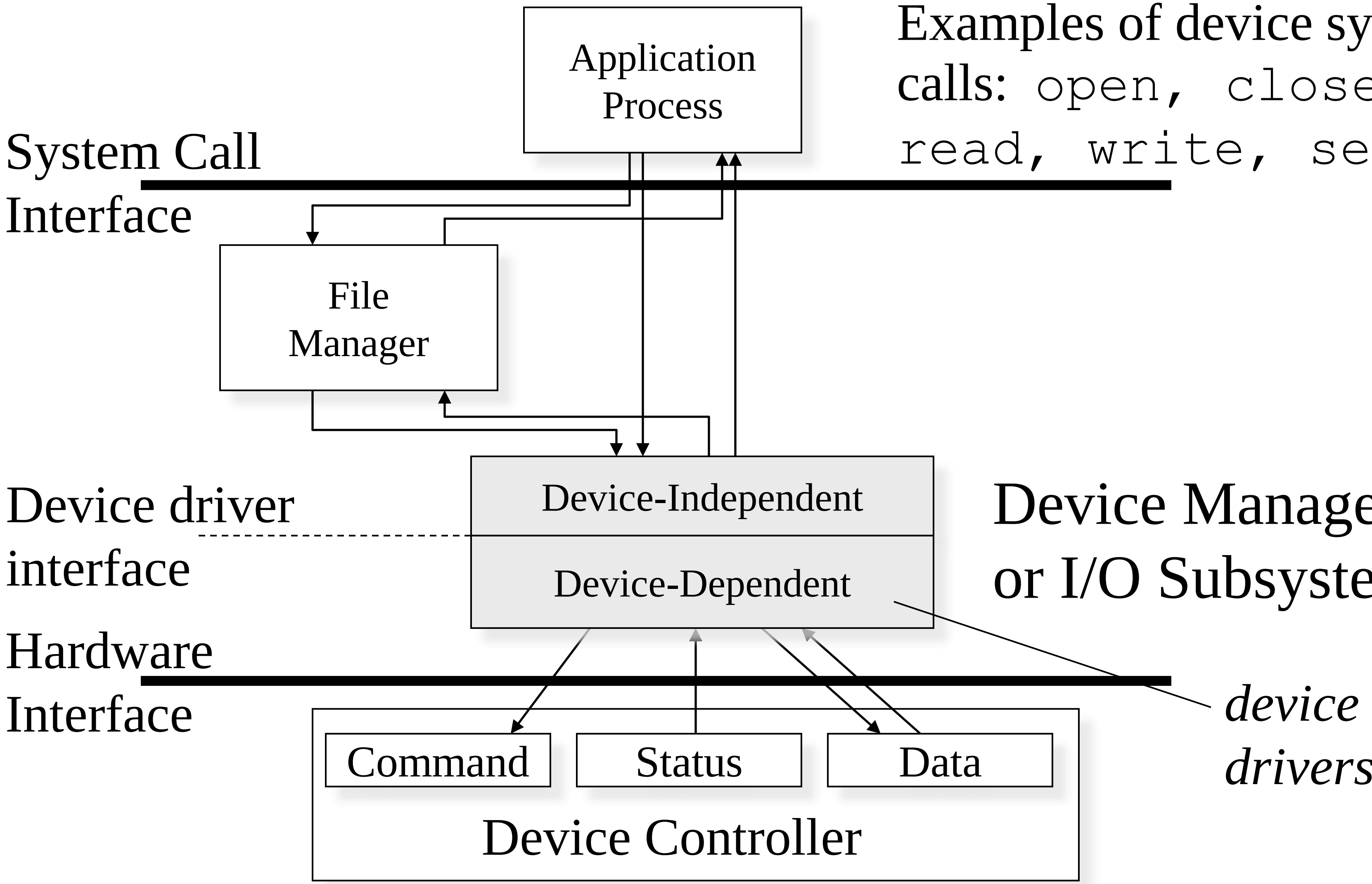
  how do they share main memory?

  - 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

Device Management Organization



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

Device Manager or I/O Subsystem

Port-Mapped I/O

- Port or port-mapped (non-memory 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.

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)

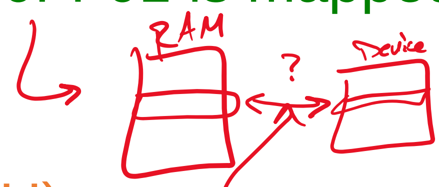
Port-Mapped I/O

- 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

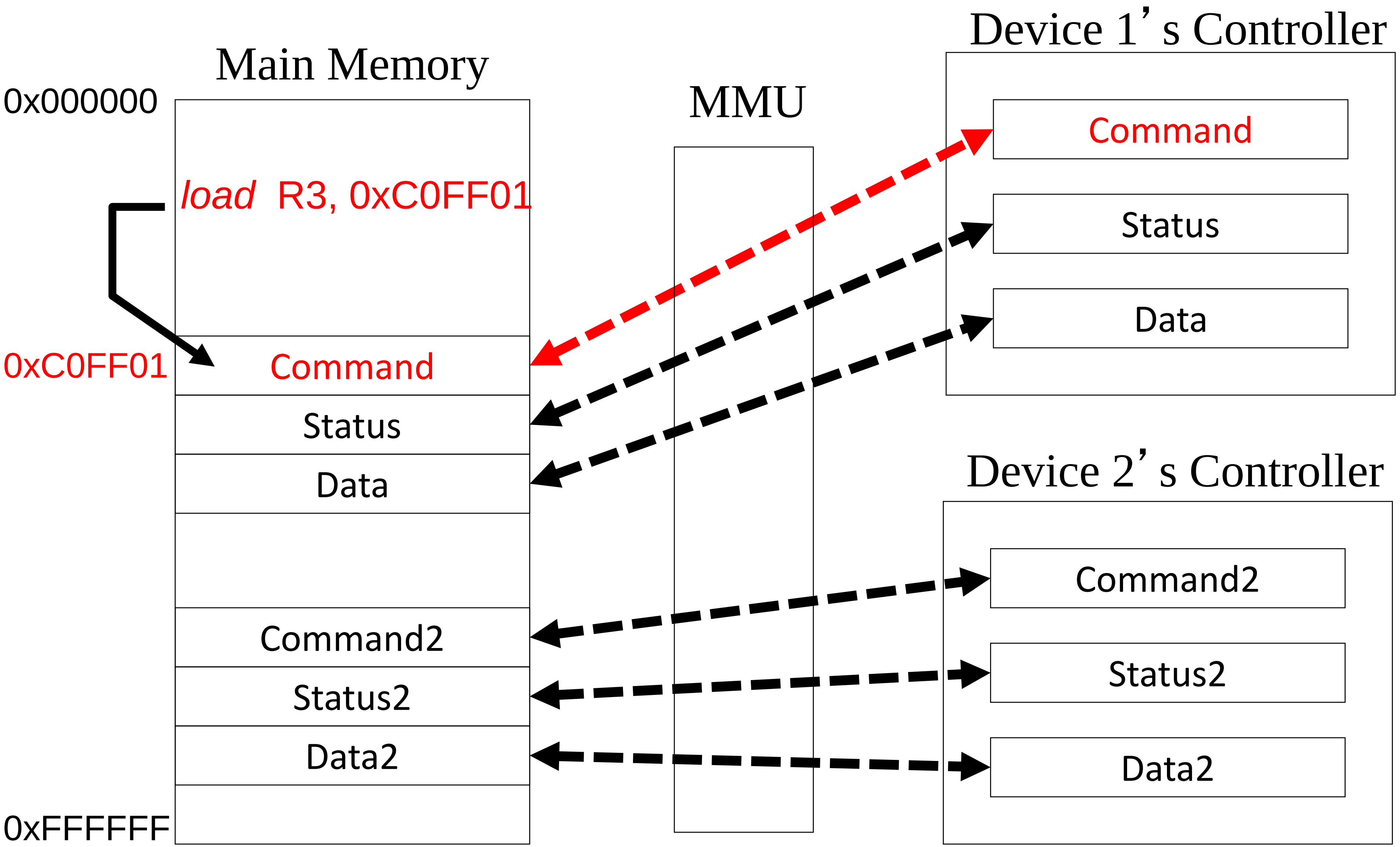
- Memory-mapped I/O: device registers and device memory are mapped to the system address space (system's memory)

- 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 to reflect on the device

Memory-Mapped I/O



University of Colorado Boulder

Memory-Mapped I/O

- 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 10-100 MB of memory, which in modern address spaces of GBs is quite modest
    - – so memory-mapped I/O is a small penalty

What is difference between Port and Memory Mapped IO?

- **Port mapped I/O** uses a separate, dedicated address space and is accessed via a dedicated set of microprocessor instructions.

- **Memory mapped I/O** is **mapped** into the same address space as program **memory** and/or user **memory**, and is accessed in the normal way.

Recap …

- What are the three device controller states?
  - Idle, Working, Busy
- What are the 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
    - large data transfer without using CPU

- What are the differences between Port and Memory-Mapped IO?
  - Only OS can access port registers at specific memory locations
  - Memory for device registers is mapped into user or kernel space and accessed in the same manner as any other memory

Lecture 5
Loadable Kernel Modules

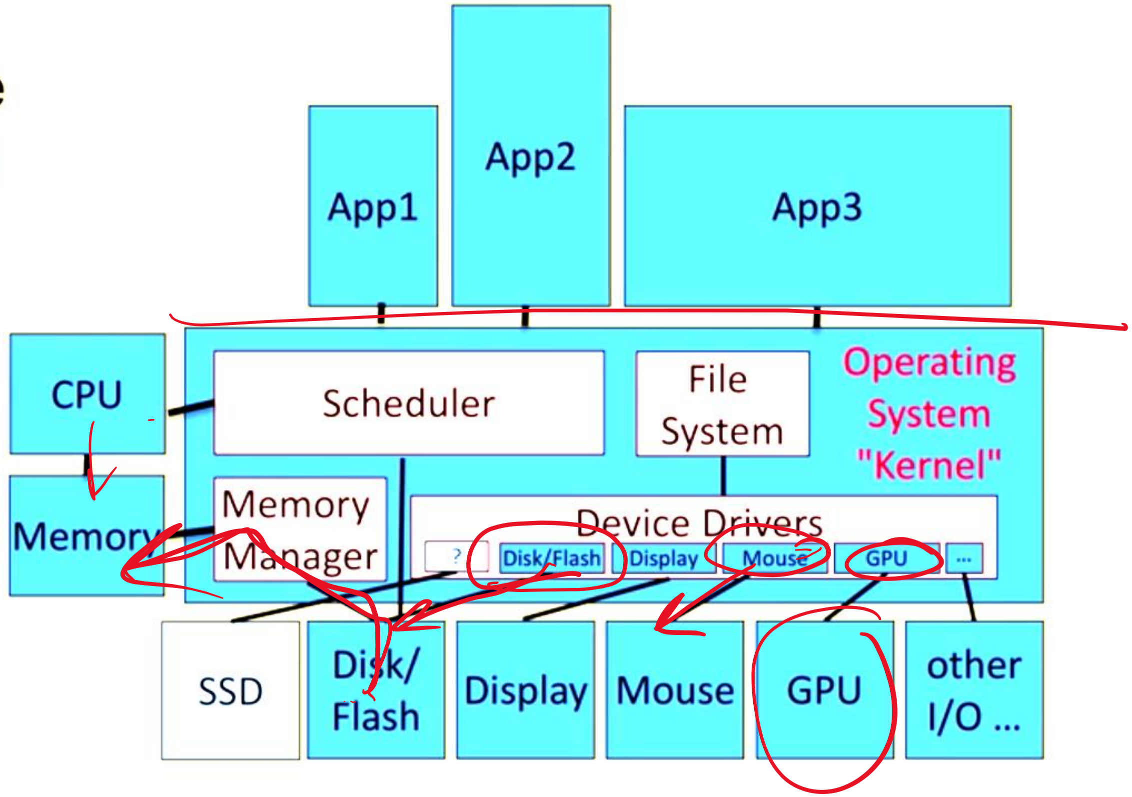
Device have both device-independent and device-dependent code

There is special device driver code associated with each different device connected to the system

1. ***Device-Independent*** API

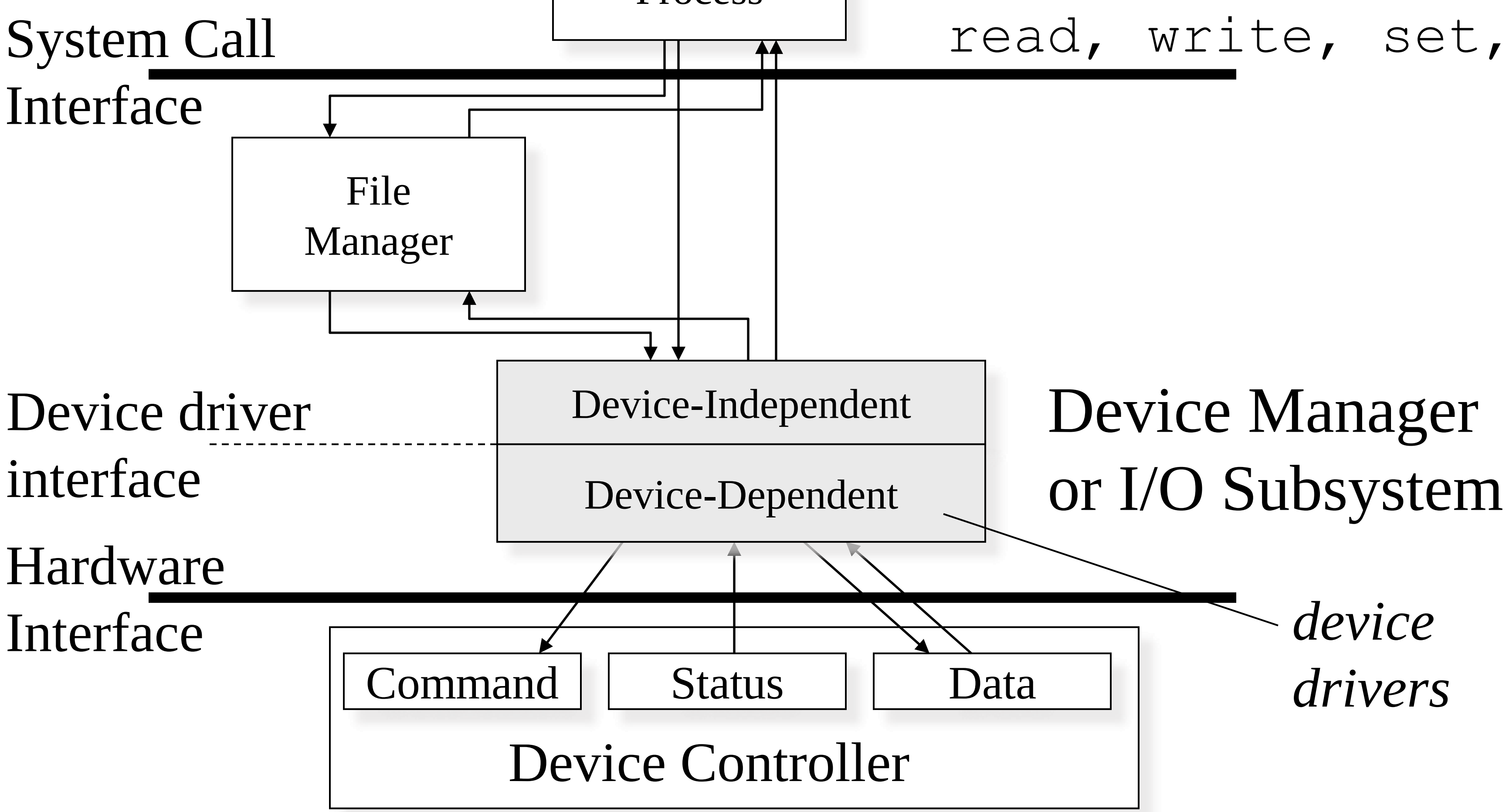
2. ***Device-Dependent*** driver code

3. ***Device Controller***



Device Management Organization

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

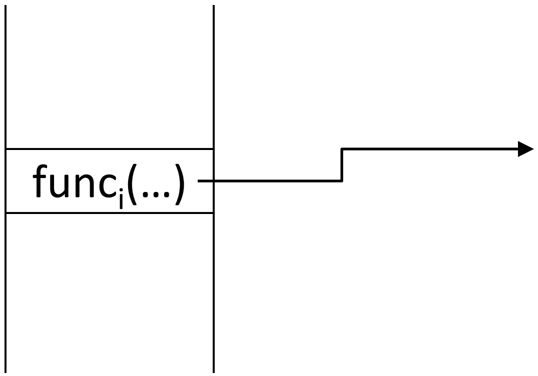


Device Independent Part

- 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

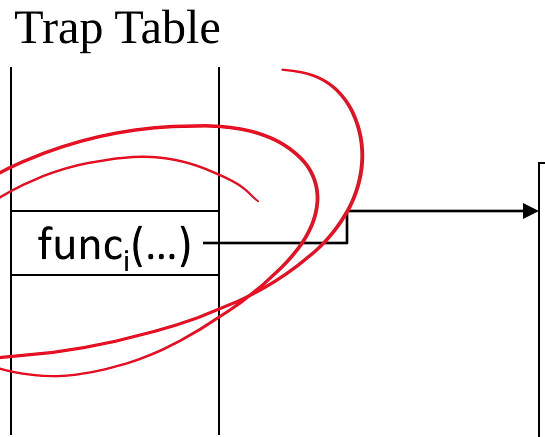



```
dev_func_i(devID, …) {
// Processing common to all devices
   …
   switch(devID) {
   case dev0:  dev0_func_i(…);
               break;
   case dev1:  dev1_func_i(…);
               break;
   …
   case devM:  devM_func_i(…);
               break;
   };
// Processing common to all devices
   …
}
```

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




```
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
   …
}
```

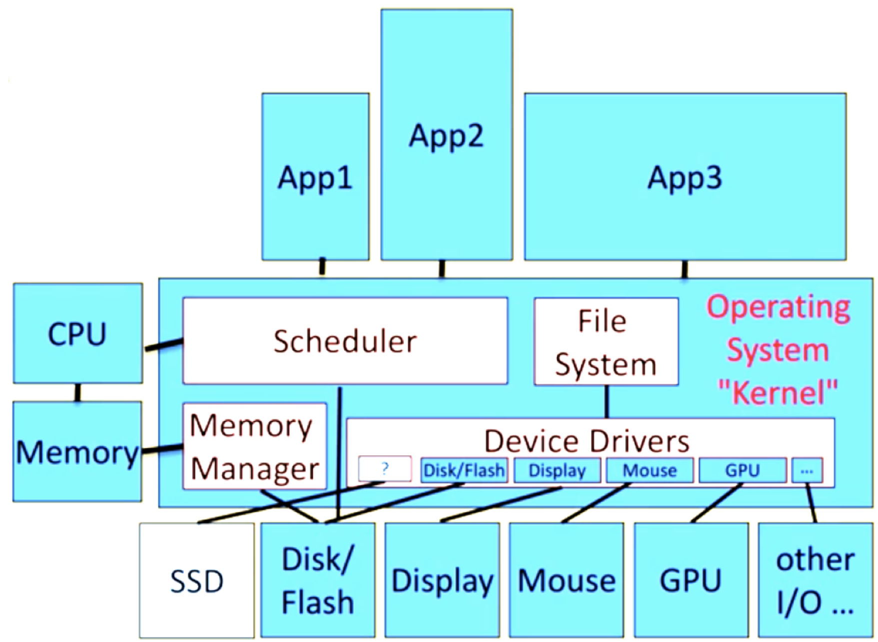
Adding a New Device

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

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

Device have both device-independent and device-dependent code

- Need a way to **Dynamically add** new code into the OS kernel when new device needs to be supported
- Load *Device-Dependent* driver code into kernel
- Only load the device drivers as needed
- No kernel recompilation for changes in the device driver



Loadable Kernel Modules

- LKM is an object file that contains code to **extend a running kernel**
- Windows (kernel-mode driver), Linux (LKM), OS X (Kernel extension: kext), VmWorks
- LKMs can be loaded and unloaded from kernel on demand at runtime

LKMs

- Offer an easy way to extend the functionality of the kernel without having to rebuild or recompile the kernel again
- Simple and efficient way to create programs that reside in the kernel and run in privileged mode
- Most of the drivers are written as LKMs

- What's out there in the kernel? See */lib/modules* for the all the LKMs
- *lsmod*: lists all kernel modules that are already loaded

How to write a kernel module?

- Kernel Modules are written in the C programming language.

- You must have a Linux kernel source tree to build your module.

- You must be running the **same kernel version** you built your module with to run it.

- Linux kernel object: ***.ko*** extension

Kernel Module: Basics

- A kernel module file has several typical components:
  - MODULE_AUTHOR(“your name”)
  - MODULE_LICENSE(“GPL”)
    - The license must be an open source license (GPL, BSD, etc.) or you will “taint” your kernel.
    - Tainted kernel loses many abilities to run other open source modules and capabilities.

Kernel Module: Key Operations

- int init_module(void)
  - Called when the kernel loads your module.
  - Initialize all your stuff here.
  - Return 0 if all went well, negative if something is not right.
- Typically, init_module( ) either
  - registers a handler for something with the kernel,
  - or replaces one of the kernel functions with its own code (usually code to do something and then call the original function)

Kernel Module: Key Operations

- void cleanup_module(void)
  - Called when the kernel unloads your module.
  - Free all your resources here.

Hello World Example

```
#include <linux/kernel.h>
#include <linux/module.h>
MODULE_AUTHOR(“Awesome Developer”);
MODULE_LICENSE(“GPL”);


 int init_module(void)
 {
 printk(KERN_ALERT “Hello world: I am a developer in CS3753
                                                        speaking from the Kernel”);
 return 0;
 }
```

Hello World Example

```
void cleanup_module(void)
{
printk(KERN_ALERT “Goodbye from a developer in CS3753,
                                          I am exiting the Kernel”);
}
```

Building Your Kernel Module

- Accompany your module with a 1-line GNU Makefile:
  - obj-m += hello.o
  - Assumes file name is "hello.c"
- Run the make command:
  - make -C <kernel-src> M=`pwd` modules
  - Produces: hello.ko
- Assumes current directory is the module source.

obj-$(CONFIG_FOO) += foo.o

- Good definitions are the main part of the kbuild Makefile.

- The most simple kbuild makefile contains one line:

```
obj-$(CONFIG_FOO) += foo.o
```

This tell **kbuild** that there is one object in that directory named foo.o and foo.o will be built from foo.c or foo.S.

- $(CONFIG_FOO) evaluates to either y (for built-in) or m (for module). If CONFIG_FOO is neither y nor m, then the file will not be compiled nor linked.

Loading Your Kernel Module: *insmod*

- *Use insmod* to manually load your kernel module

  sudo insmod helloworld.ko

- *insmod* makes an *init_module* system call to load the LKM into kernel memory
- *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

Where is our Hello World message

- Dmesg

- /var/log/system.log

Unloading Your Kernel Module

- Use *rmmod* command

  rmmod helloworld

- Should print the Goodbye message

Kernel Module Dependencies: *modprobe*

- insmod/rmmod can be cumbersome...
  - You must manually enforce inter-module dependencies.
- *modprobe* automatically manages dependent modules
  - Copy hello.ko into /lib/modules/<version>
  - Run depmod
  - modprobe hello / modprobe -r hello
- Dependent modules are automatically loaded/unloaded.

- *depmod* creates a Makefile-like dependency file,

  based on the symbols it finds in the set of modules mentioned on the command line or from the directories specified in the configuration file
- This dependency file is later used by *modprobe* to automatically load the correct module or stack of modules

modinfo command

- .ko files contain an additional .modinfo section where additional information about the module is kept
  - Filename, license, dependencies, …
- modinfo command retrieves that information

      modinfo hello.ko

How to Write an LKM?

- Begin by writing your source code
  - see helloModule.c
- Write a Makefile with the following:
  - obj-m:=helloModule.o
  - make -C /lib/modules/$(uname -r)/build M=$PWD modules
- This should generate a *.ko file in your PWD