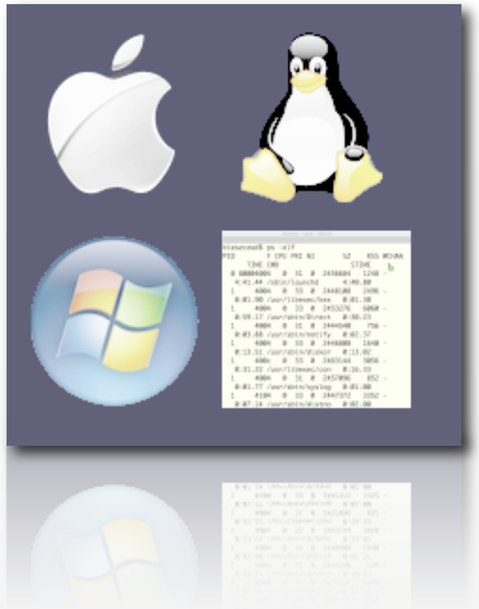


Lecture 14: I/O Systems

COMP362 Operating Systems
Prof. AJ Bieszczad

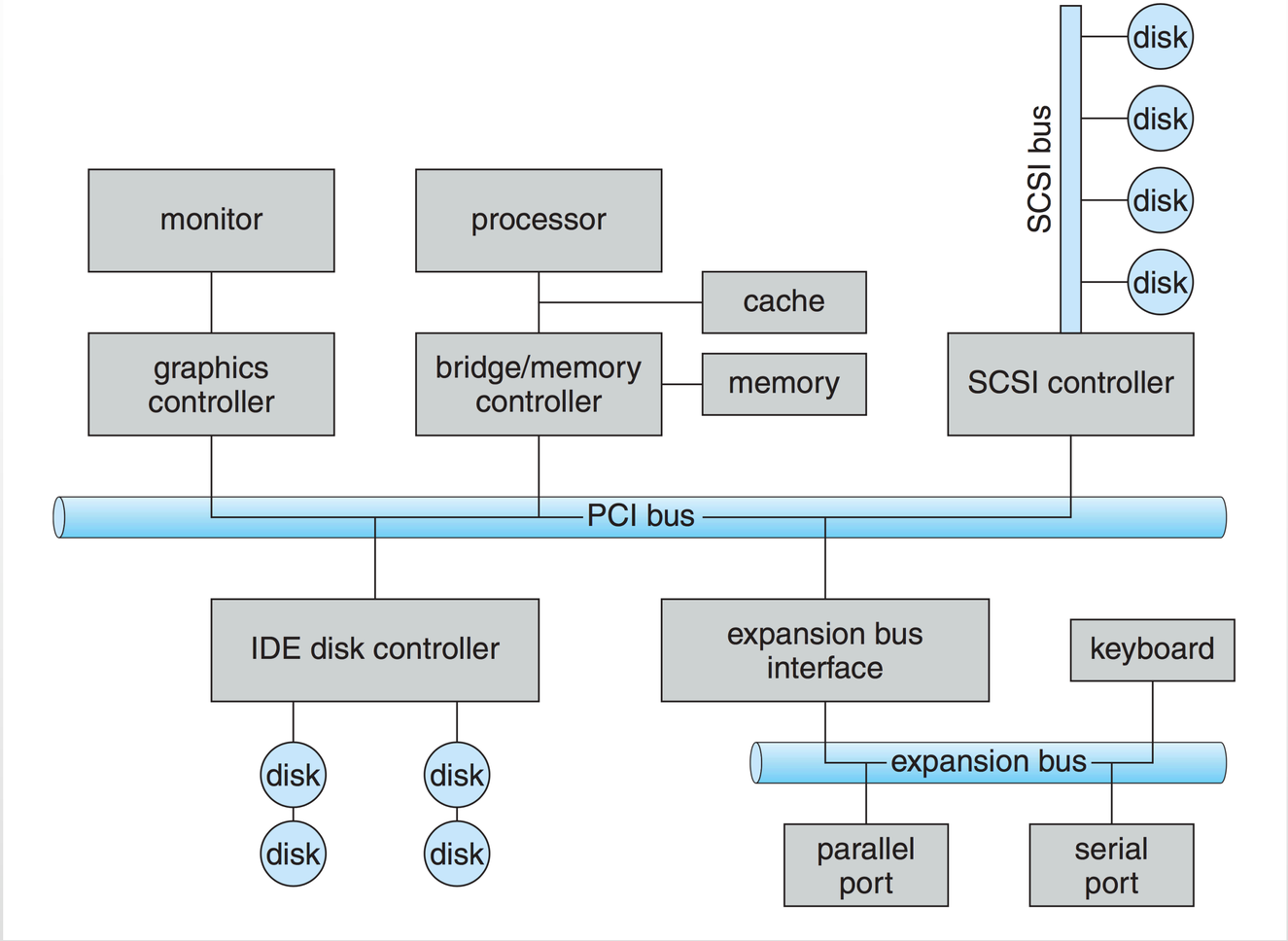
Outline

- I/O Hardware
- Taxonomy of I/O Devices
- Application I/O Interface
- Kernel I/O Subsystem
- Kernel I/O Data Structures
- Performance

- Using packed C structs with bit fields to communicate with devices
- Linux kernel modules
- Linux device drivers

I/O Hardware

- Very large variety of I/O devices
- Common concepts
  - **port**
    - 1-2-1 communication channel
  - **bus**
    - many-2-many comm channel
    - daisy chain or shared direct access
  - **I/O controller**
    - host adapter
- I/O instructions to control devices
  - POSIX: **`ioctl()`**

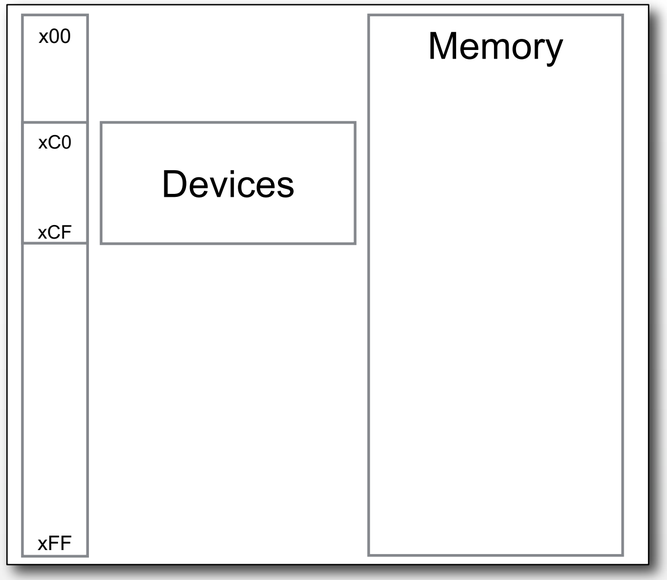


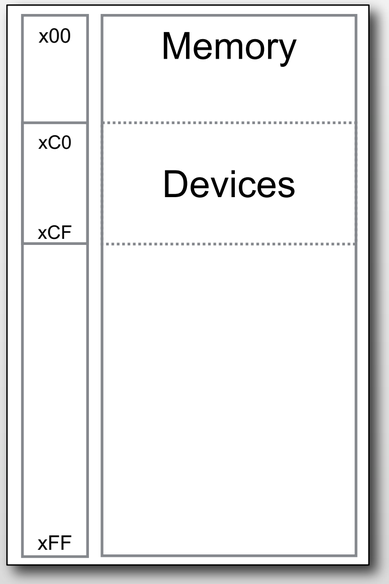
PMIO vs. MMIO

- Devices have addresses

  - **port-mapped I/O** (**PMIO**)
    - use special I/O instructions to communicate with devices
    - no “carving” from memory
      - i.e., the same addresses can be used to access memory

  - **memory-mapped I/O** (**MMIO**)
    - same instructions and same address bus used to communicate with both devices and the memory
    - part of memory “carved” for devices
      - i.e., the addresses used to communicate with devices cannot be used to access memory





Selection of Device I/O Port Locations on PCs

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)

Registers in a Typical I/O Port

- An I/O port typically consists of four registers:
  - The **data-in register** is read by the host to get input.
  - The **data-out register** is written by the host to send output.
  - The **status register** contains bits that can be read by the host. These bits indicate states, such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether a device error has occurred.
  - The **control register** can be written by the host to start a command or to change the mode of a device.
    - For instance, a certain bit in the control register of a serial port chooses between full-duplex and half-duplex communication, another bit enables parity checking, a third bit sets the word length to 7 or 8 bits, and other bits select one of the speeds supported by the serial port.

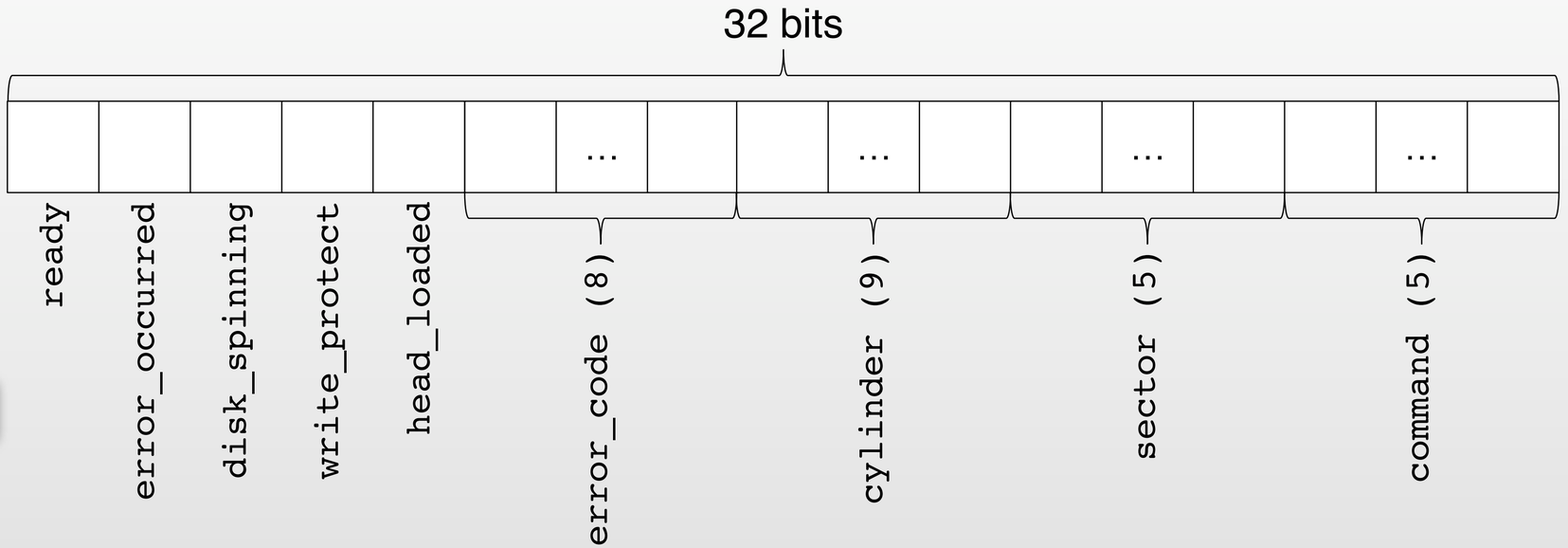
Using Bit Fields to Communicate with Devices

- C structs accept **bit fields**
- This format is ideal for packing registers for communication with devices
- For example, the following is a sample register encoding for communication with a hard drive:

```
typedef struct disk_register {
  unsigned ready:1;
  unsigned error_occured:1;
  unsigned disk_spinning:1;
  unsigned write_protect:1;
  unsigned head_loaded:1;
  unsigned error_code:8;
  unsigned cylinder:9;
  unsigned sector:5;
  unsigned command:5;
} DISK_REGISTER;

// ...

while ( ! disk_reg.ready )
   ; // wait
```

field width in bits

access the same way as regular struct fields



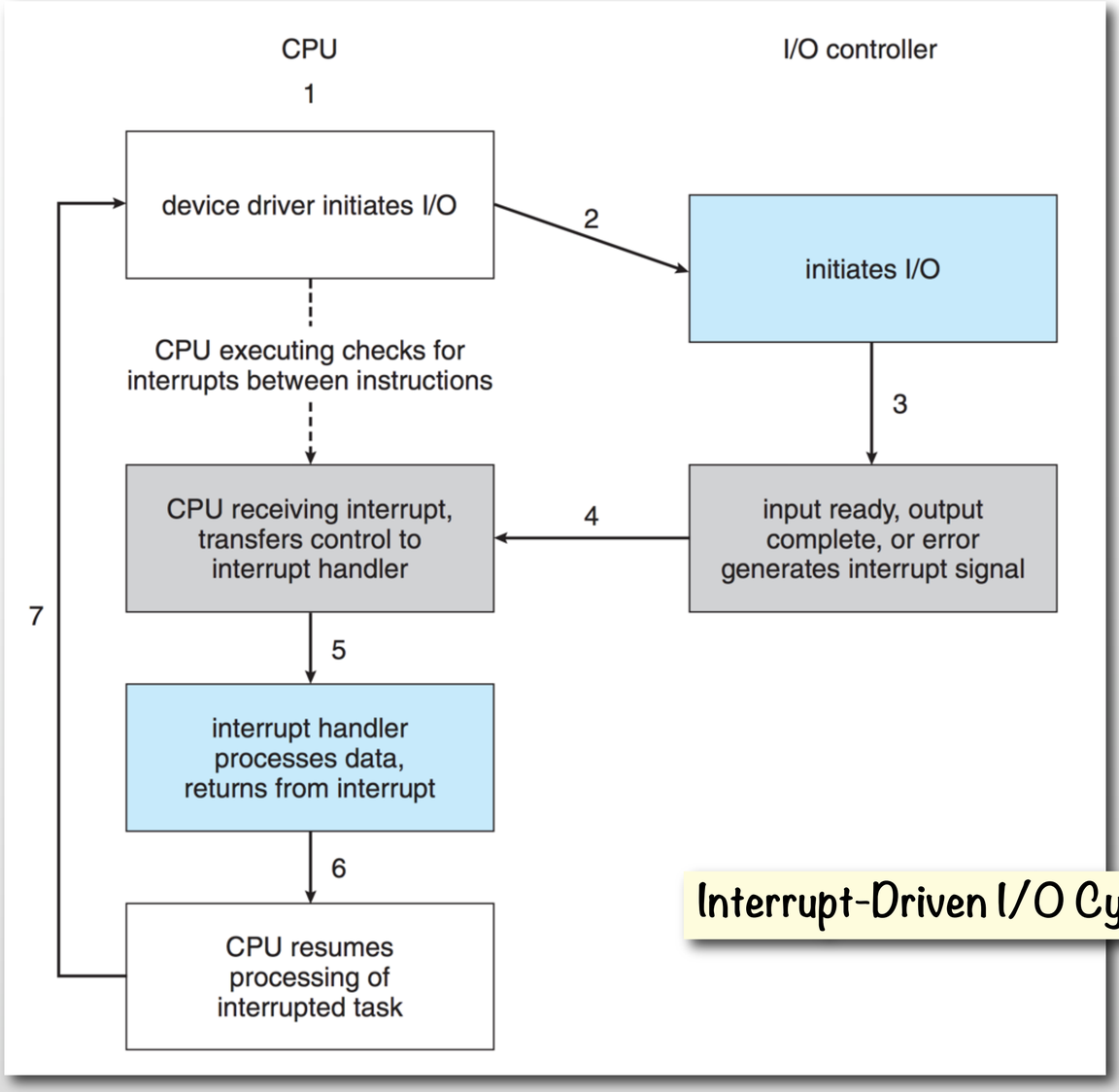
Polling

- One way of communicating with a device is to constantly poll it
- Determines state of device
  - **command-ready**
    - initiate the operation (i.e., write a command and data) if ready
  - **busy**
    - have to wait
  - **error**
    - must do something
      - repeat?
      - set error number indicator to something (e.g., errno on Unix)
- Busy-wait cycle to wait for I/O from device
  - might not be acceptable from the performance perspective
  - can use a timer as a remedy

- Interrupts commonly used as a much more efficient mechanism
  - in hardware through interrupt controller
  - or, in memory subsystem through memory mapping controller

Interrupts

- **CPU interrupt**
  - request triggered by I/O device
    - checked between executing instructions
- **Interrupt handlers** called for specific interrupts
- **Interrupt vector**
  - to dispatch interrupt to correct handler
    - may be based on some priority
- **Maskable interrupt**
  - to ignore or delay some interrupts
  - not all interrupts are maskable
- Interrupt mechanism also used for exceptions



Interrupt-Driven I/O Cycle

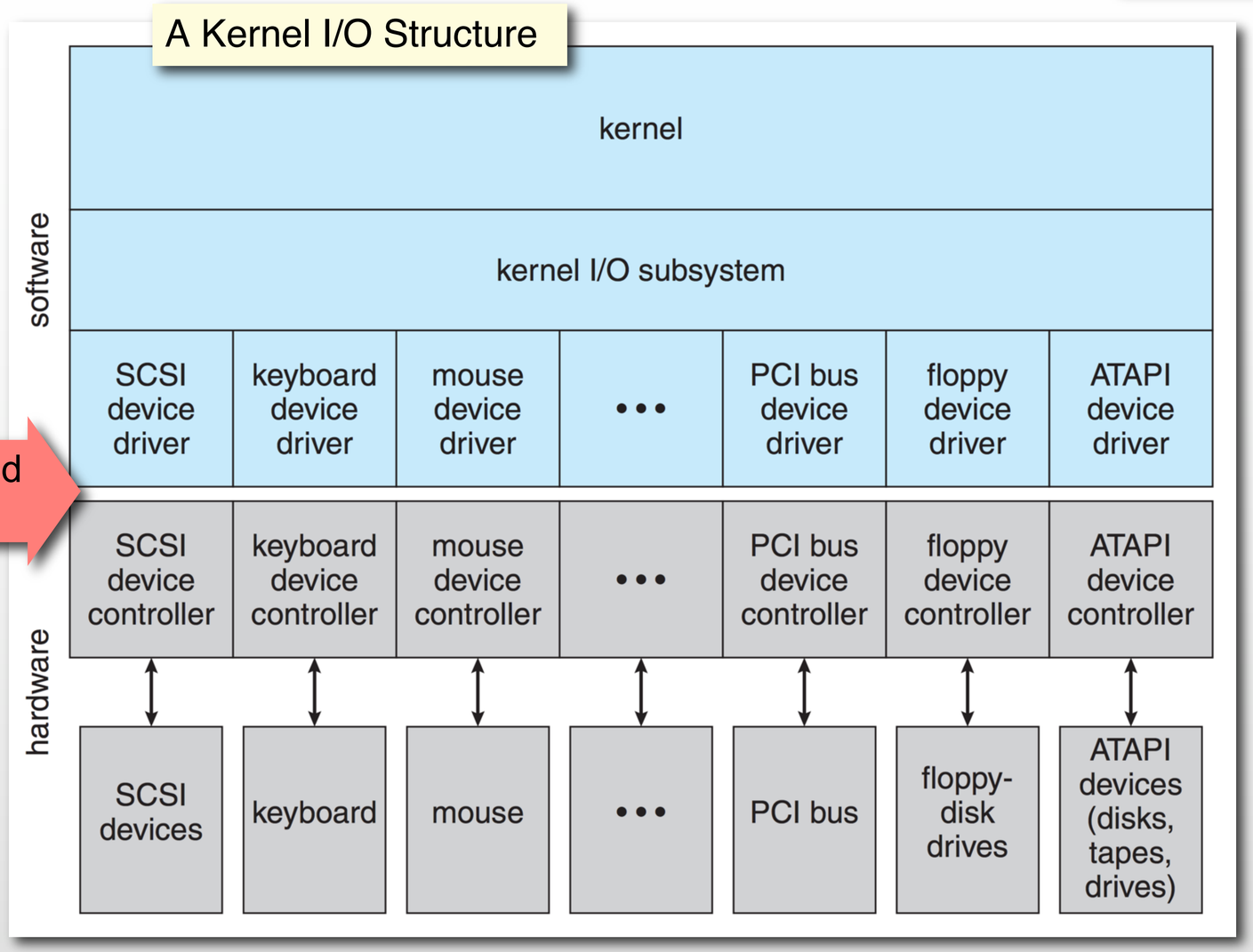
Intel Pentium Processor Event-Vector Table

- Every interrupt has a number
  - usually an index into event-vector table
    - contains pointers to the corresponding interrupt handlers

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

Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
  - each class serves similar devices
- Device-driver layer hides differences among I/O controllers from kernel



Taxonomy of I/O Devices

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Types of Devices

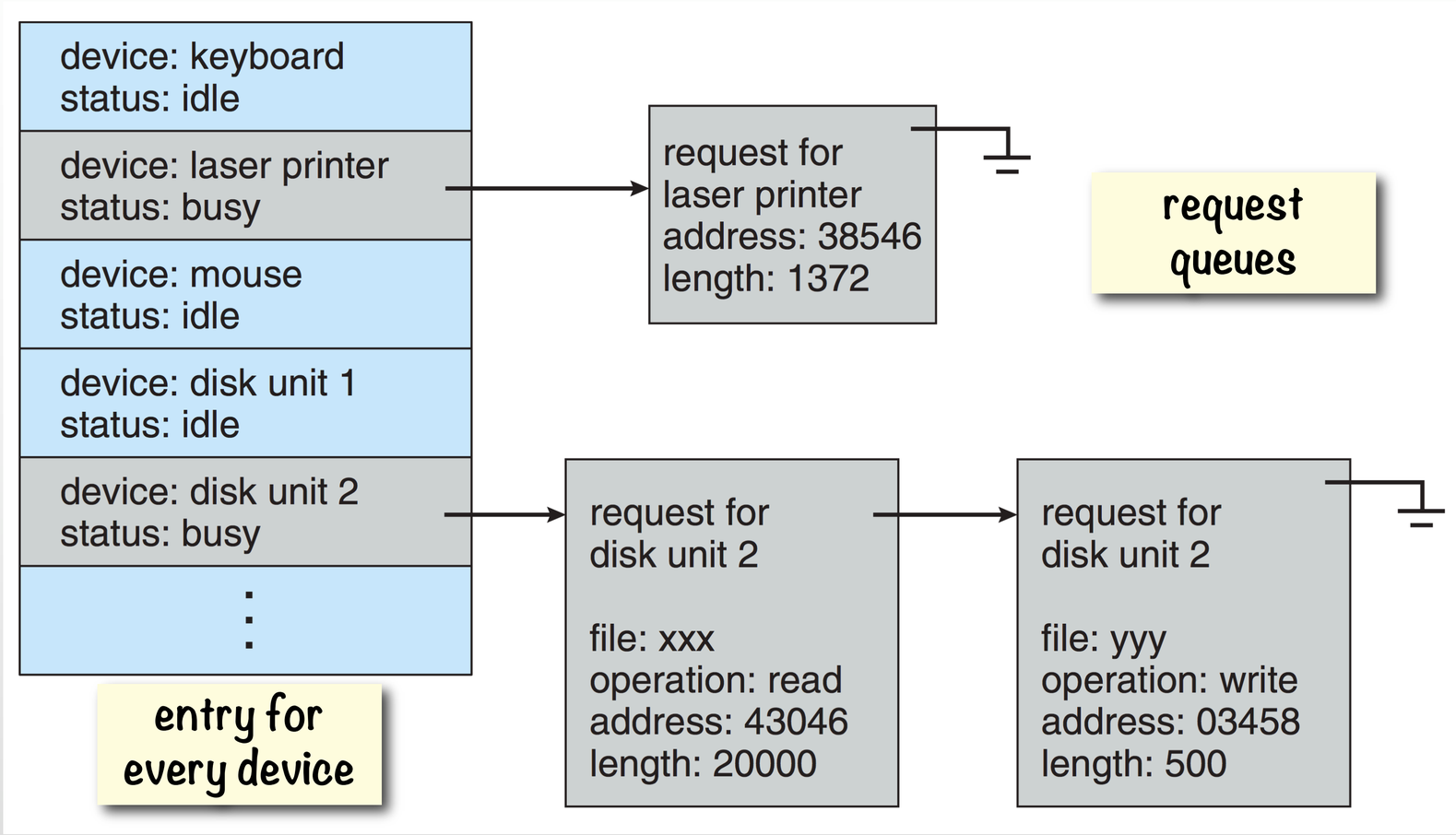
- **Block devices** include disk drives
  - commands include **`read()`**, **`write()`**, **`seek()`**
  - raw I/O or file-system access
  - memory-mapped file access possible
- **Character devices** include keyboards, mice, serial ports
  - commands include **`get`**, **`put`**
  - libraries layered on top allow line editing
    - e.g., **`readline`**
- **`ioctl()`** (POSIX) can be used to handle aspects of I/O without new system calls
- **Network devices**
  - Varying enough from block and character to have own interface
    - e.g., latency
  - Unix (including Linux and Mac) as well as Windows include socket interface
    - separates network protocol from network operation
    - includes multiplexing functionality
      - i.e., ports (software kind!)
  - approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)
- **Clocks and Timers**
  - provide current time, elapsed time, timer
  - programmable interval timer used for timings, periodic interrupts

Kernel I/O Subsystem

- Handling I/O is one of the most important functions of the kernel
- Functions
  - Scheduling I/O
  - Buffering
  - Caching
  - Spooling
  - Error handling
  - Device reservation
  - I/O protection
  - Optimization

I/O Scheduling

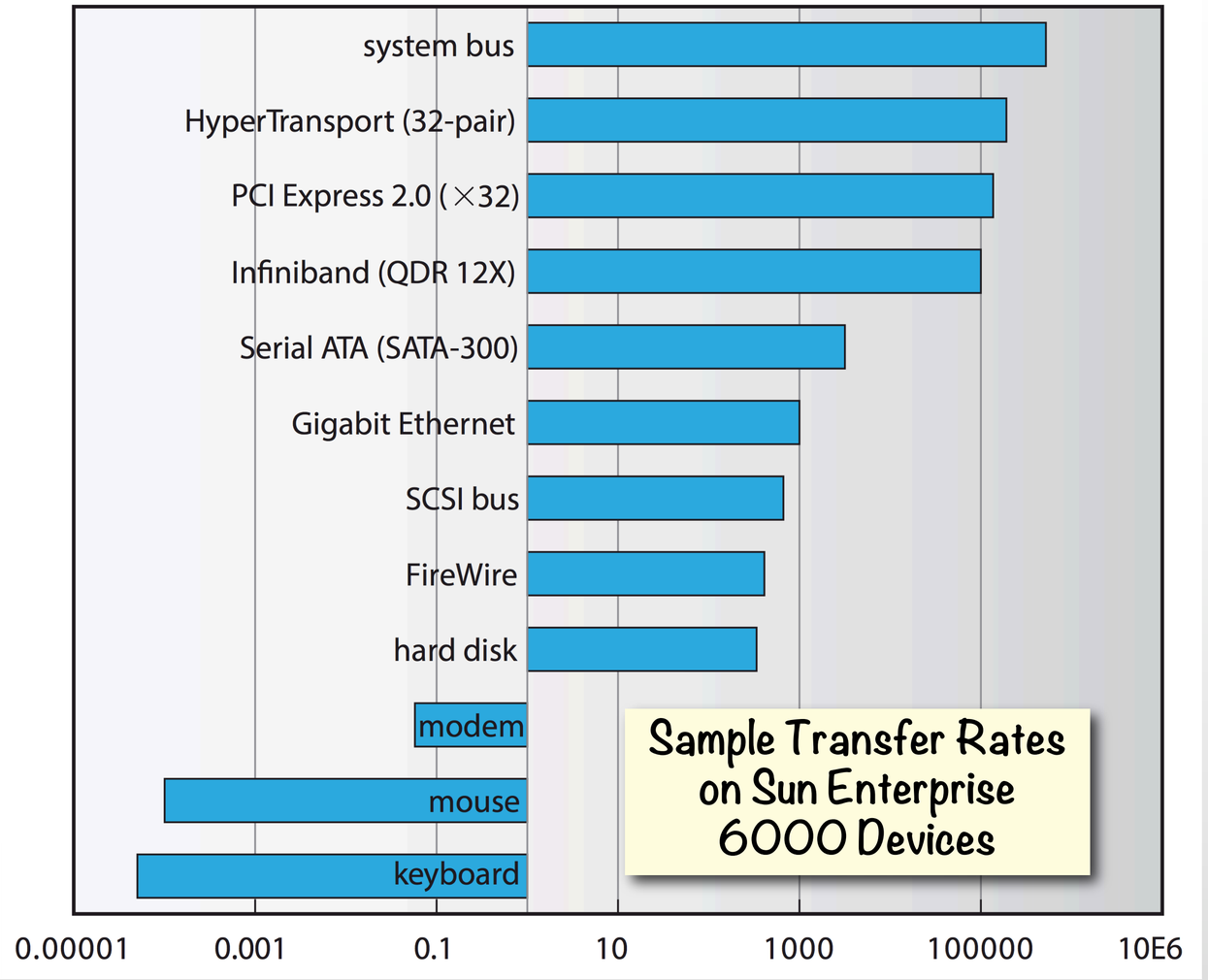
- Scheduling
  - some I/O request ordering via per-device queue
    - optimization possible; e.g., scheduling writes to a disk
  - OSes may provide some fairness
- Device Status Queue



I/O Buffering

- Buffering - store data in memory while transferring between devices
  - to cope with device **speed mismatch**
  - to cope with device **transfer size mismatch**
  - to maintain "**copy semantics**"
    - collecting coherent chunks

- **Double buffering** concept
  - switching between two buffers; e.g., in graphics to prevent flickering



I/O Caching, Spooling, Reservation

- **Caching** - fast memory holding copy of data
  - always just a copy in the cache
    - in contrast to a buffer that holds the only copy for some time
  - key to performance
- **Spooling** - hold output for a device
  - if device can serve only one request at a time
    - e.g., printing
  - may resume after interruptions
    - in contrast to buffering and in-memory caching
- **Device reservation** - provides exclusive access to a device
  - application-level control
    - system calls for allocation and deallocation
    - watch out for deadlock!

I/O Error Handling

- OS can recover from many errors
  - disk read,
  - device unavailable,
  - transient write failures
  - etc.
- Most system calls return an error number or code when I/O request fails
  - often in Unix a positive return number is an error indication
    - zero means OK
  - **`errno`** global variable set by system calls and some library functions in the event of an error to indicate what went wrong

```
$ sudo apt install errno
$ errno 24
OS error code  24:  Too many open files
$ errno –l
EPERM 1 Operation not permitted
ENOENT 2 No such file or directory
...
```

- System error logs hold problem reports
  - check /var/log for log files with **`dmesg`**

```
$ man dmesg
```

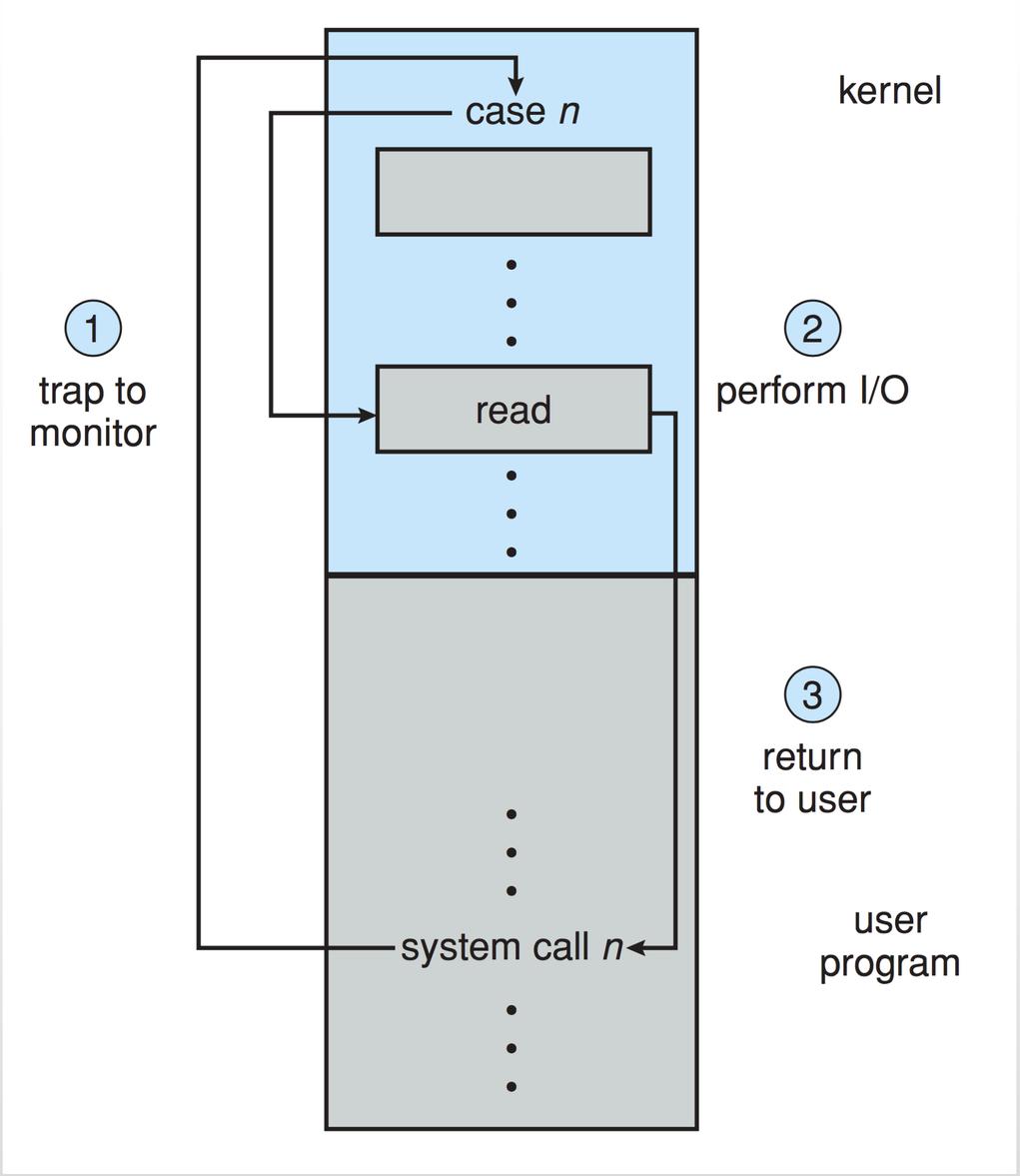
- Some systems have a nice presentation layer
  - e.g. on Mac, **`Console.app`**
  - system-specific errors

```
$ macerror 24
Mac OS error 24 (dsNoPk7): package 7 not present
```

I/O Protection

- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
  - for example:
    - several processes writing directly to video memory or sending data to a network card
    - direct access to network cards would allow for spoofing and eavesdropping

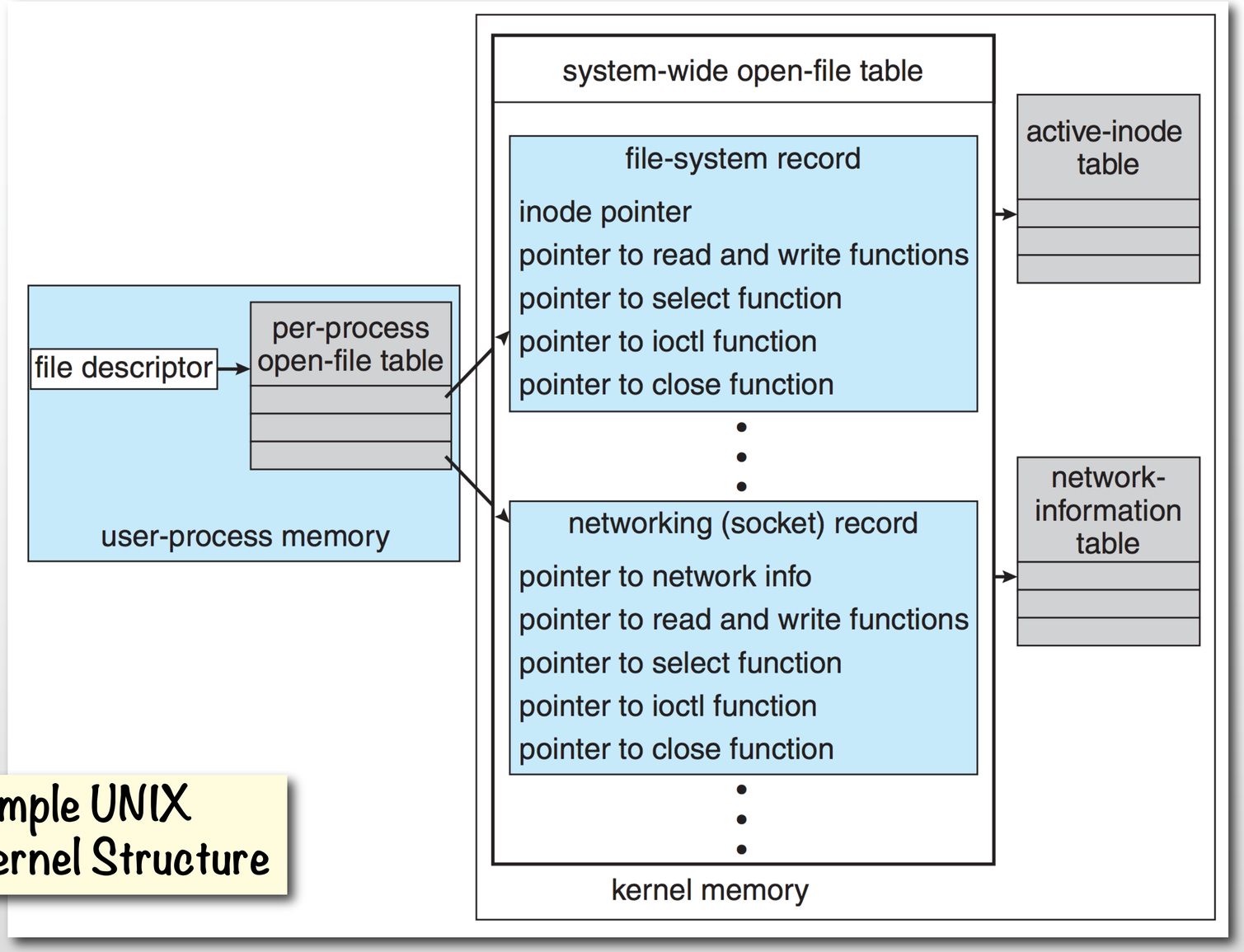
- Therefore, all I/O instructions are defined to be privileged
  - I/O must be performed via system calls
  - memory-mapped and I/O port memory locations must be protected too



Kernel Data Structures

- Kernel keeps state info for I/O components, including
  - open file tables (devices are pseudo-files)
  - network connections,
  - character device states,
  - etc.

- Many, many complex data structures to track
  - buffers,
  - memory allocation,
  - “dirty” blocks,
  - and so on...



Sample UNIX I/O Kernel Structure

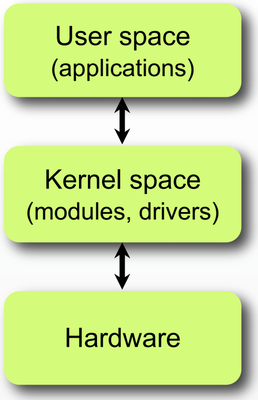
Improving Performance

- I/O is a major factor in system performance:
  - demands CPU to execute device driver, kernel I/O code
  - context switches due to interrupts
  - data copying
  - network traffic especially stressful due to common response latency

- Potential remedies
  - reduce data copying
  - reduce number of interrupts by using large transfers, smart controllers, polling
  - use DMA (direct memory access)
  - use specialized processors for I/O - I/O channels

Linux Device Drivers

- Can be implemented as a Linux module
- A module can be installed or built into the kernel
  - we will look at installable Linux modules
    - http://www.cyberciti.biz/tips/compiling-linux-kernel-module.html
  - for the built-in option, you need to rebuild Linux kernel with the modules
    - https://help.ubuntu.com/community/Kernel/Compile
- Linux modules are implemented similarly to FUSE
  - there is a mapping between system calls and user provided functions
    - see next slide
- Device driver can be accessed through the UNIX file paradigm



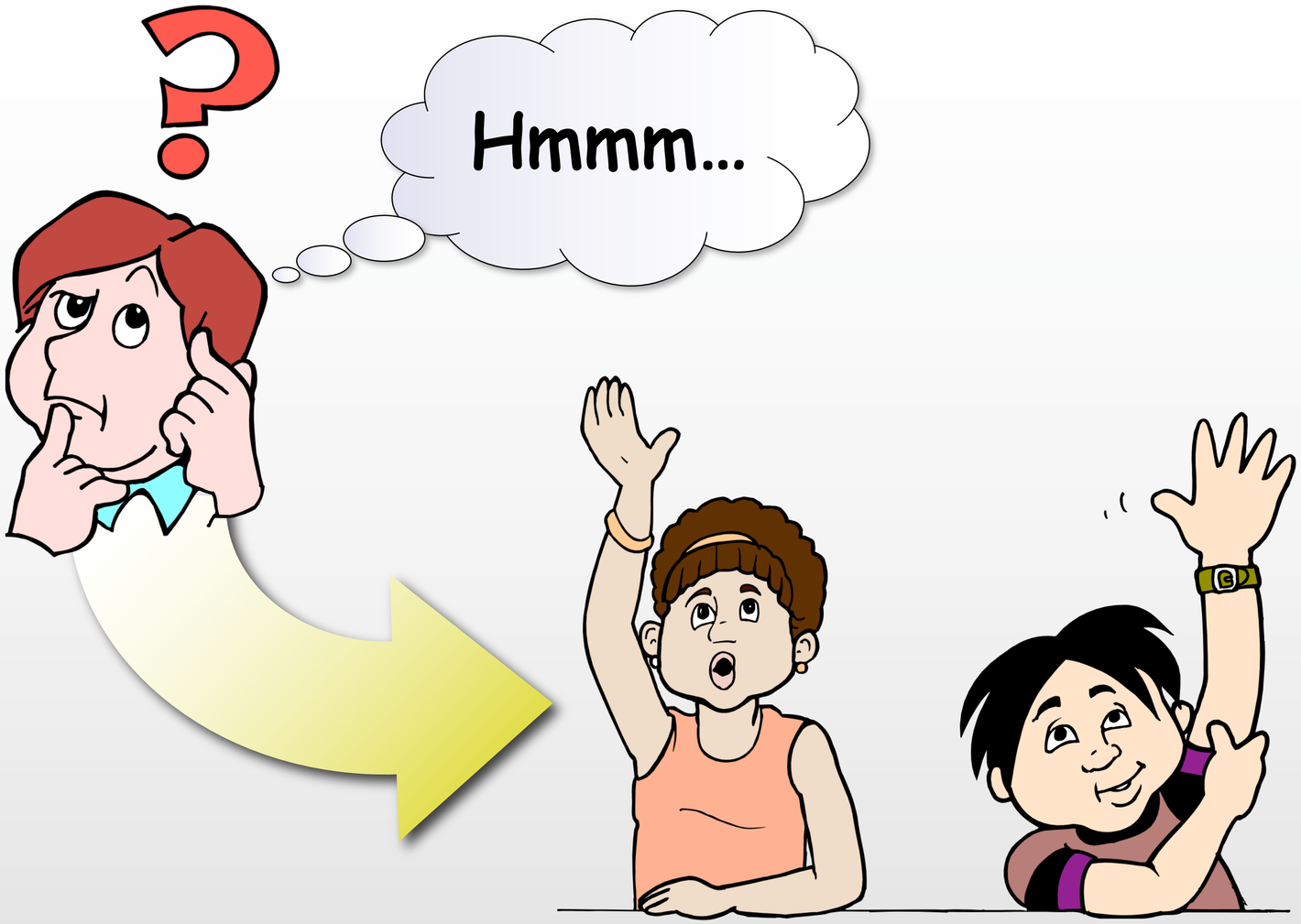

```
typedef struct disk_register
{
  unsigned ready:1;
  unsigned error_occured:1;
  unsigned disk_spinning:1;
  unsigned write_protect:1;
  unsigned head_loaded:1;
  unsigned error_code:8;
  unsigned cylinder:9;
  unsigned sector:5;
  unsigned command:5;
} DISK_REGISTER;
```

```
fd = open("/dev/sim_dev", O_RDWR | O_SYNC);

write(fd, msg, strlen(msg) + 1);
printf("'%s' written to /dev/sim_dev\n", msg);

read(fd, receive_buffer, 128);
printf("'%s' read from /dev/sim_dev\n", receive_buffer);

DISK_REGISTER register = {0, 0, 0, 0, 0, 0, 0, 0, 0};
register.cylinder = 100;
register.sector = 20;
register.command = 0xA; // arbitrary example; must fit 5 bits
ioctl(fd, IOCTL_WRITE, &register);

ioctl(fd, IOCTL_READ, &register);
printf("DEVICE RESPONSE: '%s'\n",
       !register.error_occurred ? "OK" : "ERROR");

close(fd);
```

Linux Driver Implementation

```
// open function - called when the "file" /dev/sim_dev
// is opened in userspace
static int sim_dev_open (
   struct inode *inode, struct file *file) {...}

// close function - called when the "file" /dev/sim_dev
// is closed in userspace
static int sim_dev_release (
   struct inode *inode, struct file *file) {...}

// read function called when /dev/sim_dev is read
static ssize_t sim_dev_read(
   struct file *filp,
   char __user *buf,
   size_t count,
   loff_t *f_pos) {...}

// write function called when /dev/sim_dev is written
static ssize_t sim_dev_write(
   struct file *filp,
   char __user *buf, size_t count,
   loff_t *f_pos) {...}

// ioctl function called when /dev/sim_dev is sent
// I/O control
static int sim_dev_unlocked_ioctl(
   struct file *file,
   unsigned int command,
   unsigned long arg) {...}
```

```
// mapping of file operations to the driver functions
struct file_operations sim_dev_file_operations = {
  .owner      = THIS_MODULE,
  .llseek     = NULL,
  .read       = sim_dev_read,
  .write      = sim_dev_write,
// .readdir    = NULL,
  .poll       = NULL,
  .ioctl      = sim_dev_unlocked_ioctl,
  .mmap       = NULL,
  .open       = sim_dev_open,
  .flush      = NULL,
  .release    = sim_dev_release,
  .fsync      = NULL,
  .fasync     = NULL,
  .lock       = NULL,
// .readv      = NULL,
// .writev     = NULL,
};
// on loading a module into the kernel
static int sim_dev_init_module (void) {...}
//  on removal of module from kernel
static void sim_dev_cleanup_module(void) {...}
module_init(sim_dev_init_module);
module_exit(sim_dev_cleanup_module);

MODULE_AUTHOR("http://faculty.csuci.edu/aj.bieszczad");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simulated Device Linux Device Driver");
```



COMP362 Operating Systems

Prof. AJ Bieszczad