

I/O systems

Some of the following slides adapted from Matt Welsh.

**Some slides are from Tanenbaum, Modern Operating Systems 3 e, (c)
2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639**

I/O systems

- **The two main jobs of a computer are I/O and processing.**
 - In many cases, the main job is I/O and the processing is merely incidental, e.g.
 - browse a web page or
 - edit a file
- **I/O system provides the means for the computer to interact with the rest of the world.**

I/O devices in OS

I/O devices vary greatly

- **Data rate:**
 - may vary by several orders of magnitude
- **Complexity of control:**
 - exclusive vs. shared devices
- **Unit of transfer:**
 - stream of bytes vs. block-I/O
- **Data representations:**
 - character encoding, error codes, parity conventions
- **Error conditions:**
 - consequences, range of responses
- **Applications:**
 - impact on resource scheduling, buffering schemes

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner	400 KB/sec
Digital camcorder	3.5 MB/sec
802.11g Wireless	6.75 MB/sec
52x CD-ROM	7.8 MB/sec
Fast Ethernet	12.5 MB/sec
Compact flash card	40 MB/sec
FireWire (IEEE 1394)	50 MB/sec
USB 2.0	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
SATA disk drive	300 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec

I/O systems in OS

■ I/O system in the OS should

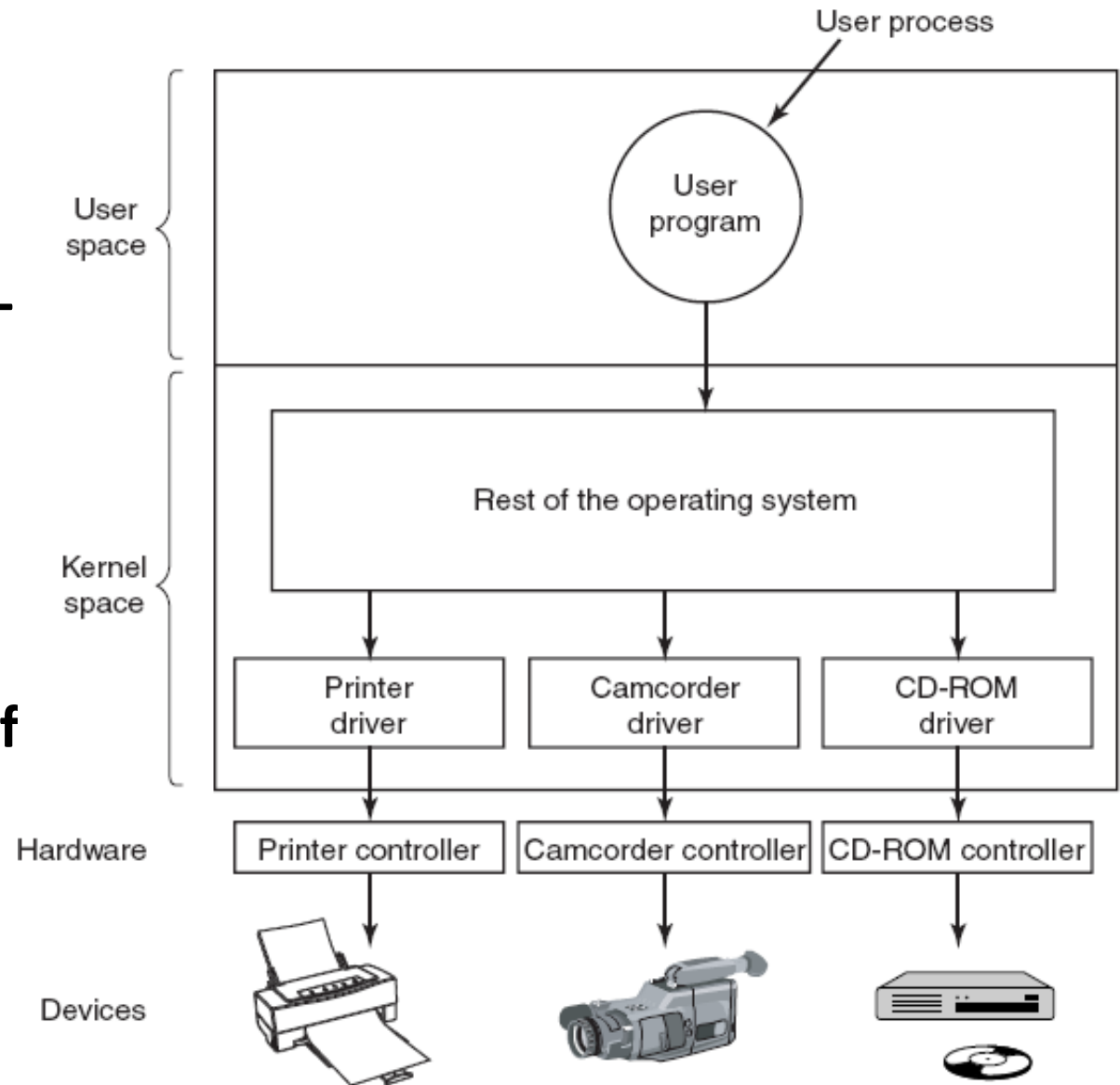
- abstract away the detailed differences in I/O devices by identifying a few general types,
- Provide access to each type through a standardized set of functions—an interface
- Provide a modular software structure to support vendor-specific software

I/O system in OS

- **Users should not be allowed to issue illegal I/O instructions.**
- **All I/O instructions should be privileged to provide a proper protection.**
- **Note that the kernel cannot simply deny all user access.**
 - Most graphics games and video editing/playback software need direct access to memory-mapped graphics controller memory to speed access.
 - The kernel may provide a locking mechanism to allow a section of graphics memory to be allocated to a process at a time.

I/O system

- Converts the I/O request of the application into low-level commands for the device and send it to the device controller,
- Take the response of the I/O device and send it to the application.



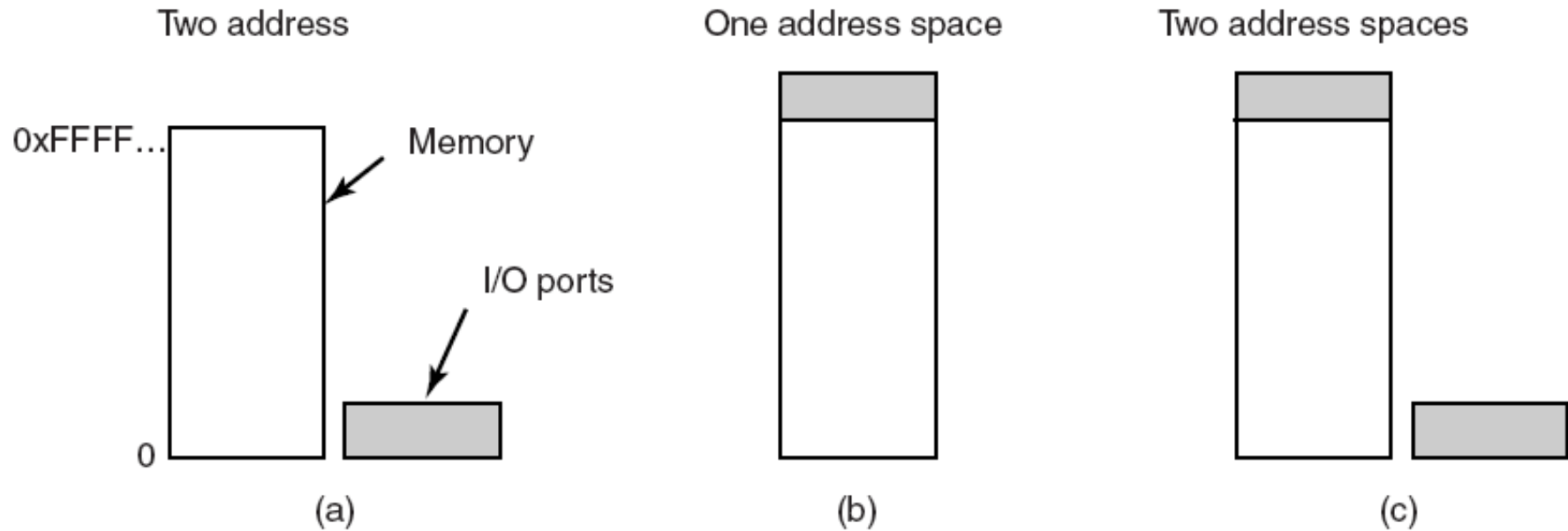
Issues related to I/O system

- How to access I/O devices in HW?
- How to interact with I/O devices?
- How are I/O devices categorized?
 - Character vs. Block

How to access I/O devices in HW?

- OS needs to send/receive commands and control to device controller to accomplish I/O.
- Device controller* has one or more registers for control and data. (*will be described later)
- Processor communicates controller through reading/writing to these registers
- How to address these registers?
 - Memory-based I/O
 - Port-based I/O
 - Hybrid I/O

Memory-mapped/ Port-based/Hybrid I/O



■ How to read/write registers:

- a) Special CPU instructions (IN/OUT)
- b) Memory mapped: Regions of memory is reserved for HW I/O registers. Standard memory instructions update them.
- c) Hybrid: Some controllers mapped to memory, some uses I/O instructions

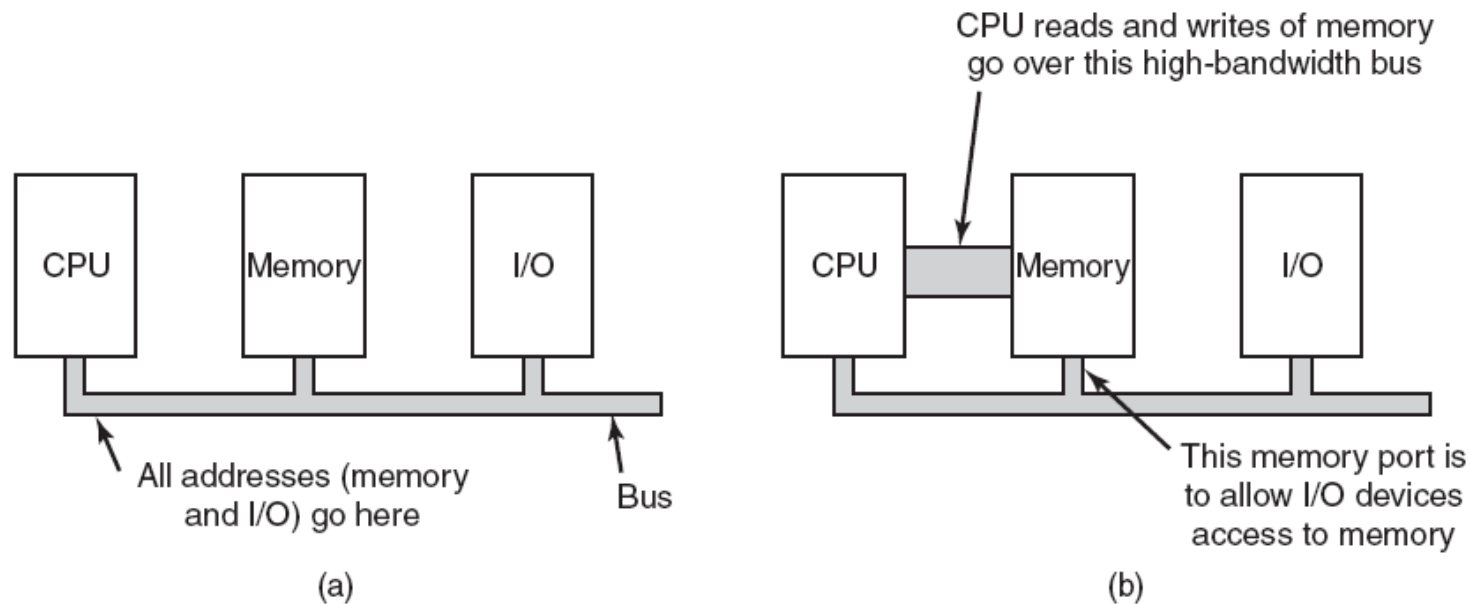
Memory-mapped IO on Intel Architecture

One address space



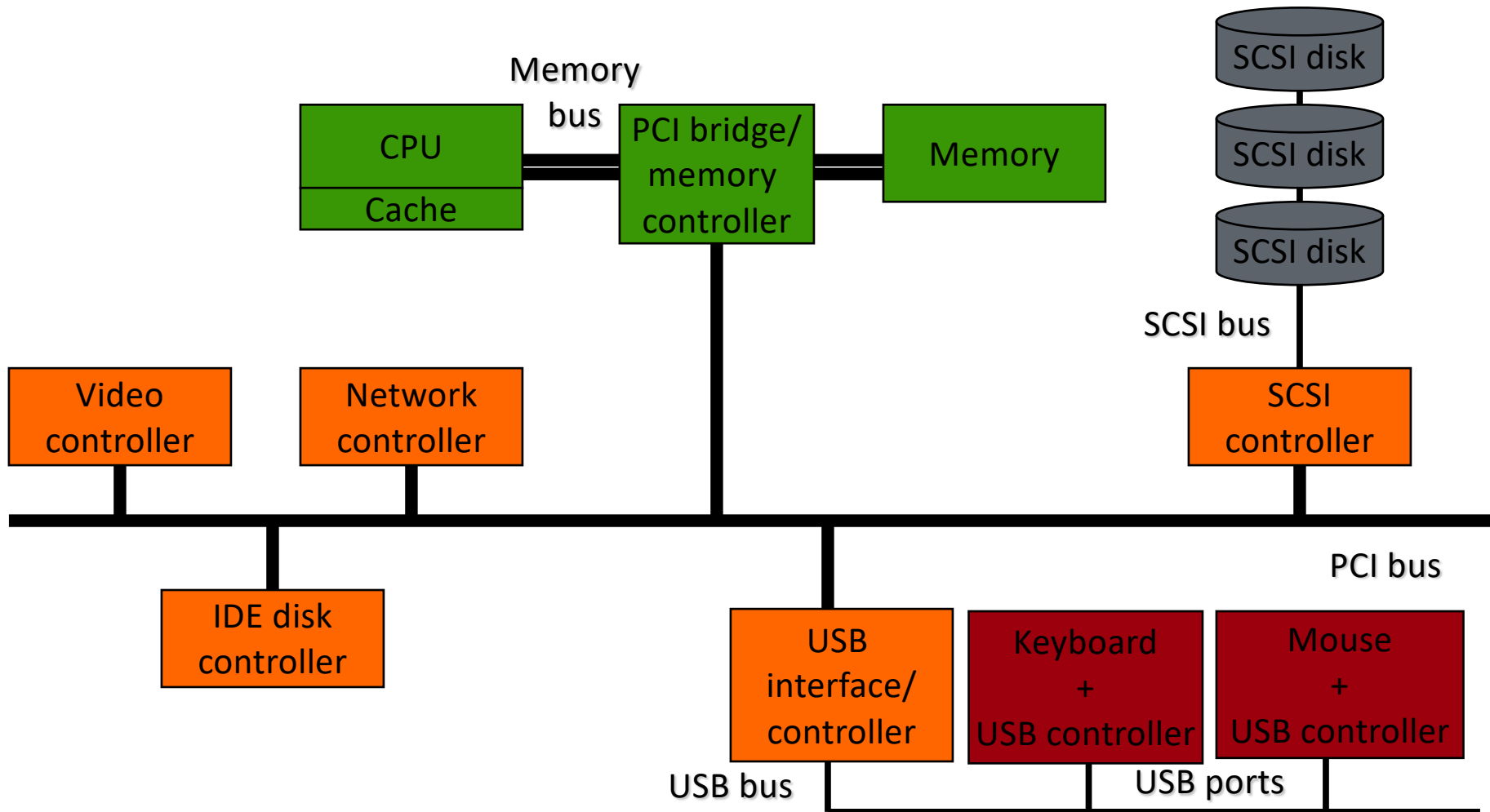
I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Single Bus and dual bus I/O



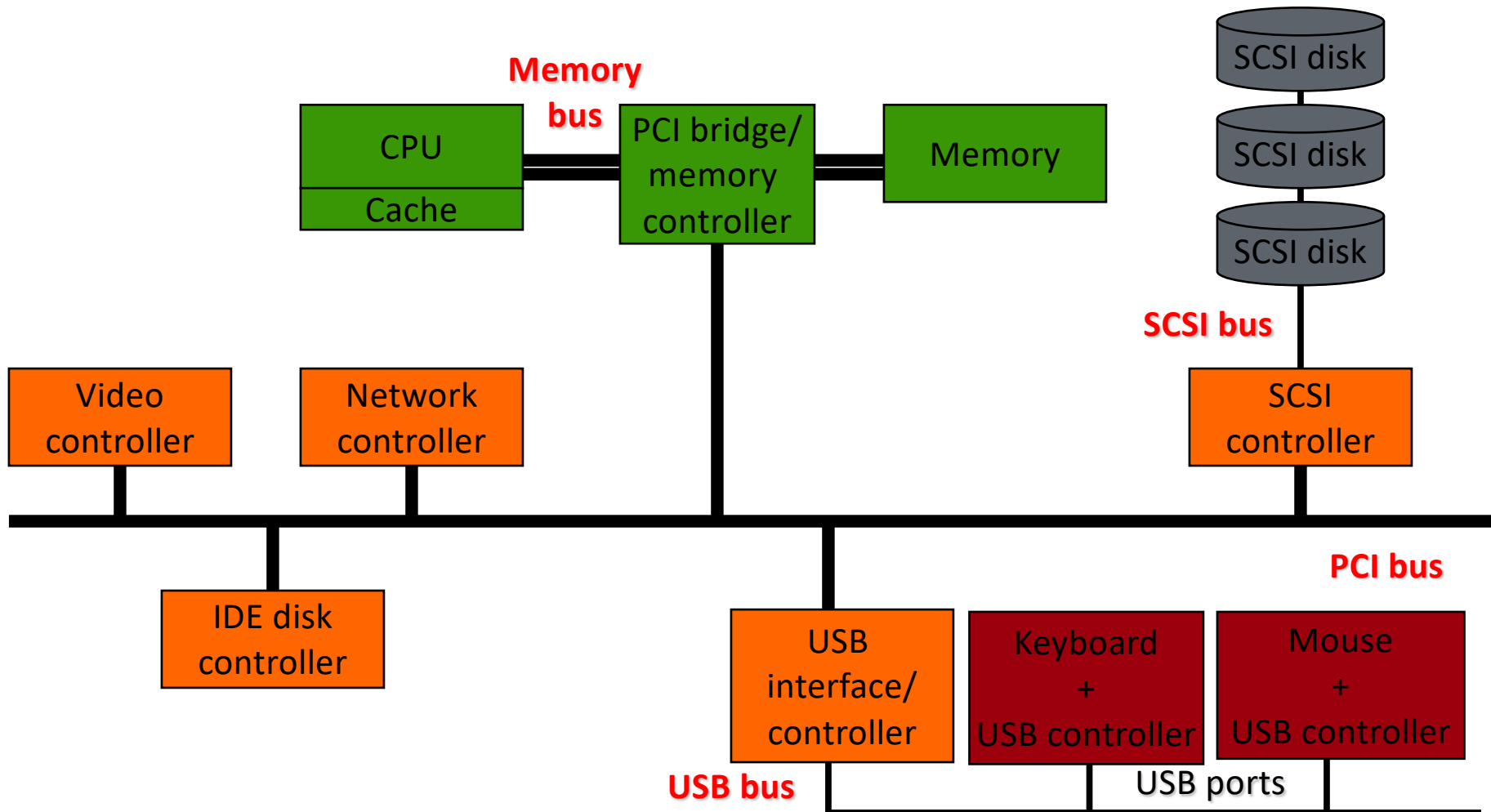
- **Memory mapped I/O has a single address space,**
 - Memory mapped I/O is simpler to implement and use.
 - Frame buffers, or similar devices, are more suitable for memory mapped I/O.
- **Port based I/O has two address spaces: one for memory, one for ports.**
 - Dual bus allows parallel read/write of data and devices.

I/O Hardware interfaces

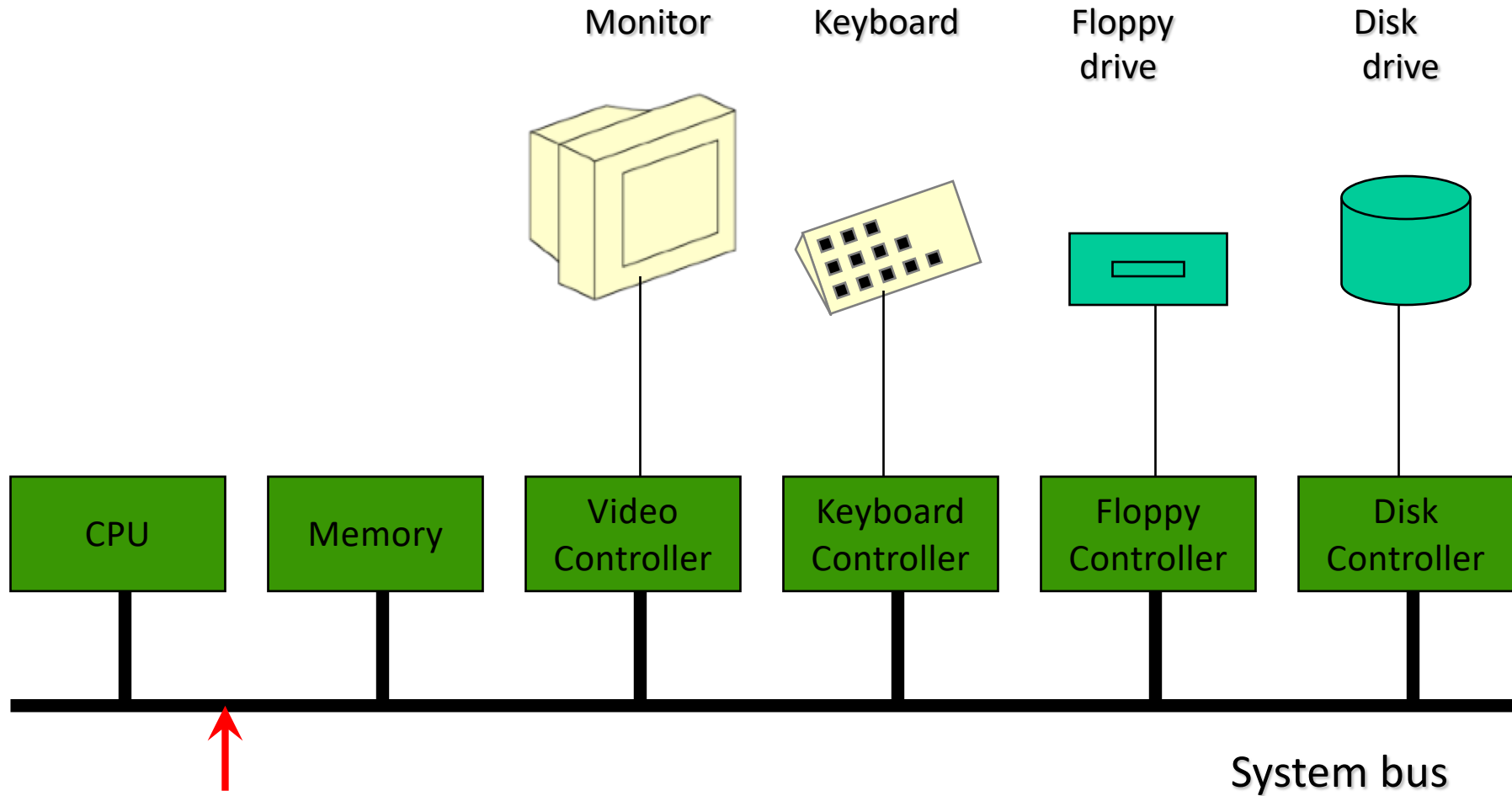


I/O Hardware interfaces - Bus

- **Bus:** An interconnection between components (including CPU)
 - More than one device can be connected



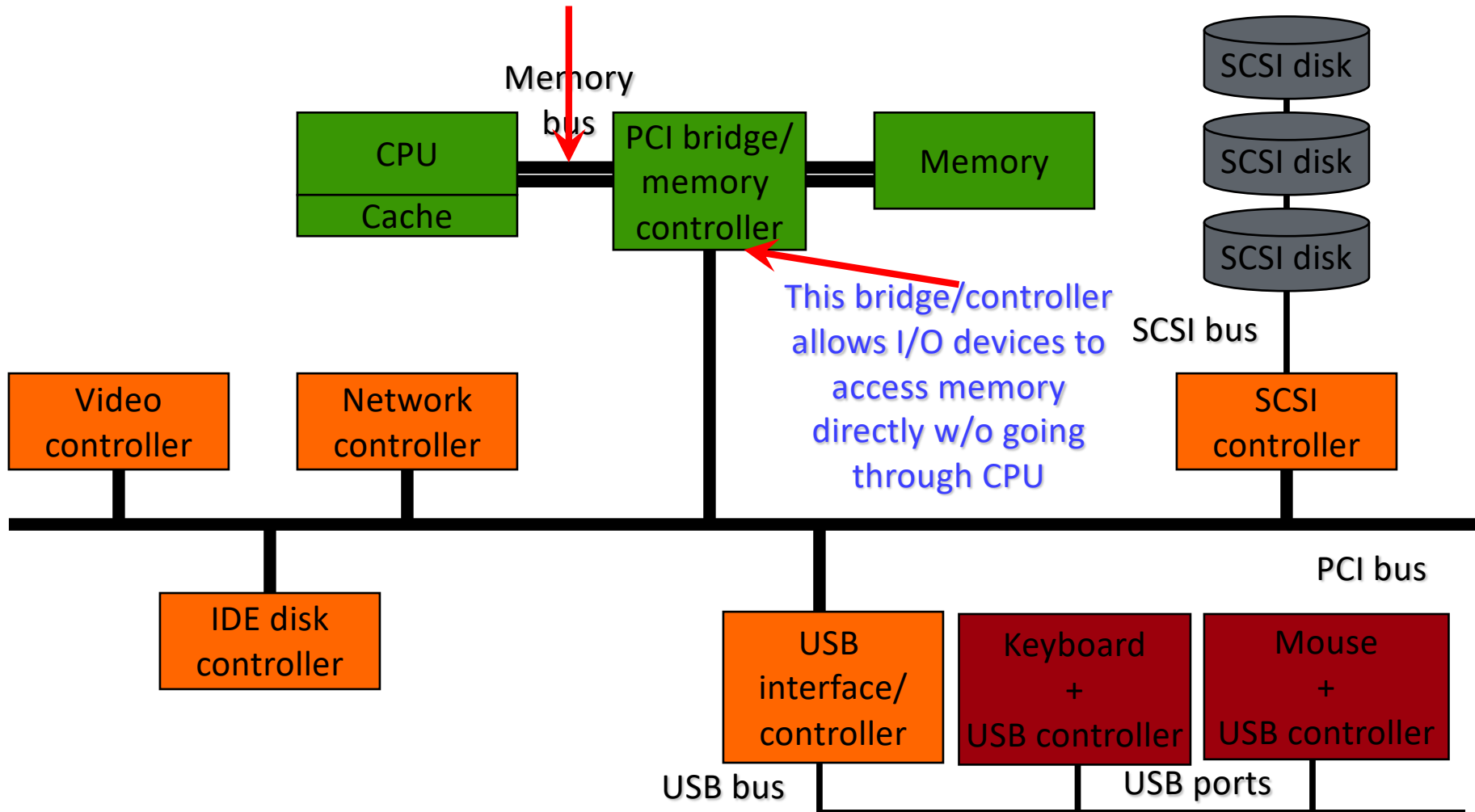
I/O Hardware - Single Bus



All addresses (memory and I/O) go here.
Memory is just another I/O.

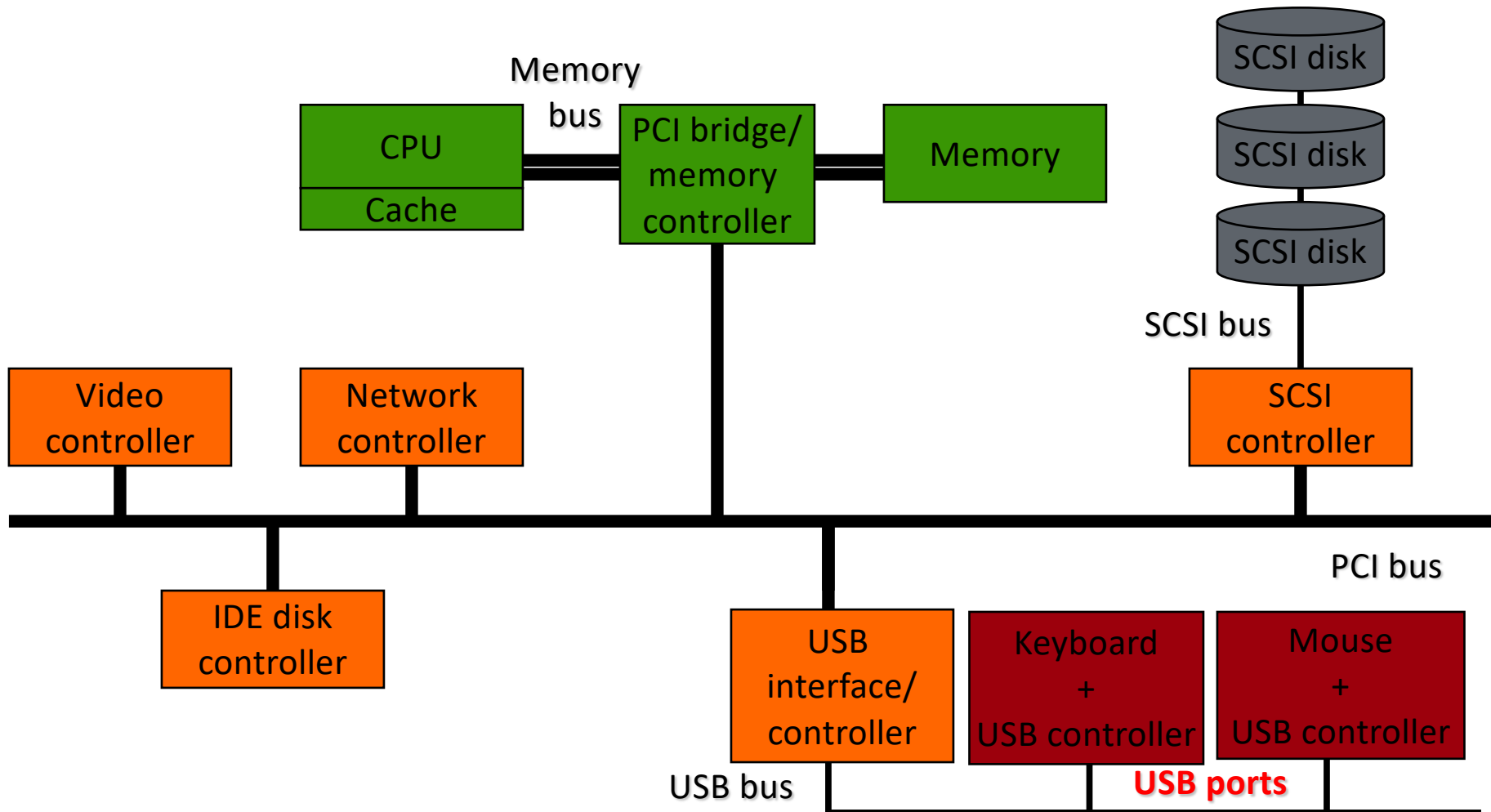
I/O Hardware – Dual bus

CPU reads and writes to the memory takes place through this high-speed bus.



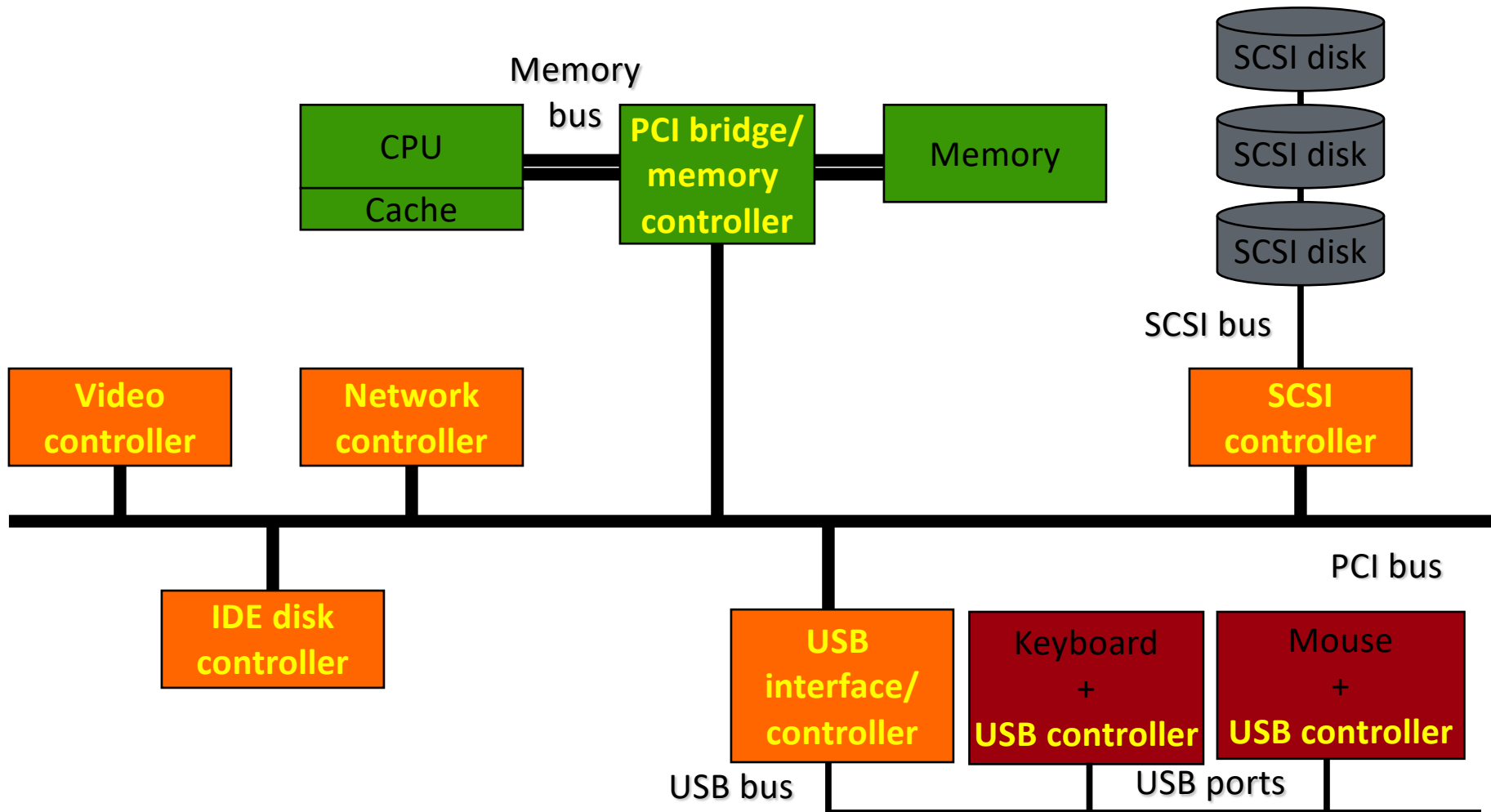
I/O Hardware interfaces - Port

- **Port:** An interface for plugging in only one I/O device



I/O Hardware interfaces - Device Controller

- **Device controller:** Connects physical device to system bus/port.



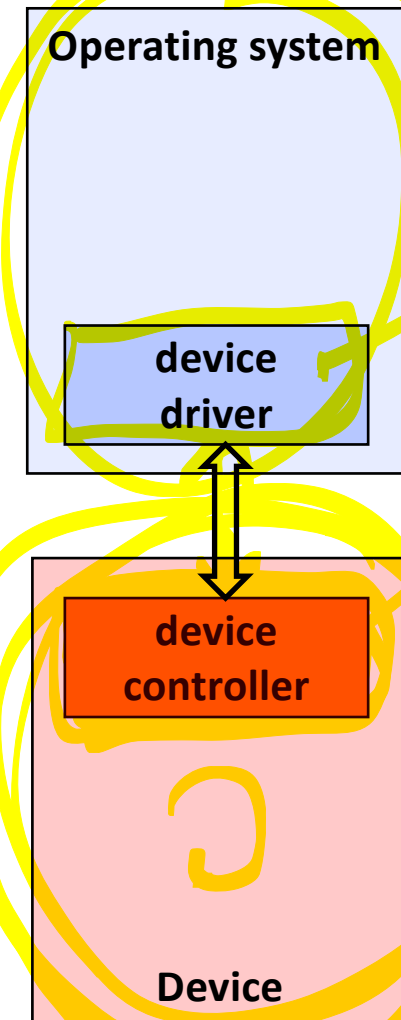
Issues related to I/O system

- **How to access I/O devices in HW?**
 - Device controllers and device drivers
- **How to interact with I/O devices?**
 - **Poll based vs. Interrupt based I/O**
 - CPU checks if I/O is complete
 - An interrupt is generated when I/O is complete
 - **Programmed vs. DMA based I/O**
 - Data is transferred to/from CPU
 - DMA controller transfers data from device buffer to main memory without CPU intervention
- **How are I/O devices categorized?**
 - **Character vs. Block**
 - Streams of chars (e.g. printer, modem)
 - Units of blocks (e.g. disks)

I/O in OS

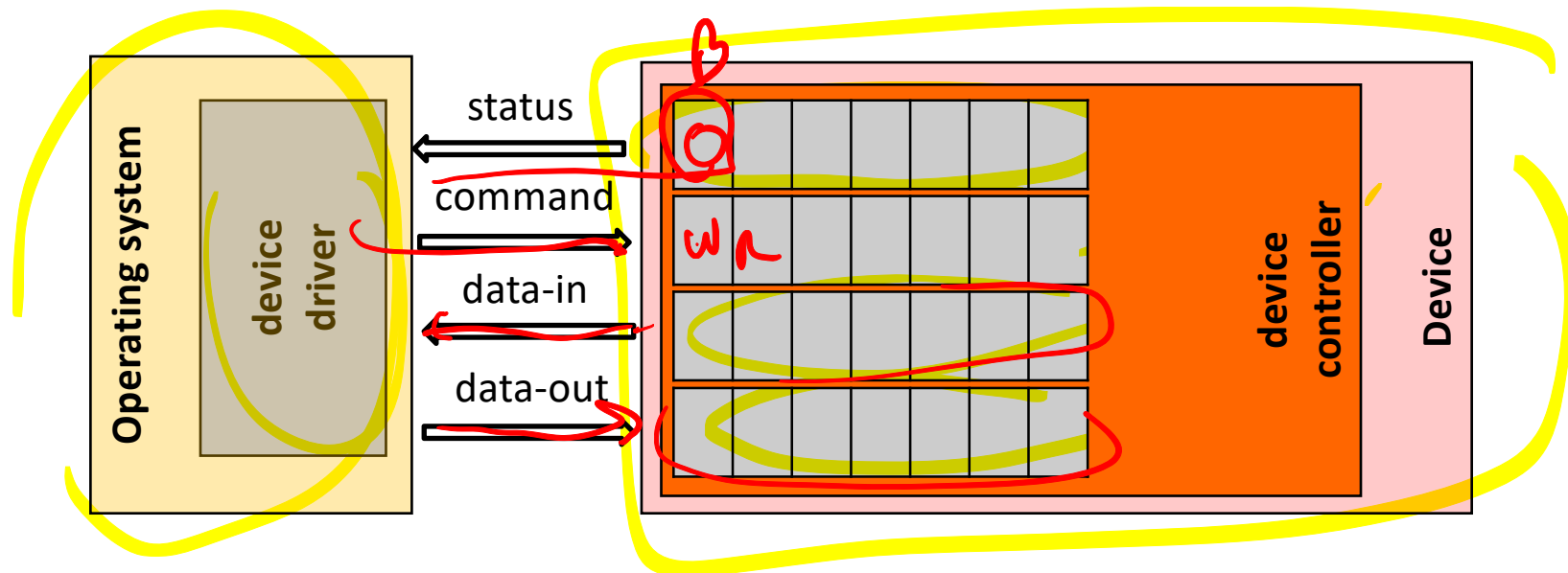
Device controllers and device drivers

- There is always a **device controller** and a **device driver** for each device to communicate with the OS.
- **Device drivers** are software modules that can be plugged into an OS to handle a particular device.
- **Device controllers** works as an interface between a device and a device driver.
 - A device controller may be able to handle multiple devices.
 - As an interface its main task is to convert serial bit stream to block of bytes, perform error correction as necessary.



I/O port registers

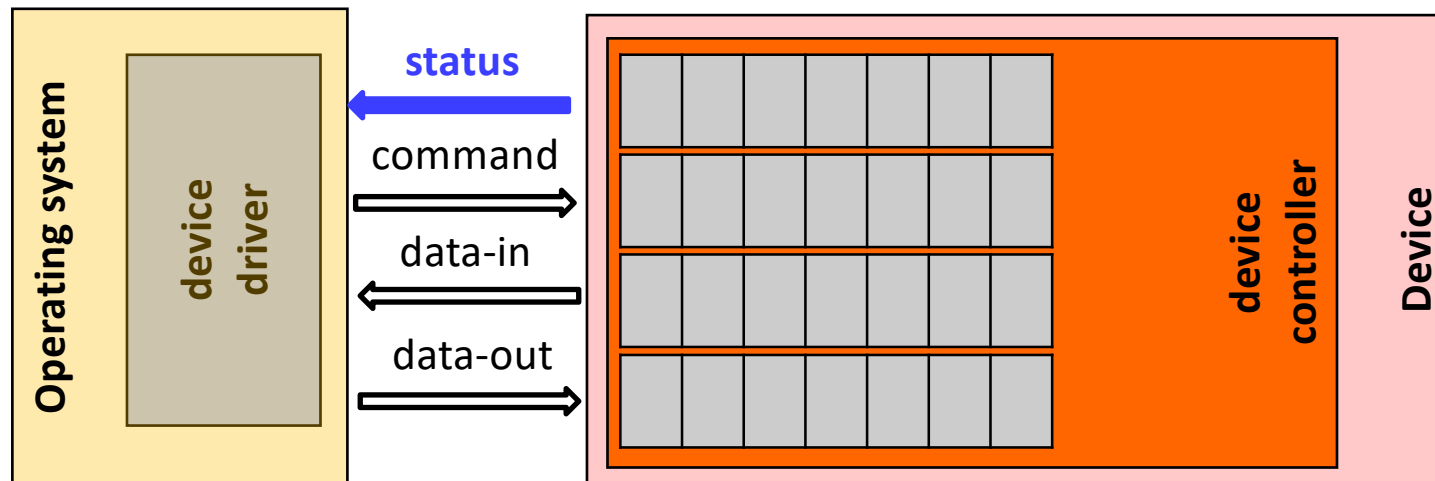
- **status register**
 - Read by the host.
- **command register**
 - Written by the host
- **data-in register**
 - Read by the host to get input.
- **data-out register**
 - Written by the host to send output.



I/O port registers - status register

■ status register

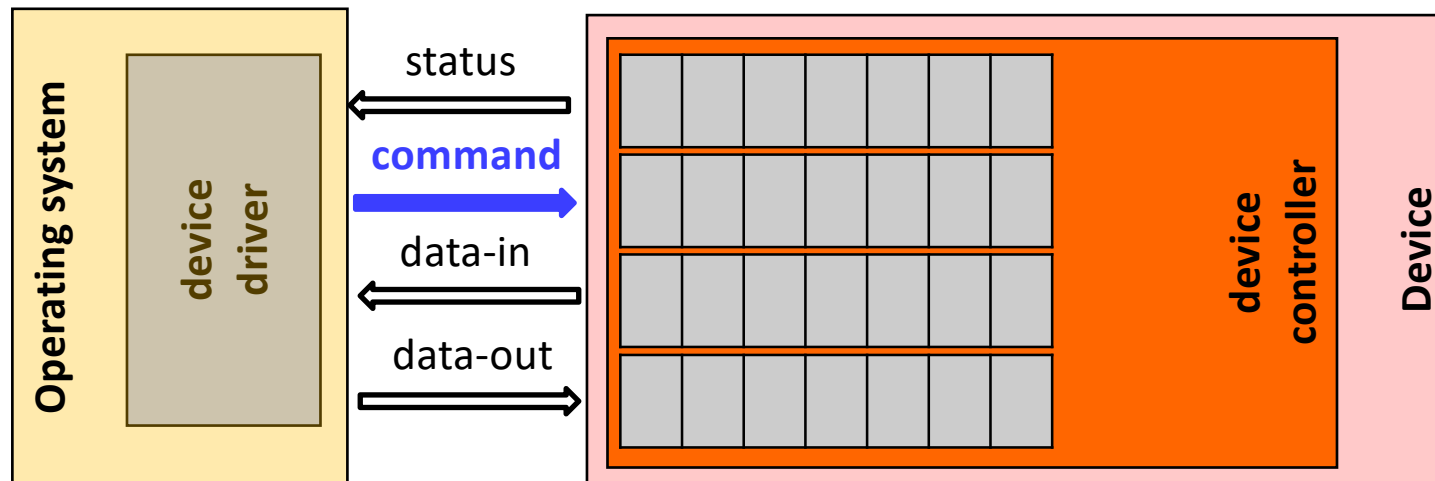
- Read by the host.
- Bits indicate states such as
 - whether the current command has completed,
 - whether a byte is available to be read from the data-in register,
 - whether there has been a device error.



I/O port registers - command register

■ command register

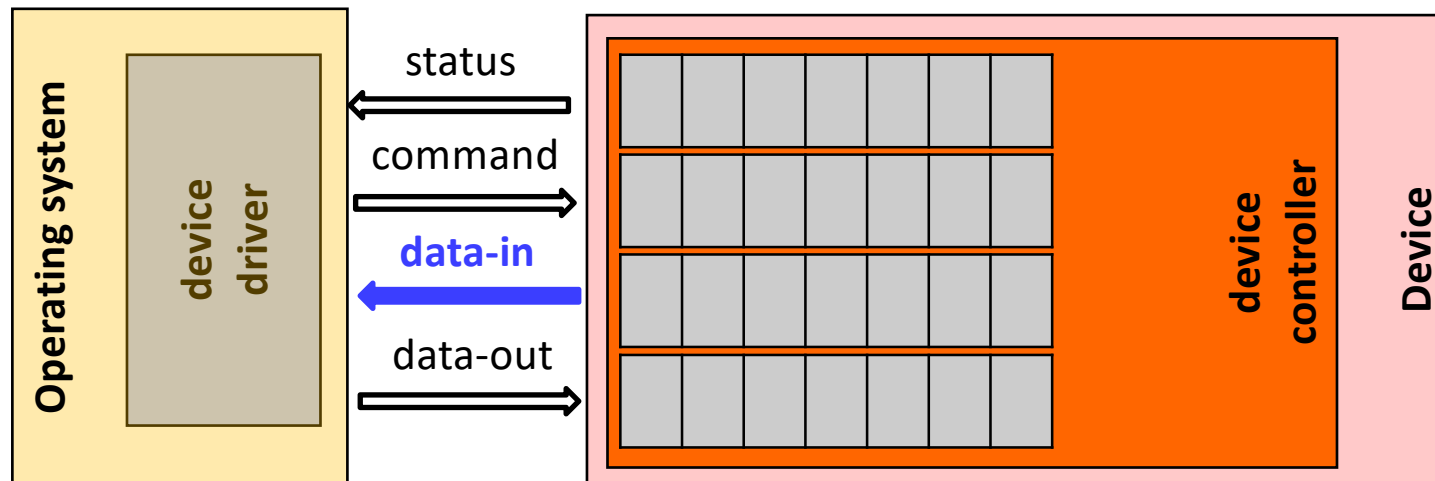
- Written by the host
 - The command requested from the device
 - E.g. read or write for a disk



I/O port registers - data-in register

■ data-in register

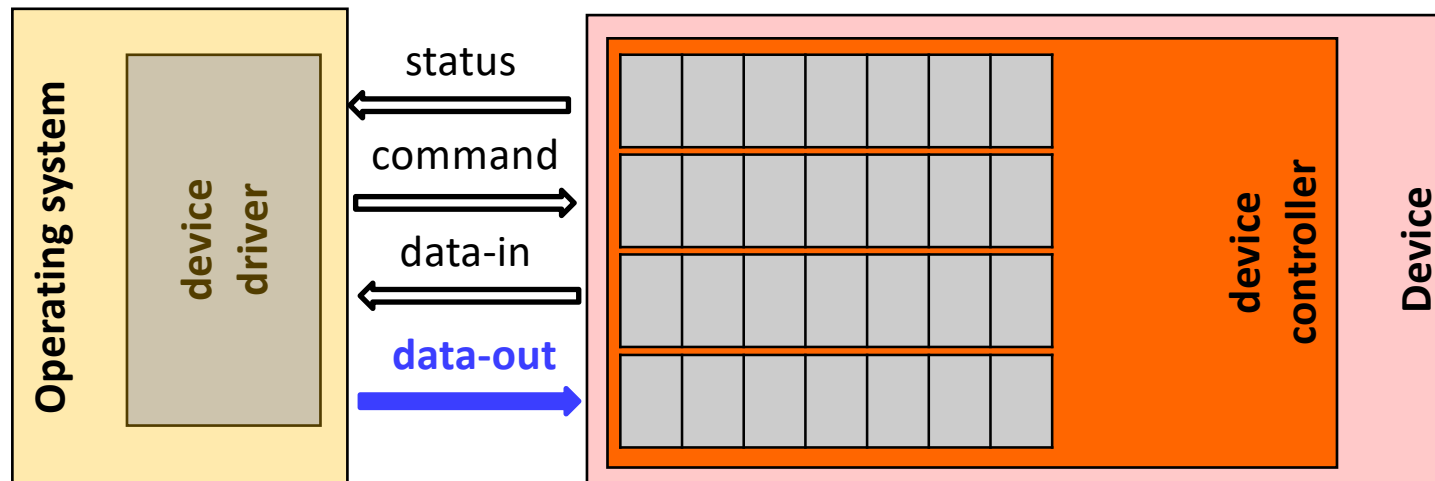
- Read by the host to get input.
 - E.g. the data read from the disk when the command is read



I/O port registers - data-out register

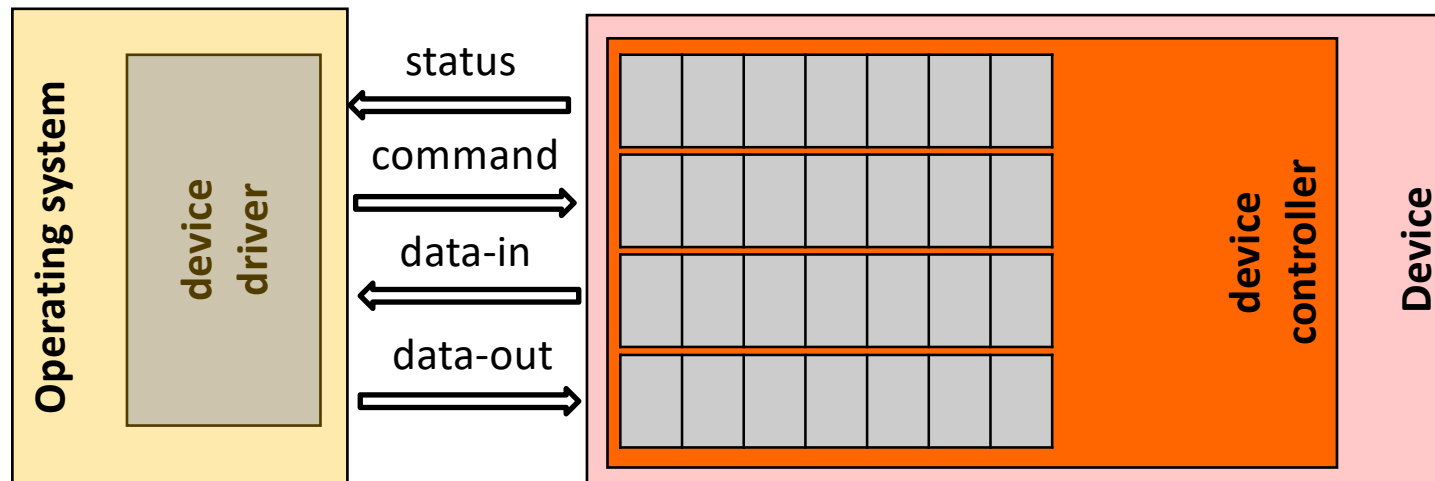
- **data-out register**

- Written by the host to send output.
 - E.g. the data to be written on the the disk when the command is write



I/O device communication

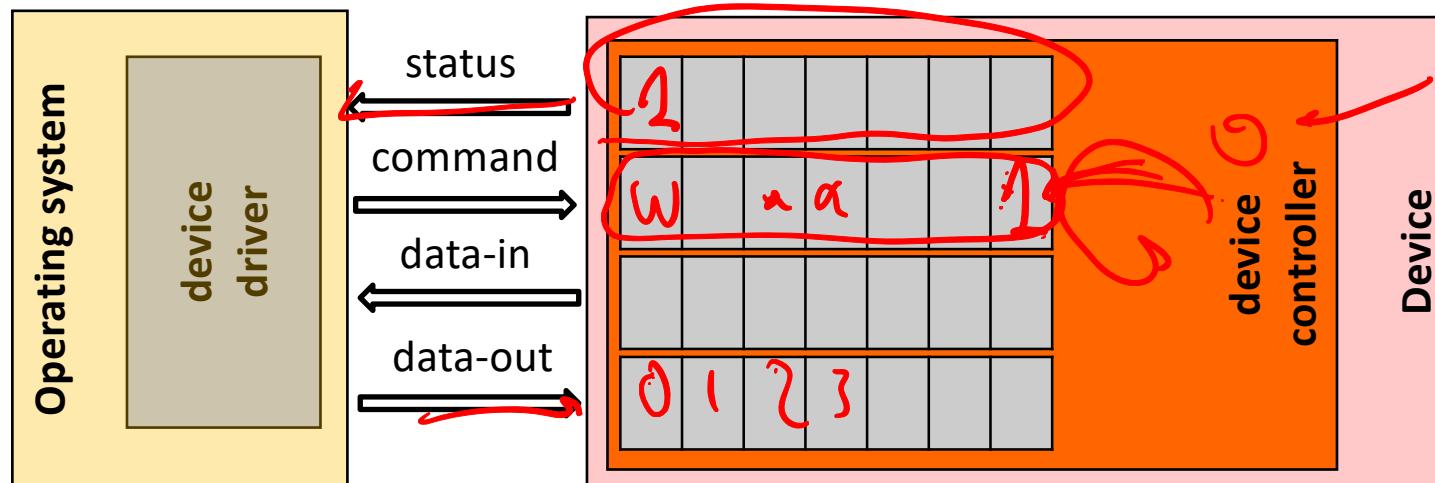
- Note that both the device driver and the device controller are running as “separate processes” to access the registers.
- Hence need to ensure the atomicity of register updates.
 - For example, how can the device controller know that the data to be written on the disk is fully copied onto its registers?



Polling: I/O interfacing in software

```
/* DEVICE DRIVER CODE */  
while (*deviceStatus & BUSY); /* POLL: repeatedly check the busy bit */  
*deviceDataOut = data_byte; /* write a byte into the data-out register */  
*deviceCommand |= WRITE; /* sets the command as WRITE */  
*deviceCommand |= READY; /* sets the command-ready bit */
```

```
/* DEVICE CONTROLLER CODE */  
while(TRUE){  
    while (*deviceCommand & READY); /* repeatedly check the command ready bit */  
    *deviceStatus = BUSY; /* set the busy bit */  
    /* ..... */  
}
```



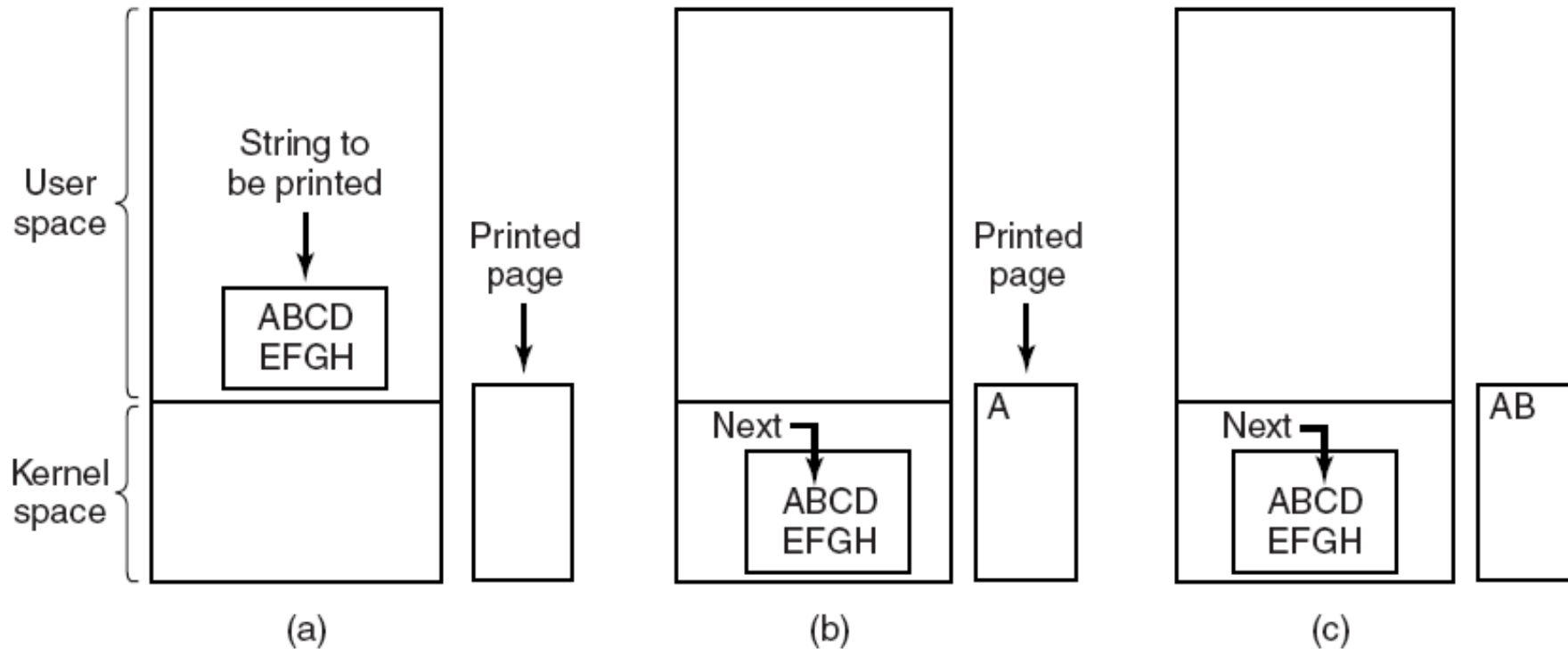
Polling: I/O interfacing in software

```
/* DEVICE DRIVER CODE */
while (*deviceStatus&BUSY); /* repeatedly check the busy bit in status */
*deviceDataOut = data_byte; /* write a byte into the data-out register */
*deviceCommand |= WRITE; /* sets the command as WRITE */
*deviceCommand |= READY; /* sets the command-ready bit */
```

```
/* DEVICE CONTROLLER CODE */
while(TRUE){
    while (*deviceCommand&READY); /* repeatedly check the command ready bit*/
    *deviceStatus |= BUSY; /* set the busy bit */

    Command = *deviceCommand; /* read the command */
    if (Command&WRITE) /* if the command is WRITE */
        dataOut = *deviceDataOut; /* read the data put by the OS */
        success= WriteToDevice(dataOut); /* do the I/O on the device */
        if (success){
            *deviceCommand &= !READY; /* clear the command-ready bit */
            *deviceStatus &= !ERROR; /* clear the error bit */
            *deviceStatus &= !BUSY; /* clear the busy bit */
        }else{
            *deviceCommand &= !READY; /* clear the command-ready bit */
            *deviceStatus |= ERROR; /* SET the error bit */
            *deviceStatus &= !BUSY; /* clear the busy bit */
        }
    }
}
/* code for other commands */
}
```

Polling I/O example: Steps in printing a string



- a) Copy the string to be printed into a buffer in the kernel space**
- b) Poll the printer and send a character if not busy.**
- c) Loop until the end of the string.**

Programmed Polling I/O example: Pseudocode for printing a string

```
/* p is the kernel buffer */
copyFromUser(buffer, p, count);

/* loop on every character */
for (i=0; i<count; i++){
    /* loop until device is ready */
    while (*printerStatusRegister != READY);
        /* POLLING/BUSY WAITING! */

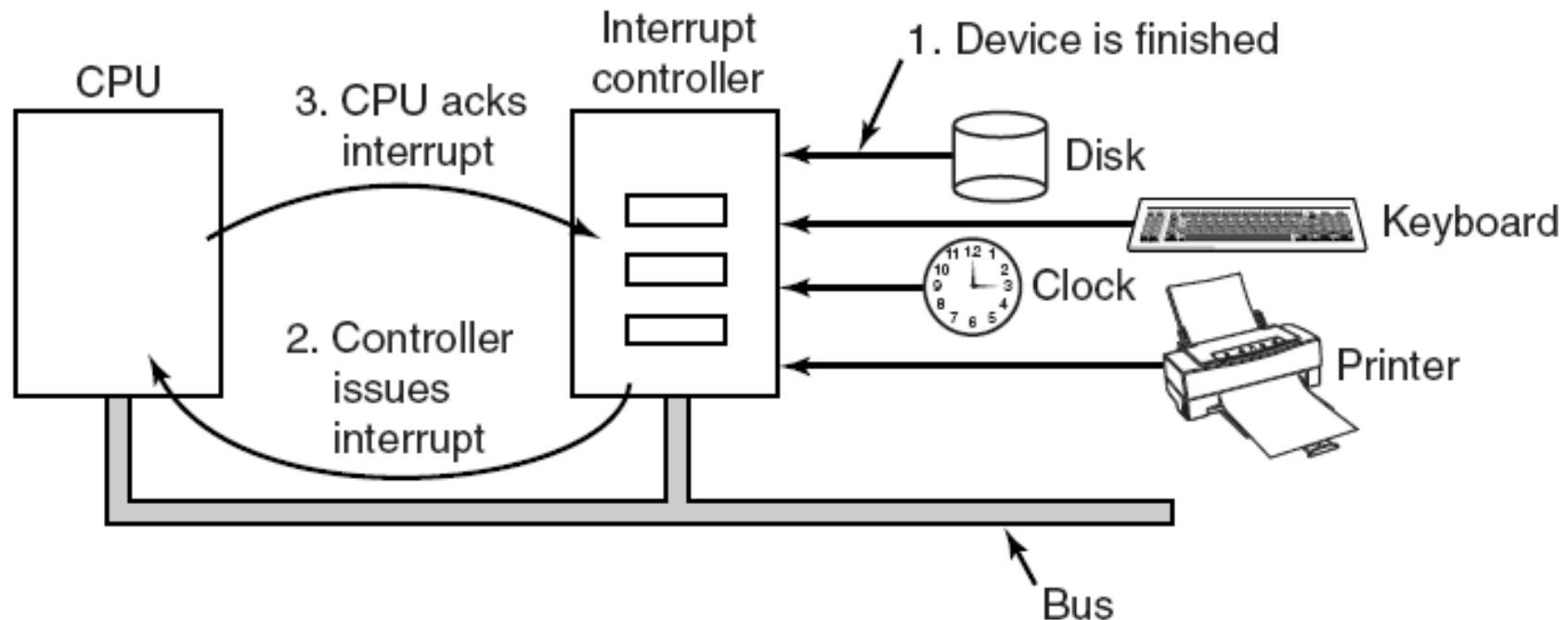
    /* output one char */
    *printerDataRegister = p[i];
}

returnToUser();
```

- **Polling is essentially busy waiting and wastes CPU time!**
 - Any ideas how to fix it?

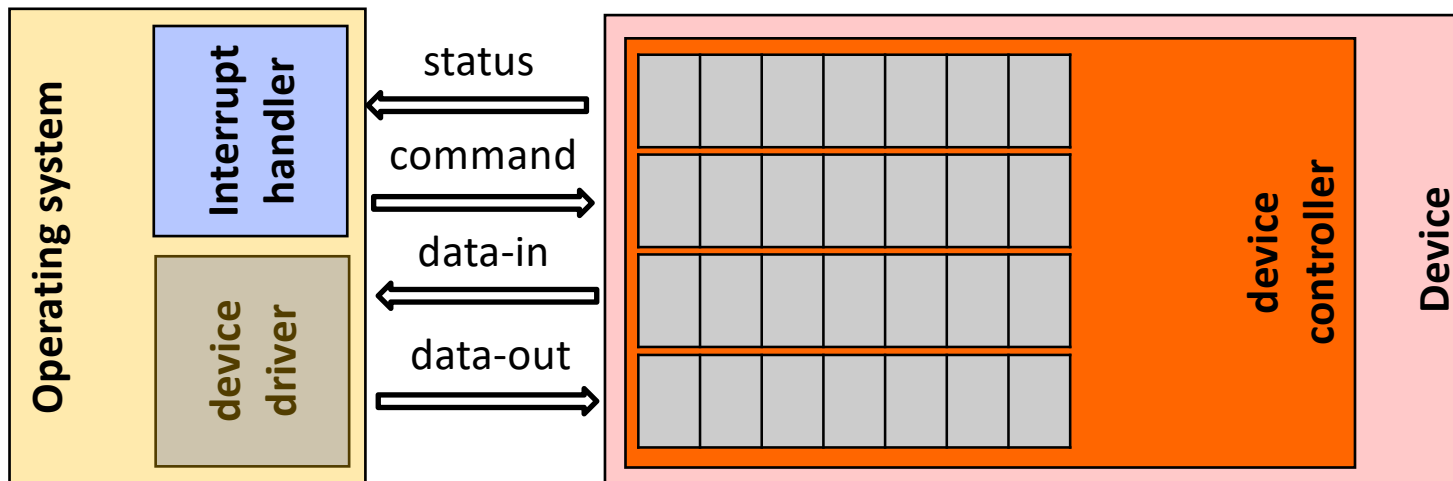
Interrupts - refresher

- **Interrupts are hardware exceptions.**
- **CPU has an interrupt wire**
 - Connected to an interrupt controller, which in turn is
 - connected to I/O devices
- **When one of the devices generate an interrupt signal, the controller informs the CPU**
 - The CPU acknowledges the interrupt and
 - Jumps to the interrupt service routine (ISR) if needed.



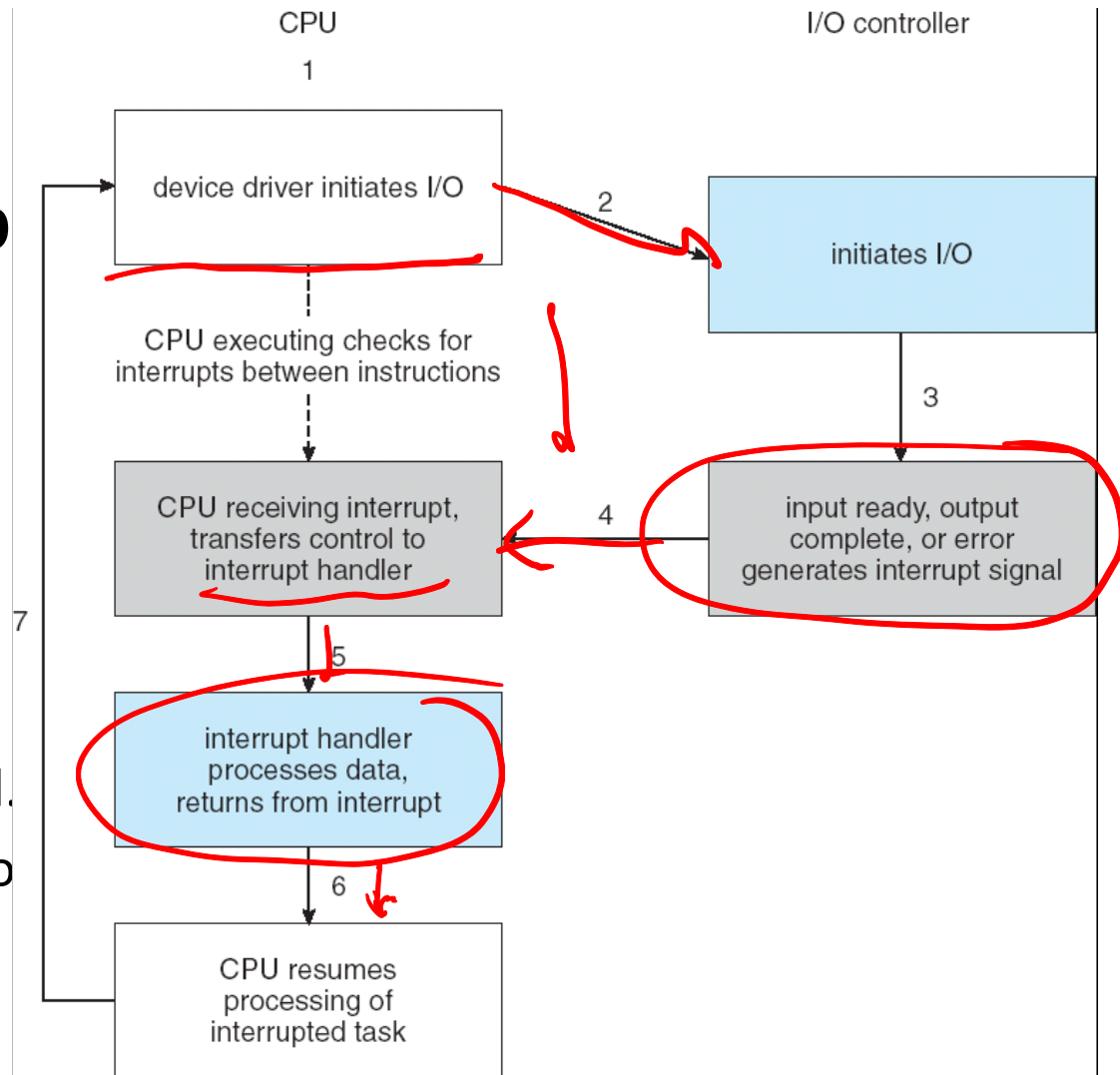
Interrupt-Driven I/O

- **Use interrupts!**
 - Recall that interrupts are “hardware exceptions” that allow I/O devices to signal the CPU that they need attention!
- **The device driver initiates the I/O and resume (instead of busy waiting)**
- **When done, the device raises an interrupt to let the CPU (hence OS) know that it’s ready to accept more**
- **The interrupt service routine (handler) sends some more.**



Interrupt Based I/O

- CPU is not blocked during I/O Schedules user tasks.
- Upon interrupt:
 - Current CPU state is saved
 - Interrupt Service Routine corresponding to device is jumped.
 - Necessary actions are executed.
 - Return from Interrupt instruction restores the state prior to the interrupt.



- Depending on the architecture a separate stack can be used as ISR context
- Task requesting I/O is put into sleep until interrupt handler marks I/O ready and wakes up the task.

Interrupt-Driven I/O

```
/* Code executed when the print system call is made */
```

```
copyFromUser(buffer, p, count);  
enableInterrupts();  
while (*printerStatusRegister != READY);  
*printerDataRegister = p[i];  
scheduler();
```

```
/* Interrupt Service Routine (ISR) for the printer */
```

```
if (count == 0)  
    unblockUser();  
else{  
    *printerDataRegister = p[i];  
    count = count - 1;  
    i++;  
}  
acknowledgeInterrupt();  
returnFromInterrupt();
```

Interrupt servicing: Advanced

- **Modern interrupt controllers provide**
 - The ability to defer interrupt handling during critical processing
 - Efficient way to dispatch the proper interrupt handle w/o polling all the devices
 - Multi-level interrupts, to distinguish low and high priority interrupts
- **Most CPU's have two interrupt request lines**
 - Nonmaskable
 - Reserved for unrecoverable errors
 - Maskable
 - Used by device controllers
 - Can be turned off before the execution of critical instruction sequences

Interrupt servicing: Advanced (cont)

■ Interrupt mechanism needs

- Address: to select a specific interrupt handling routine
 - Typically an offset in a table called interrupt vector which contains addresses of interrupt handlers

■ What if there are more devices than the interrupt vector size?

- Interrupt chaining

■ Interrupt priority levels

- defer the handling of low-priority interrupts without masking off all interrupts,
- and makes it possible for a high-priority interrupt to pre-empt the execution of a low-priority interrupt.

Direct Memory Access (DMA)



- For a device that does large transfers, such as a disk drive,
 - it seems wasteful to use an expensive general-purpose processor to watch status bits and
 - to feed data into a controller register **1 byte at a time** —a process termed programmed I/O
- Interrupt-based I/O is not a remedy since, each byte would create a context switch to the Interrupt Handler Routine.
- In both polling-based and interrupt-based I/O, all the bytes need to be passed through the CPU and has a lot of overhead
 - I/O device <-> CPU <-> Memory
- It would be nice if we can off-load this mundane task to a special-purpose processor that can move the data from/to I/O device to memory directly!
 - Direct-memory-access (DMA) controller.

Direct Memory Access

- To initiate a DMA transfer, the host writes a DMA command block into memory.
 - a pointer to the source of a transfer,
 - a pointer to the destination of the transfer, and
 - a count of the number of bytes to be transferred.
- The CPU writes the address of this command block to the DMA controller, then goes on with other work.
- The DMA controller proceeds to operate the memory bus directly,
 - placing addresses on the bus to perform transfers without the help of the main CPU.
 - A simple DMA controller is a standard component in PCs, and bus-mastering I/O boards for the PC usually contain their own high-speed DMA hardware

I/O Using DMA

```
/* Code executed when the print system call is made */
```

```
copyFromUser(buffer, p, count);  
setupDMAController();  
scheduler();
```

End of
break

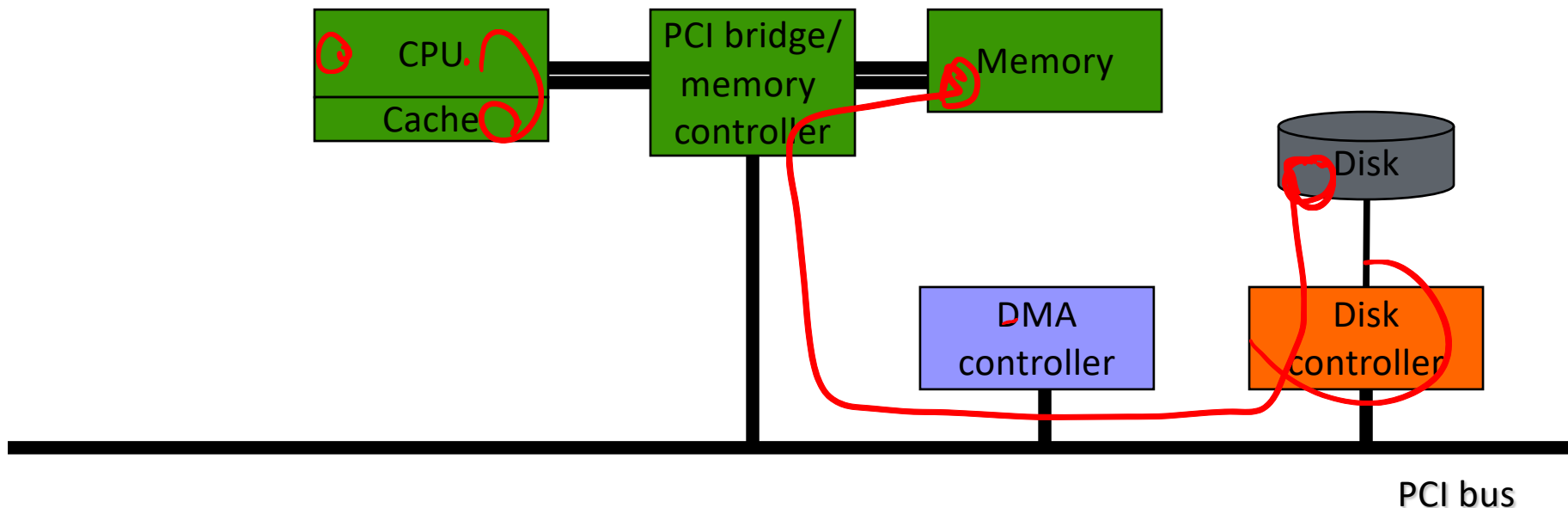
```
/* Interrupt Service Routine Procedure for the printer */
```

```
acknowledgeInterrupt();  
unblockUser();  
returnFromInterrupt();
```

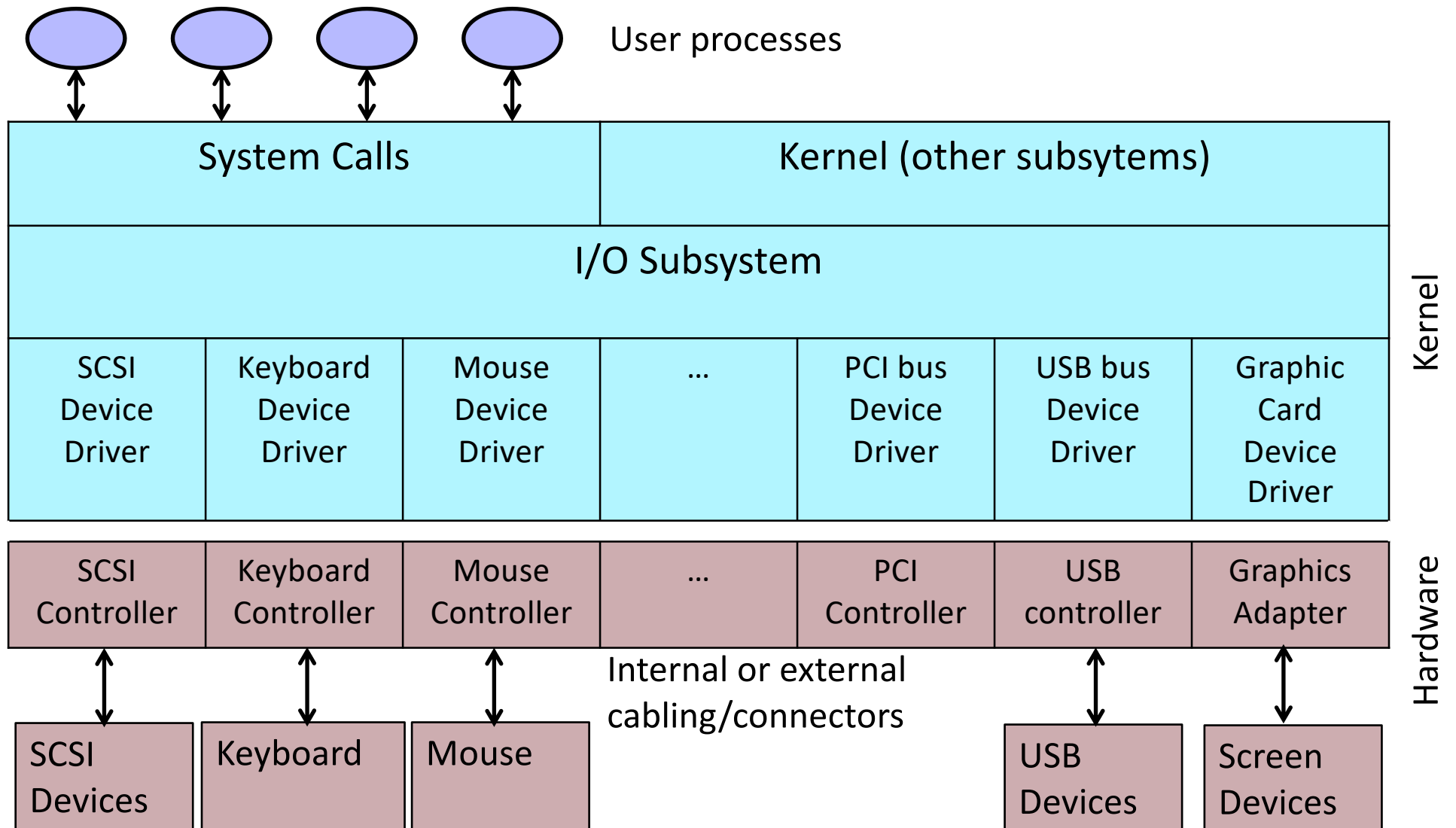
- Note that the interrupt is generated once per I/O task as opposed to once per byte (in the case of interrupt-based I/O).

I/O Hardware interfaces

- Device driver is told to transfer disk data to buffer at address X
- Device driver tells the disk controller to transfer C bytes from disk to buffer at address X
- Disk controller initiates DMA transfer
- Disk controller sends each byte to DMA controller
- DMA controller transfers bytes to buffer X
 - Incrementing memory address and decrementing C until 0.
- When $C == 0$, DMA interrupts CPU to signal transfer completion.



Application I/O Interface



Application I/O Interface

- **OS needs to provide the interface and I/O subsystem from user area down to the HW and the device controller.**
- **Interface and I/O subsystem should:**
 - Cover all different device types
 - E.g. graphic cards, network interface cards, disk controllers, HCI devices, etc.
 - Allow addition of new devices.
 - A vendor introduced a new product, support it.
 - Provide device driver development interfaces for HW vendors.
 - Plug and play support for different device types and buses (i.e. PCI, USB).
 - Dynamic loading of device drivers.
 - A kernel supporting all possible HW will be huge.
 - Load device drivers on demand or selectively.

Application I/O Interface

- **System calls are the basic kernel interface for user applications.**
- **A different set of system calls for each different:**
 - Device vendor?
 - Device driver?
 - Device type?
 - Device class?
- **Set of system calls should be minimum. A uniform and simple I/O interface is required.**
- **Simple set of device types:**
 - Character devices (character special files)
 - Block devices (block special files)
 - Network devices (socket interface)
 - Special hardware (graphics/GPU)

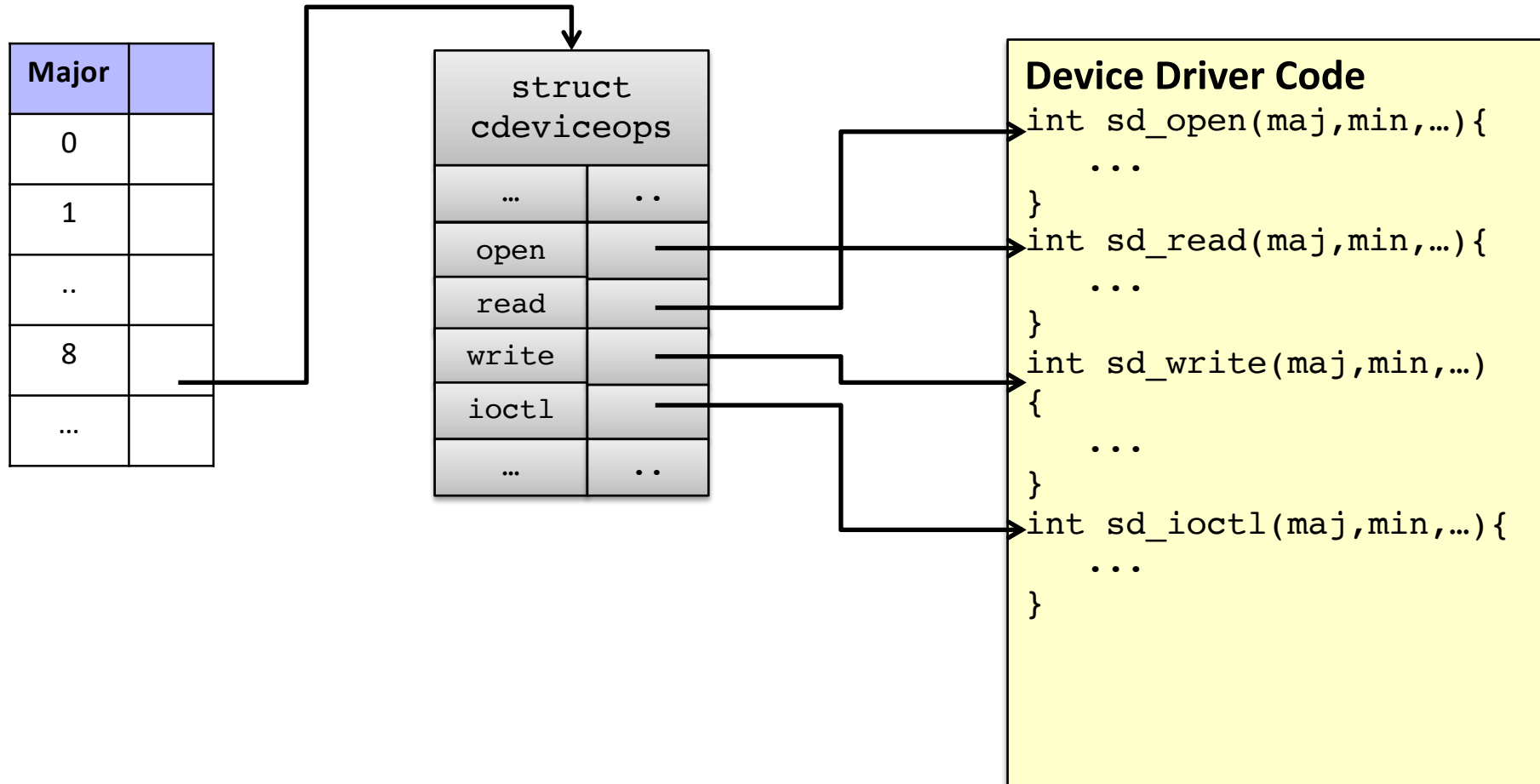
Application I/O Interface

- **Unix solution: Use simple file interface:**
 - `open/read/write/seek/close/mmap + ioctl` (for configuring device)
- **Handler/Entry point for a device:**
 - a **special file** on file system which resides traditionally under `/dev`
- **All system calls on special files are directed on device drivers.**

```
crw----- 1 root root 10, 1 Mar 21 13:43 /dev/psaux
brw-rw---- 1 root disk 8, 16 Mar 21 13:43 /dev/sdb
brw-rw---- 1 root disk 8, 17 Mar 21 13:43 /dev/sdb1
brw-rw---- 1 root disk 8, 18 Mar 21 13:43 /dev/sdb2
brw-rw---- 1 root disk 8, 19 Mar 21 13:44 /dev/sdb3
crw--w---- 1 root tty 4, 0 Mar 21 13:43 /dev/tty0
crw----- 1 onur tty 4, 3 May 22 13:35 /dev/tty3
```

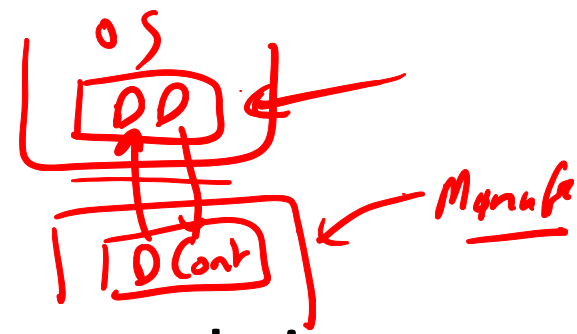
- **A special file is either a character or a block device.**
- **Each file has a major and minor number**
 - Major number selects device driver code.
 - Minor number selects between multiple devices handled by same driver.

Device Driver Switch



A read on block special file major 8, minor 4 translates:
`bdevsw[8] -> read(8, 4, ...)`

Device drivers



- Each I/O device attached to a computer needs some device-specific code for controlling it.
 - written by device manufacturer
 - each OS needs its own device drivers
- Each device driver supports a specific type or class of I/O devices.
 - A mouse driver can support different types of mice but cannot be used for a webcam.
- The OS defines what a driver does and how it interacts with the rest of the OS.
- A device driver has several functions
 - to accept abstract read and write requests from device-independent software above it and make sure that they are carried out, +
 - initialization of the device
 - manage is power requirements and log events

Device driver structure

Disk: $0 \dots NCM$
R, W

- **check whether input parameters are valid**
- **translation from abstract to concrete terms**
 - for a disk driver, converting a block id into head, track, sector and cylinder numbers for the disk's geometry
- **check if the device is currently in use by checking its status register**
 - if not, insert the request into queue
 - if the device is not on, turn it on
- **issue the sequence of commands**
 - after issuing each command, check whether the device is ready to accept the next one
 - [in most cases] the driver blocks itself until an interrupt comes
 - [in fewer cases] the driver waits for the completion of the command
- **the driver is awakened up by the driver to continue its operation**
- **the data and the error information is passed to the device-independent OS I/O software**

Device drivers - issues

■ Note that

- an I/O device may complete while the device driver is running and create an interrupt
- the interrupt may cause the current driver to run

■ device drivers must be reentrant,

- a running driver has to expect that it will be called a second time before the first call has completed.

■ Drivers cannot make system calls but are allowed to call some kernel procedures for interaction.

I/O devices - character and block devices

- From units of transfer perspective, I/O devices can be divided into two categories:
 - **Block devices** – A block device is one with which the driver communicates by sending entire blocks of data.
 - Hard disks, USB cameras, Disk-On-Key etc.
 - **Character devices** – A character device is one with which the driver communicates by sending and receiving single characters.
 - serial ports, parallel ports, sounds cards etc.

Block devices

- The block-device interface captures all the aspects necessary for accessing disk drives and other block-oriented devices.
 - `read()` and `write()`, and, if it is a random-access device, it has a `seek()` command to specify which block to transfer next.
- Applications normally access such a device through a file-system interface.
 - The operating system itself, and special applications such as database-management systems, may prefer to access a block device as a simple linear array of blocks (also called raw I/O).
- OS device cache is used to accelerate block device operations.
 - Block based I/O is tightly coupled with paging.
- File systems require block devices.

Character devices

- A keyboard is an example of a device that is accessed through a character-stream interface.
 - `get()` or `put()` one character.
- Character device drivers implements their own buffers and caching internally.
 - for example, when a user types a backspace, the preceding character is removed from the input stream.
- Example devices:
 - Keyboard, modem, mouse

Character vs. Block Devices

- **Character and Block device switches are separate.**
- **Interface is also different.**
 - Block devices can combine read and write functions (see I/O scheduling later)
- **Character vs. Block devices:**
 - Character devices can transfer data 1 byte at a time. Block devices work in block units (i.e. 4K)
 - Character device devices have buffering and caching internal, block devices use systems page cache.
 - Block devices can contain file system partitions and swap area.
 - Block device drivers may implement I/O scheduling algorithms, system call interface support it.

ioctl()

- A single function configuration interface for all devices.
`int ioctl(int fd, unsigned long request, ...);`
- Types of requests and optional parameter is driver or even vendor specific.
- Each driver implements its own set of configuration requests and parameter types.
- Devices and drivers change but libc and kernel interface is fixed.

Windows I/O subsystem

- Microsoft Windows uses **device shortcuts on filesystem** to **address devices**.
- Device Access API provides an **interface to application programmers to inspect and interact with the devices**.
- WDK, Windows Device Framework provides user and kernel interfaces for device driver development.

I/O Categorization (OS perspective)

- **Character stream vs block.**
- **Sequential vs Random access**
device driver allow seeking to an offset in device
- **Synchronous vs Asynchronous**
I/O operation on device driver is synchronized with I/O completion on device controller. Asynchronous I/O returns earlier and report success/failure later.
- **Buffered vs Direct**
The reported operation result is completed on buffers or on device controller.
- **Shareable or Dedicated**
I/O on each device instance is mutually exclusive. (i.e. printer)
- **Read only, Write only, Read-write**
i.e: mouse, printer, hard disk

Kernel I/O System

Kernels provide many services related to I/O:

- scheduling,
- buffering,
- caching,
- pooling,
- device reservation, and
- error handling

built on the hardware and device-driver infrastructure.

I/O scheduling

- **I/O is usually slow and some devices have physical characteristics requiring optimizations.**
 - E.g. **hard disks**: Mechanical devices and **delays caused by head movement and rotation**.
 - If I/O is executed in a **FIFO strategy**, **mechanical zigzag movements** can overrule the I/O operations.
- **I/O scheduling gets a set of I/O requests on a device and determines an optimal order and timing to execute the requests on the device.**
 - Operating-system developers implement scheduling by **maintaining a queue of requests** for each device.
 - When an application issues a **blocking I/O system call**, the request is placed on the **queue** for that device.
 - The **I/O scheduler rearranges the order** of the queue to improve the overall system efficiency and the **average response time** experienced by applications.

Disk I/O Scheduling

- Given multiple outstanding I/O requests, what order to issue them?
 - Why does it matter?
- Major goals of disk scheduling:
 - Minimize latency for small transfers
 - Primarily: Avoid long seeks by ordering accesses according to disk head locality
 - Maximize throughput for large transfers
 - Large databases and scientific workloads often involve enormous files and datasets
- Note that disk block layout (where we place file blocks, directories, file system metadata, etc.) has a large impact on performance
- On modern (smart) disk drives, I/O scheduling is done by the disk controller, which executes incoming I/O requests in a out-of-order fashion.
 - More on this will be discussed in the Disk Technology slides.

Disk I/O Scheduling

■ Given multiple outstanding I/O requests, what order to issue them?

■ **FIFO:** Just schedule each I/O in the order it arrives

- What's wrong with this?
 - Potentially lots of seek time!

■ **SSTF:** Shortest seek time first

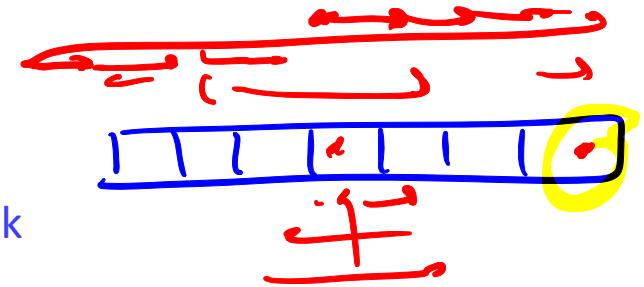
- Issue I/O with the nearest cylinder to the current one
 - Favors middle tracks: Head rarely moves to edges of disk

■ **SCAN (or Elevator) Algorithm:**

- Head has a current direction and current cylinder
- Sort I/Os according to the track # in the current direction of the head
- If no more I/Os in the current direction, reverse direction

■ **CSCAN Algorithm:**

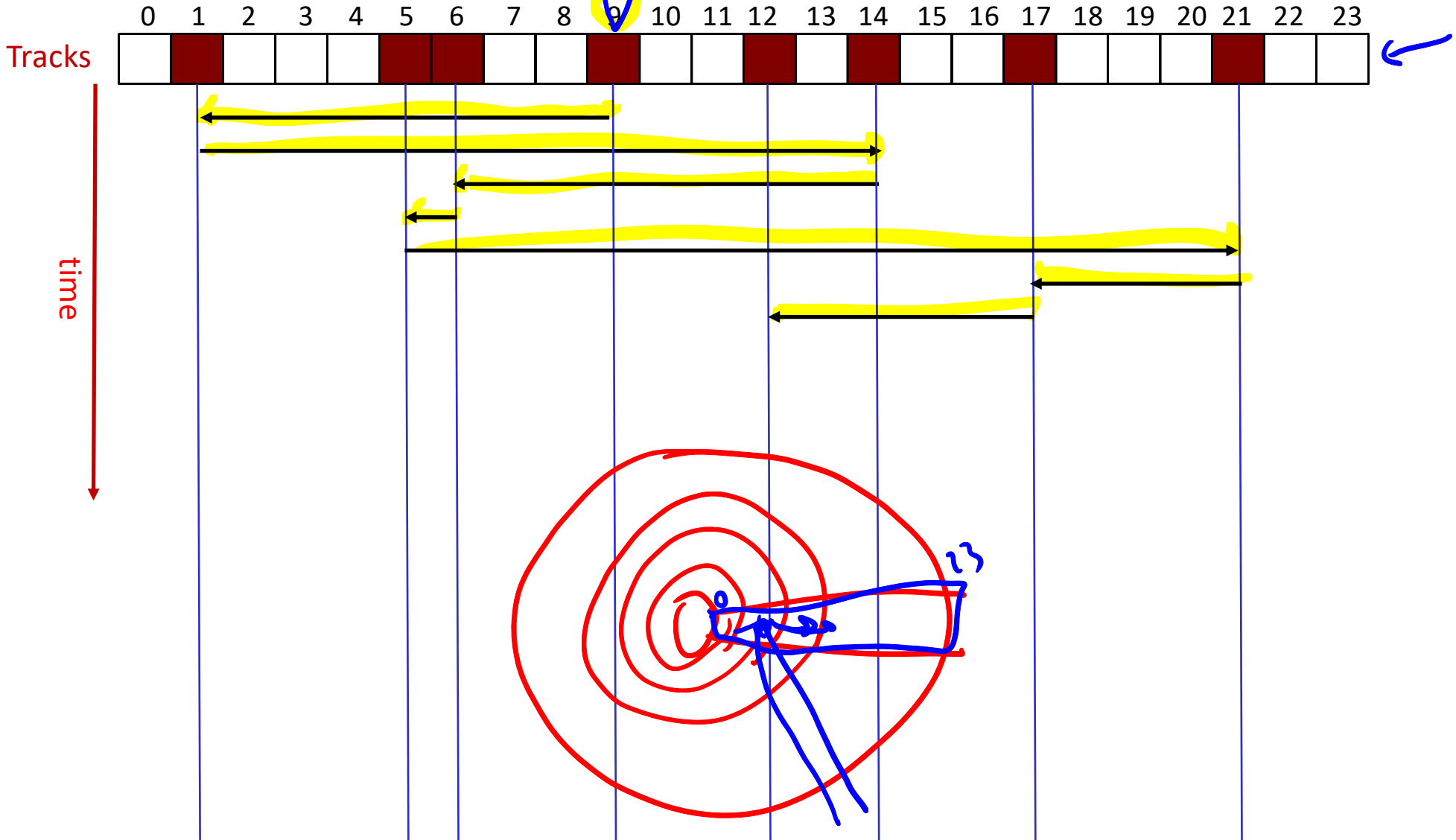
- Always move in one direction, "wrap around" to beginning of disk when moving off the end
- Reduce variance in seek times, avoid discrimination against the highest and lowest tracks



FIFO example

Current track: 9

I/O queue at t0: 9,1,14,6,5,21,17,12

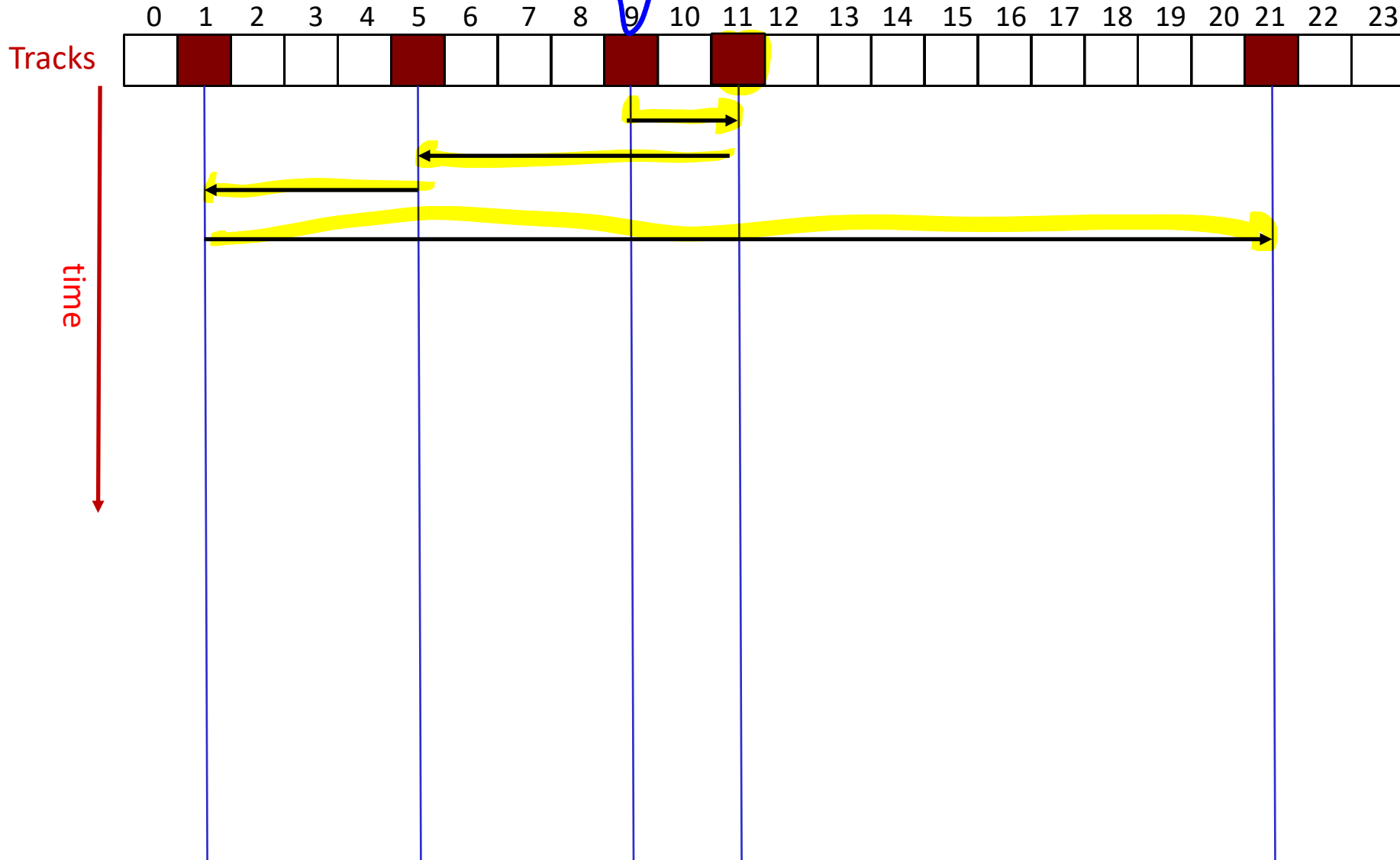


SSTF: Shortest seek time first example

Current track: 9

I/O queue at t0: 9, 1, 5, 21, 11

6, 13

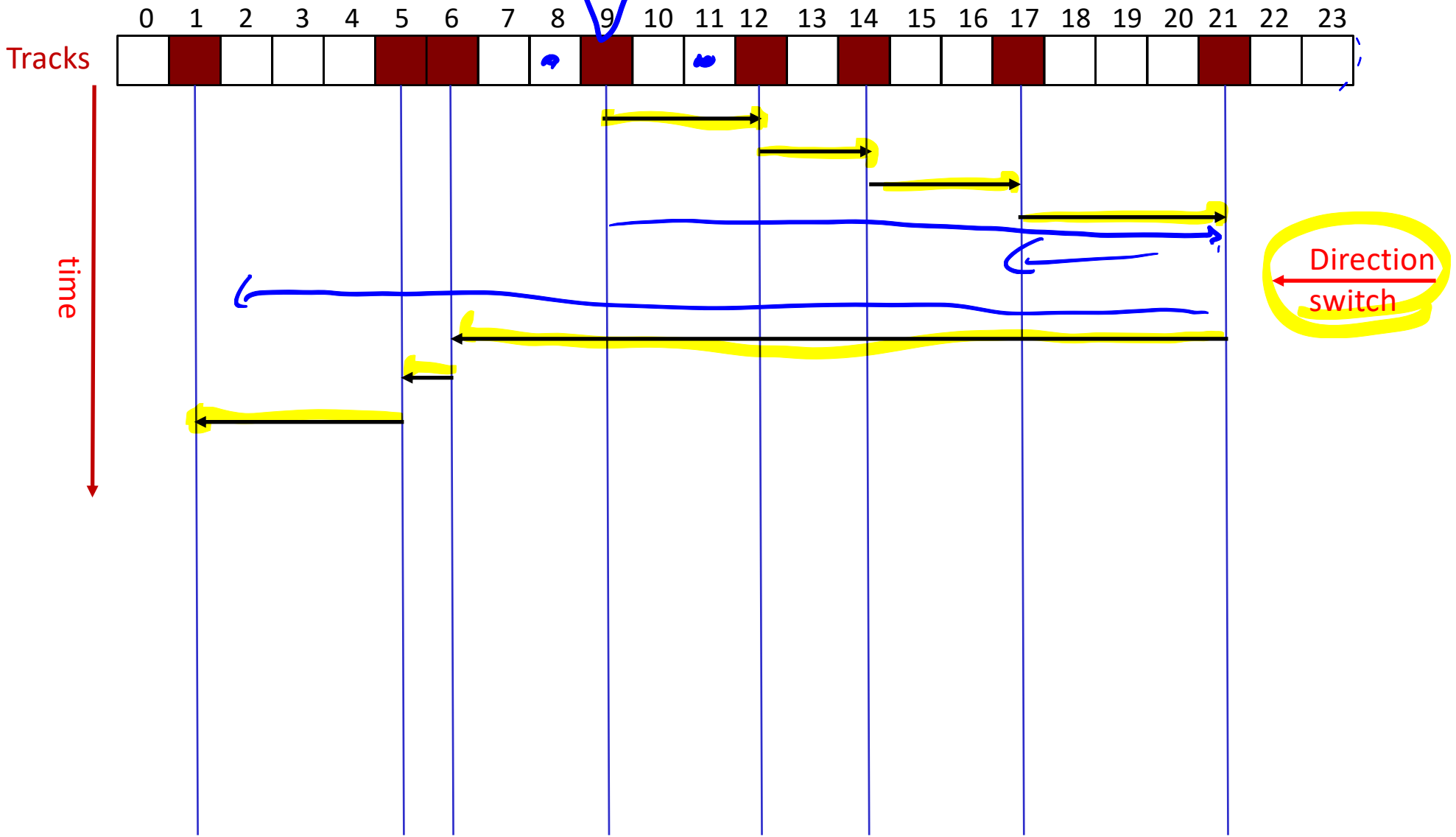


SCAN example

Elevator

Current track: 9 Current direction: →

I/O queue at t0: 9,1,14,6,5,21,17,12

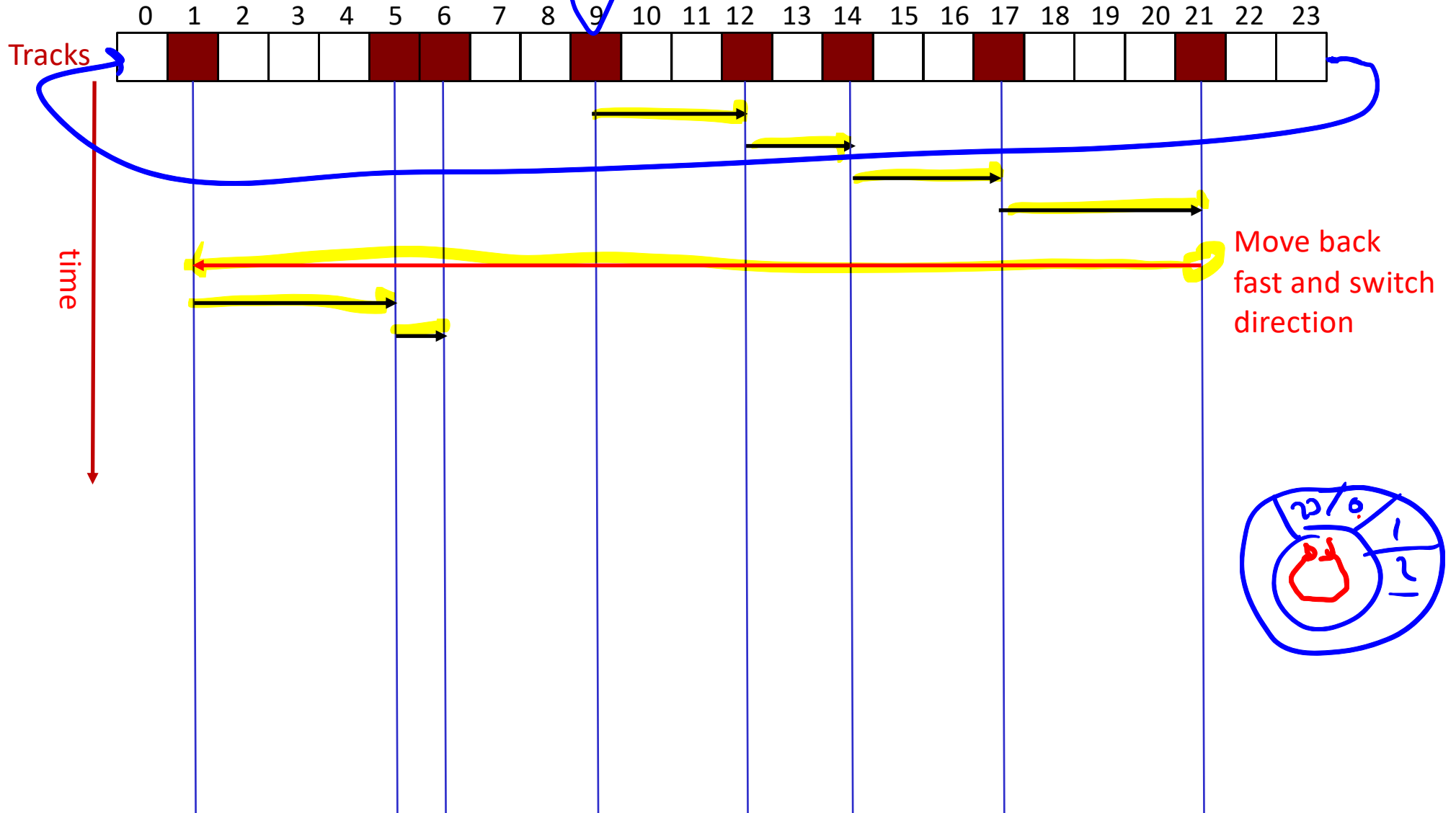


Circular

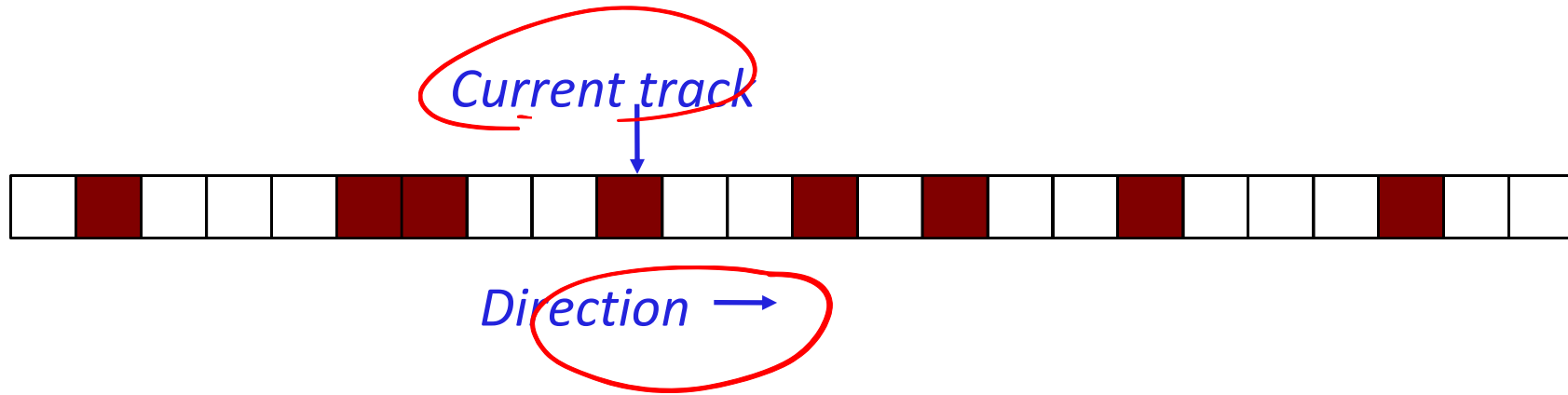
CSCAN example

Current track: 9 **Current direction:** →

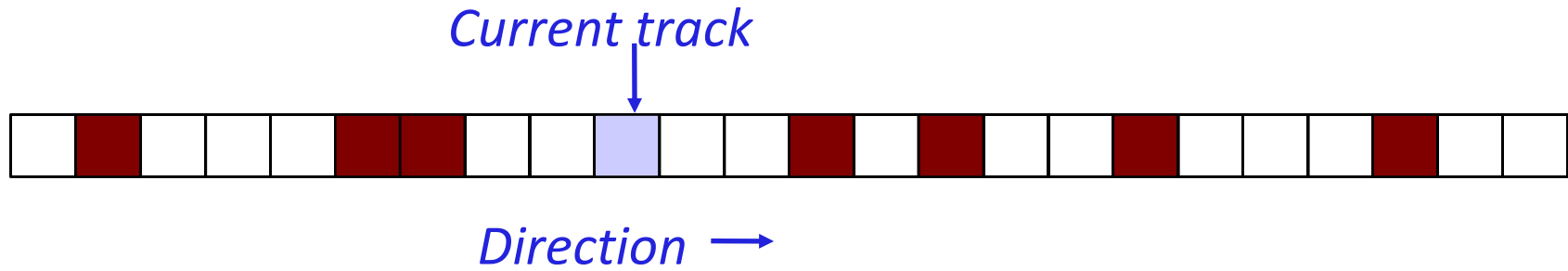
I/O queue at t0: 9,1,14,6,5,21,17,12



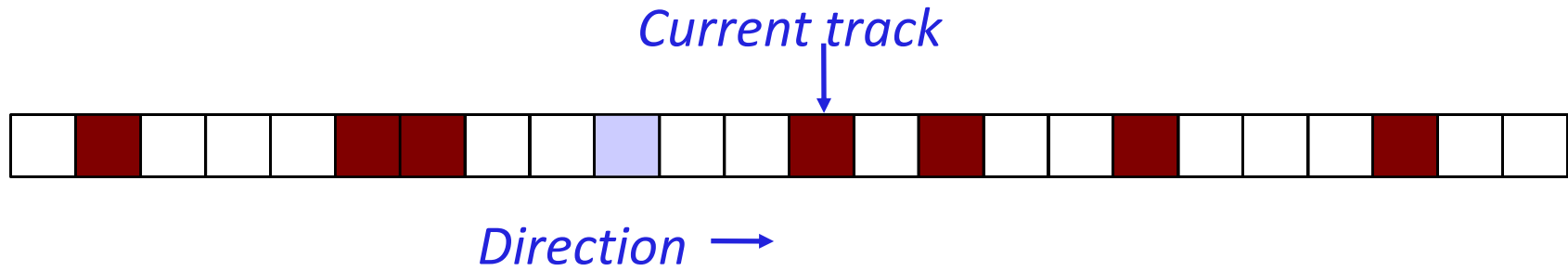
SCAN example



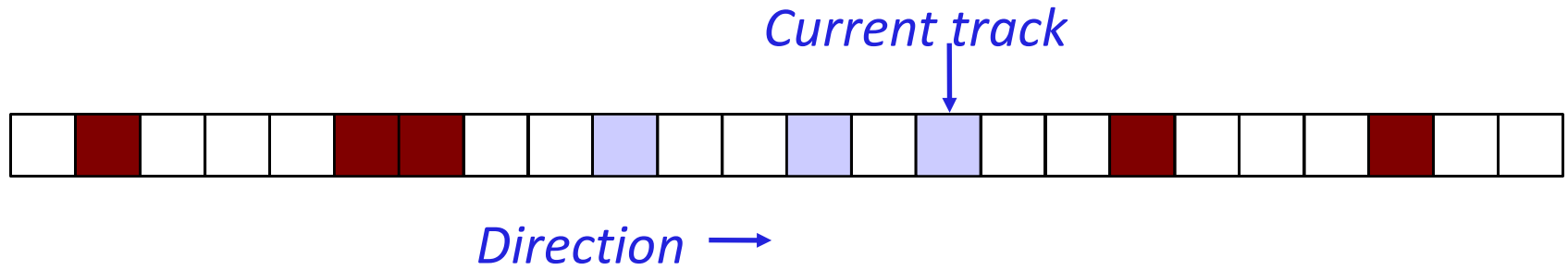
SCAN example



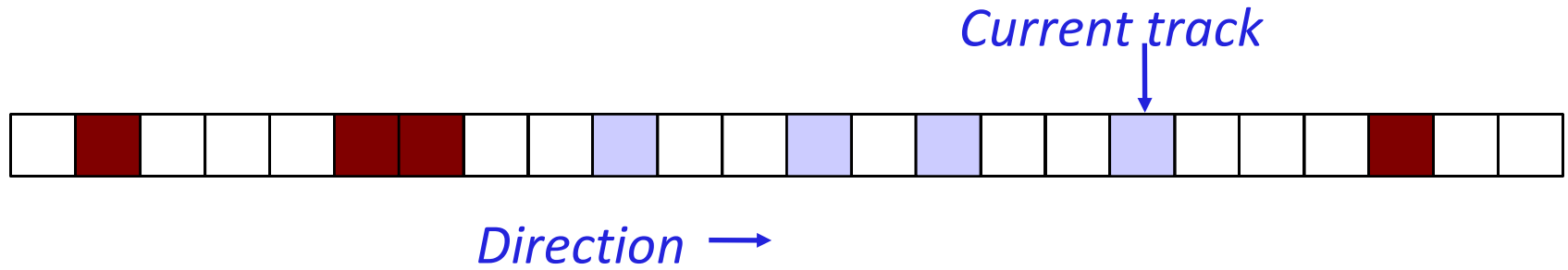
SCAN example



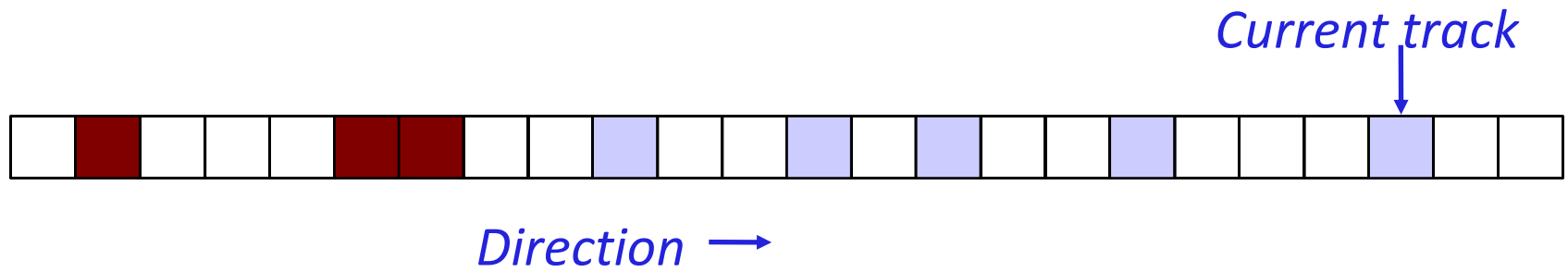
SCAN example



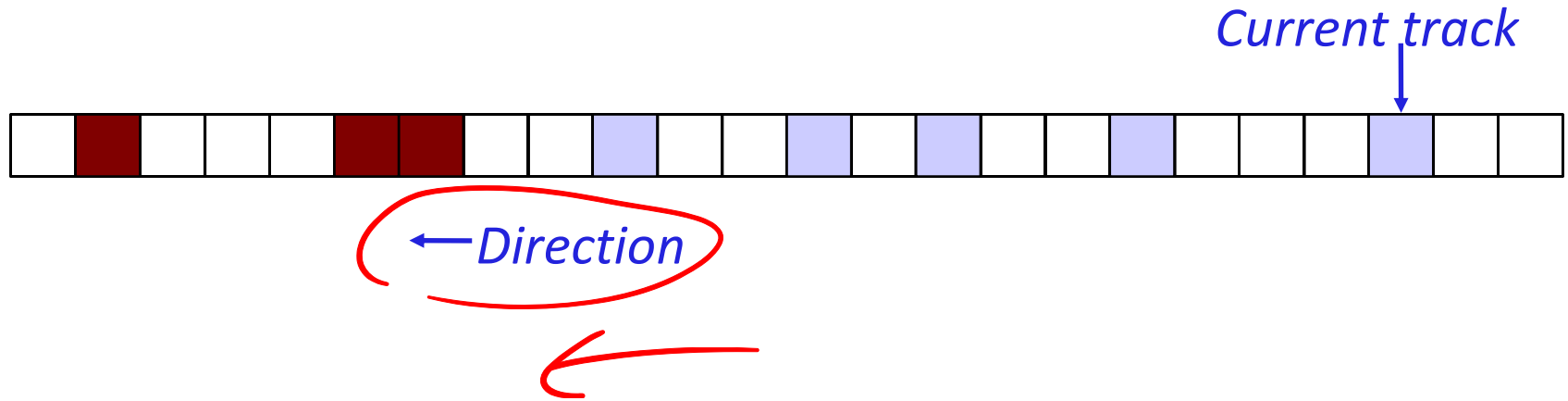
SCAN example



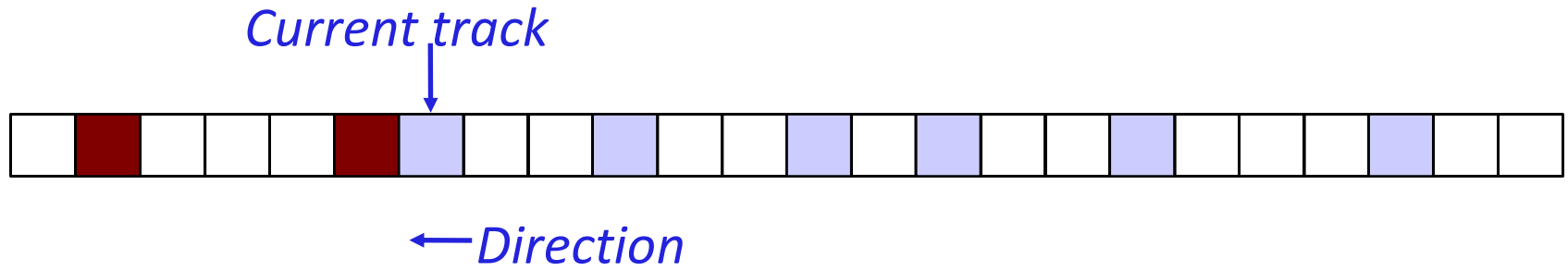
SCAN example



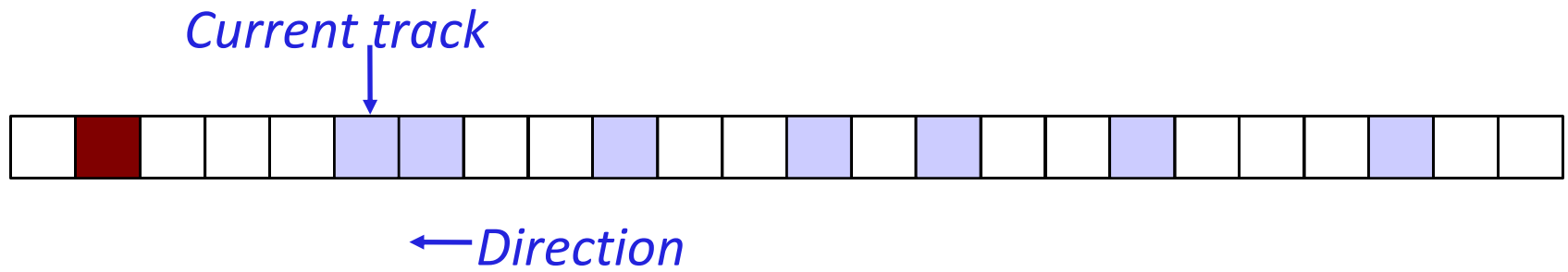
SCAN example



SCAN example

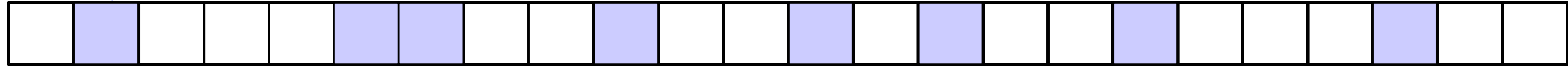


SCAN example



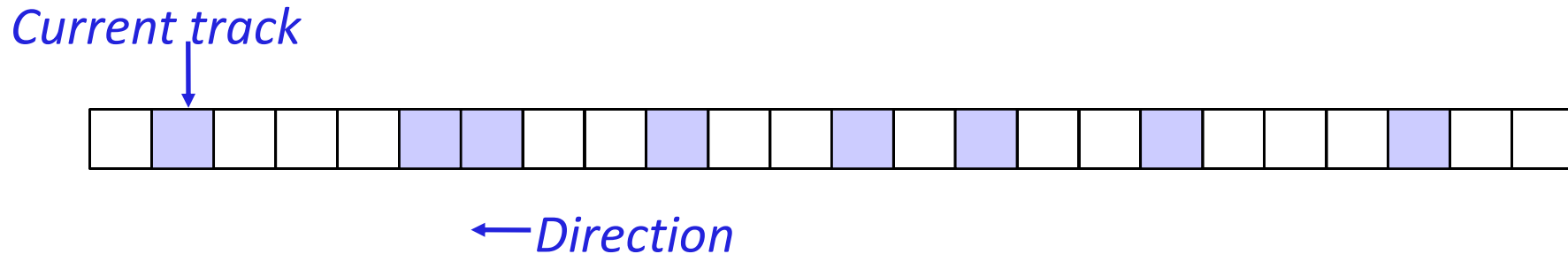
SCAN example

Current track



← Direction

SCAN example

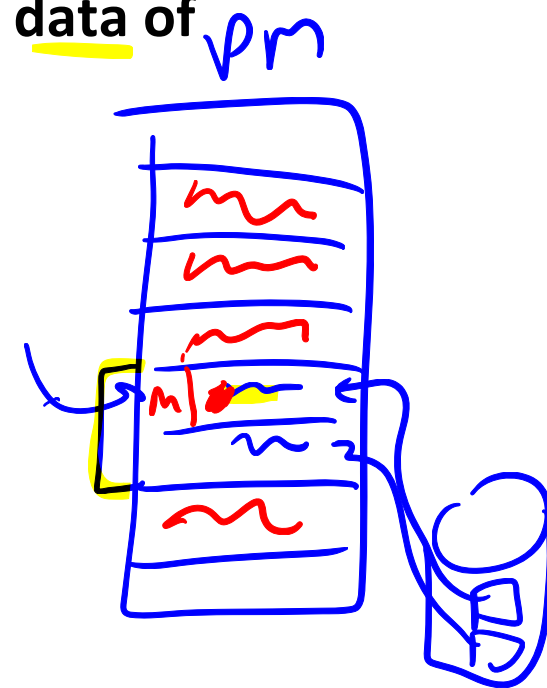


■ What is the overhead of the SCAN algorithm?

- Count the *total amount of seek time* to service all I/O requests
- In this case, 12 tracks in \rightarrow direction
- 15 tracks for long seek back
- 5 tracks in \leftarrow direction
 - Total: $12+15+5 = 32$ tracks

Page Cache : Caching Block devices

- Block device operations are tightly coupled with virtual memory and paging.
- Some of the frames are used as page cache and keeps data of block devices in systems.
- I/O in a block device:
 - Search if block is already in page cache (in physical memory):
 - if found read/write buffer from/to existing frame
 - Else allocate a frame, read device block into frame, mark frame as caching device-block pair read/write buffer from/to this frame.
- Dirty pages are written on block device periodically.
- Accelerates I/O operations significantly, especially file system meta data operations.



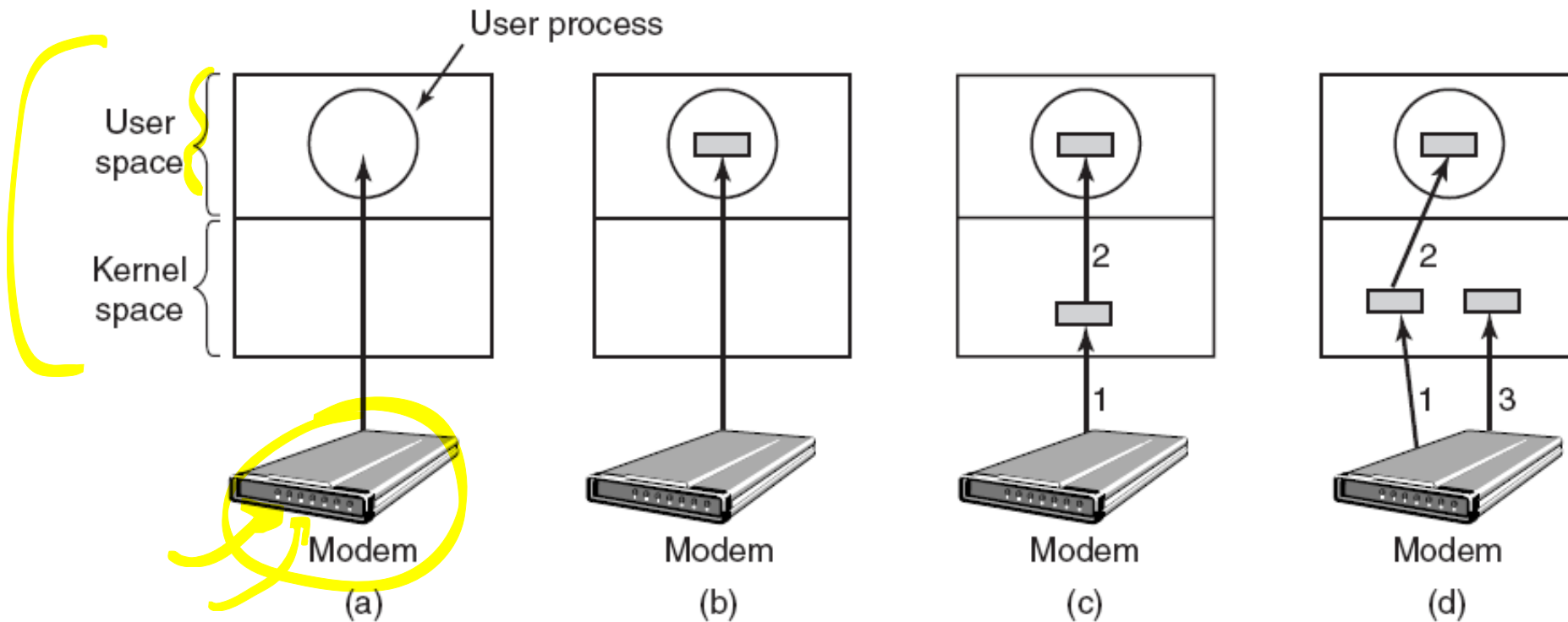
File Cache

- Page cache idea also couples with memory mapped I/O. `mmap ()`'ed files work in a similar mechanism.
- Virtual memory of a process map a page backed as a file (instead of a block device). Changes are updated on memory, cached frames are forced on disk periodically.
- VM system keeps track of frames of page and file caches together with other (resident and free) pages.
- VM system adapts sizes of file and device cache based on memory state of the system.

Buffering

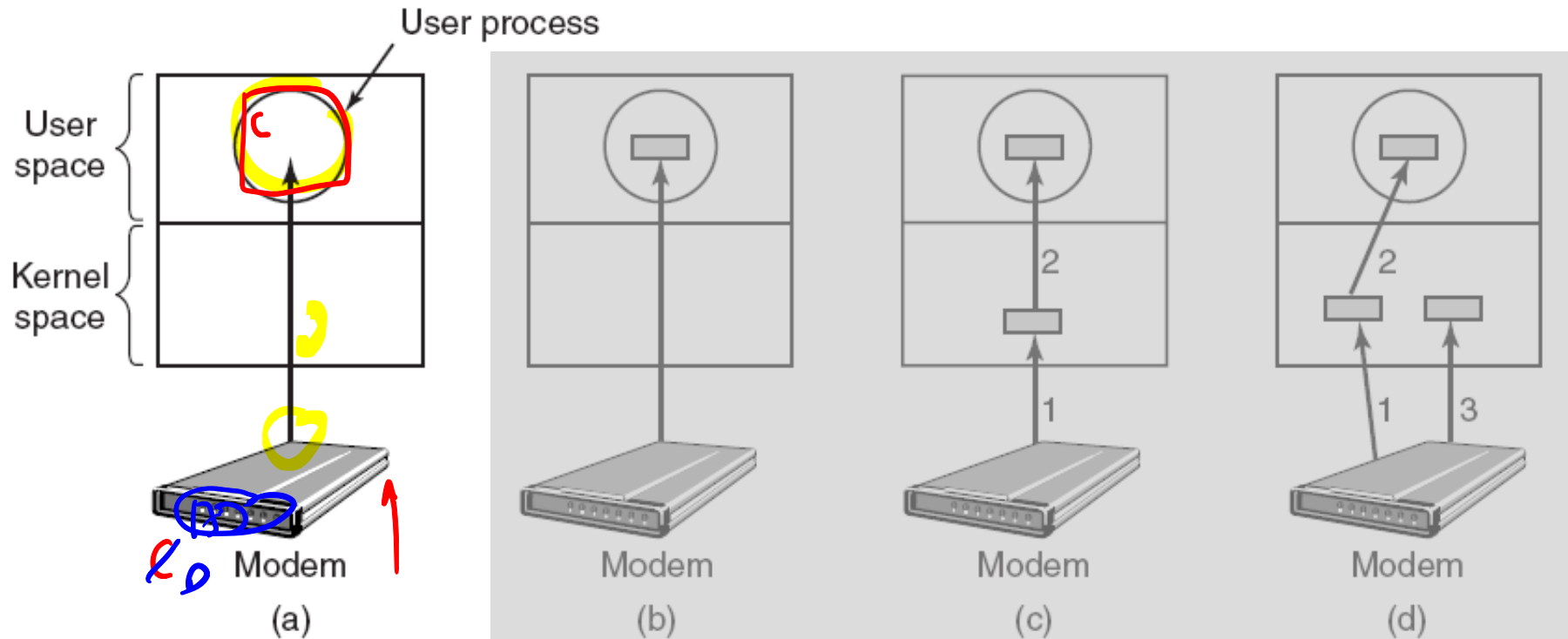
- A **buffer** is a memory area that stores data while they are transferred between two devices or between a device and an application.
- Buffering is done for three reasons.
 - to cope with a **speed mismatch** between the producer and consumer of a data stream.
 - a file is being received via modem for storage on the hard disk
 - to adapt between devices that have **different data-transfer sizes**.
 - **networking**: messages are typically fragmented during sending and receiving
 - to **support copy semantics** for application I/O.
 - application calls the **write()** system call, providing a pointer to the buffer and an integer specifying the number of bytes to write.
 - After the system call returns, what happens if the application changes the contents of the buffer?
 - When processing **write()** system call, OS copy the application data into a kernel buffer before returning control to the application.
 - The disk write is performed from the kernel buffer, so that subsequent changes to the application buffer have no effect.

Buffering



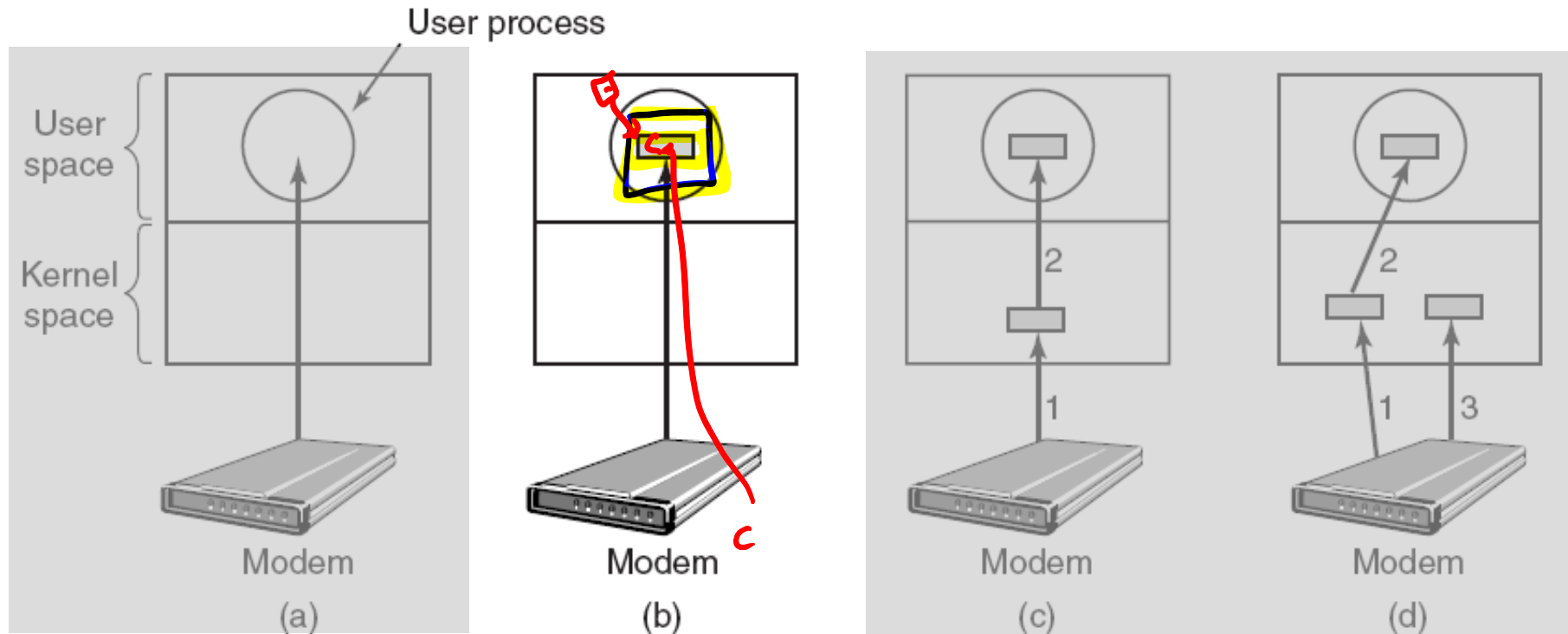
- a) Unbuffered input:
- b) Buffering in user space.
- c) Buffering in the kernel followed by copying to user space.
- d) Double buffering in the kernel.

Buffering: Unbuffered input



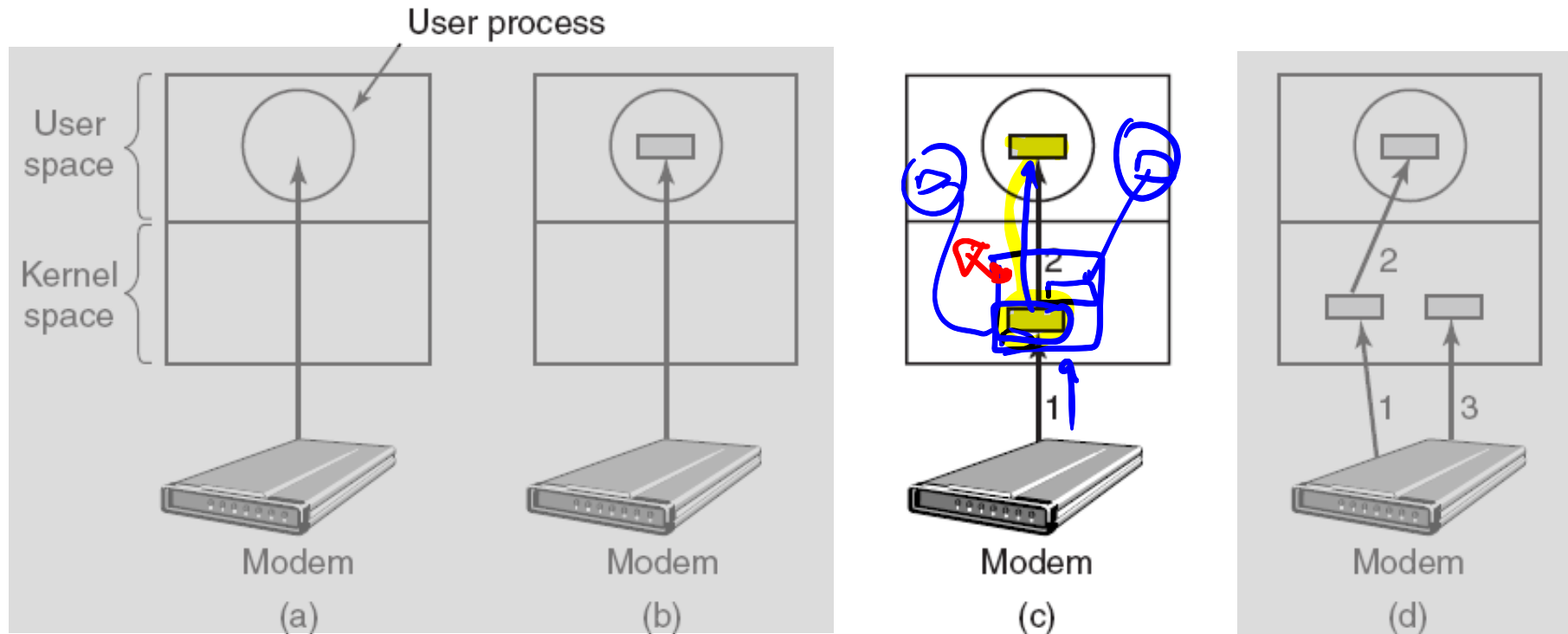
- I/O device writes directly into the user's address space.
- If the page frame that the I/O device need to write is not in the memory, then a page fault is generated.
- The I/O device may fail to write the incoming data, if new data arrives and overwrites the unwritten data in the memory/registers of the device.

Buffering: Buffering in user space



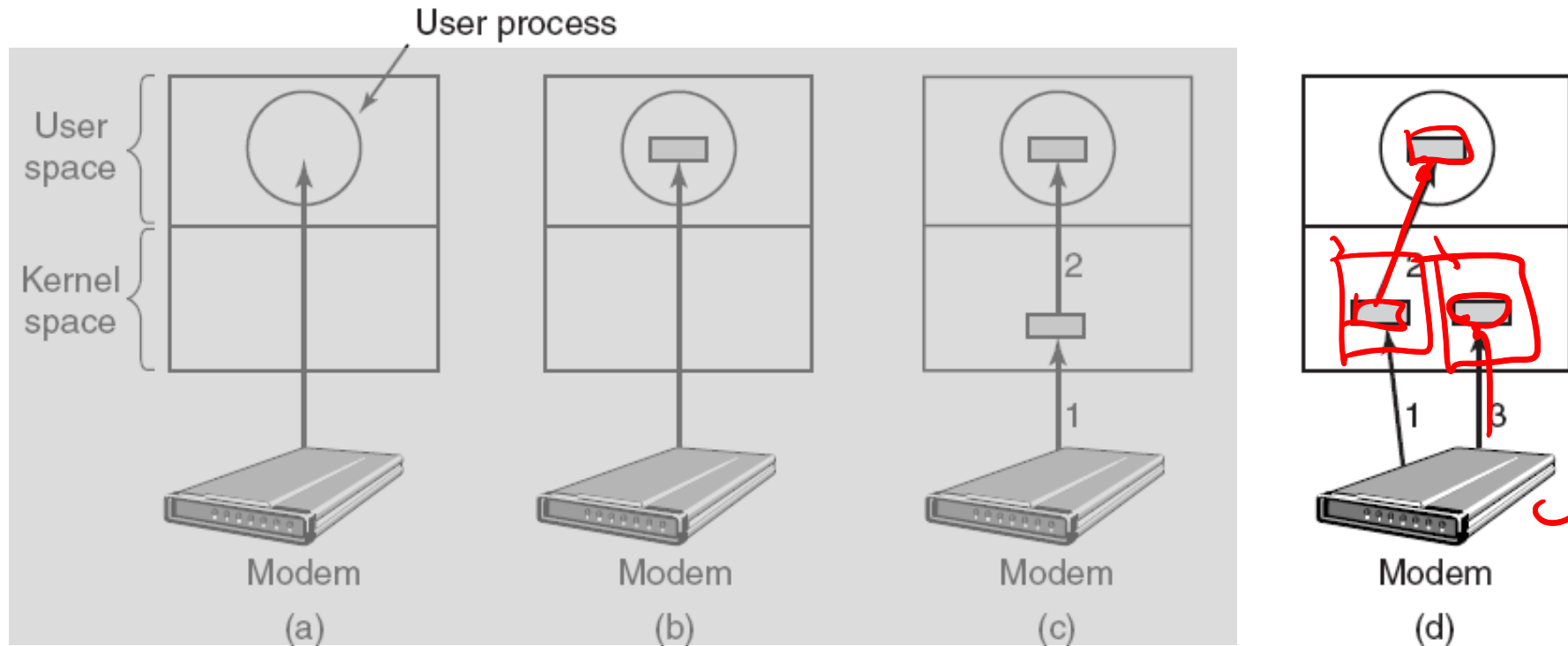
- I/O device writes directly into the user's address space. The page frame that the I/O device need to write is «pinned» into the memory, i.e. The page frame is never evicted.
- If all such page frames (of all the processes) are «pinned», then the physical memory will become tight!

Buffering: Buffering in kernel space



- I/O device writes directly into the kernel's address space.
- The kernel page frame that the I/O device need to write is «pinned» into the memory, i.e. The page frame is never evicted.
- Note that the kernel can combine many such buffers into a single kernel page frame, minimizing the number of «pinned» kernel page frames in the physical memory!
- When the kernel buffer becomes full, it is copied to the user's address space, and emptied.
- **What if, during copying, new data arrives?**
 - It may be lost!

Buffering: Double buffering in kernel space



- I/O device writes directly into the kernel's address space.
- The kernel page frame that the I/O device need to write is «pinned» into the memory, i.e. The page frame is never evicted.
- When the kernel buffer becomes full, it is copied to the user's address space, and emptied.
- During copying, the I/O device is directed to write into a second buffer!

Caching vs. Buffering




■ The difference between a buffer and a cache is that

- a buffer may hold the only existing copy of a data item,
- whereas a cache, by definition, just holds a copy on faster storage of an item that resides elsewhere.

■ Caching and buffering are distinct functions, but sometimes a region of memory can be used for both purposes.

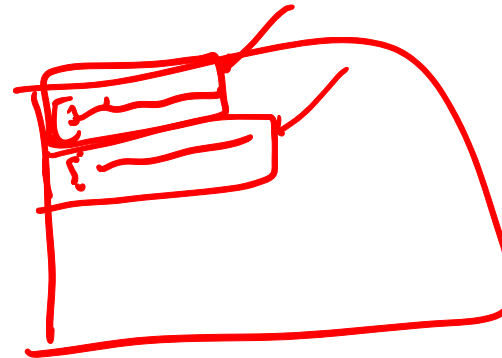
- For instance, to preserve copy semantics and to enable efficient scheduling of disk I/O, the operating system uses buffers in main memory to hold disk data.

I/O call semantic

- I/O calls can be categorized into two based on their semantics
 - Blocking 
 - Non-blocking 
- For most I/O calls, both types of semantics are available 

Blocking I/O semantic

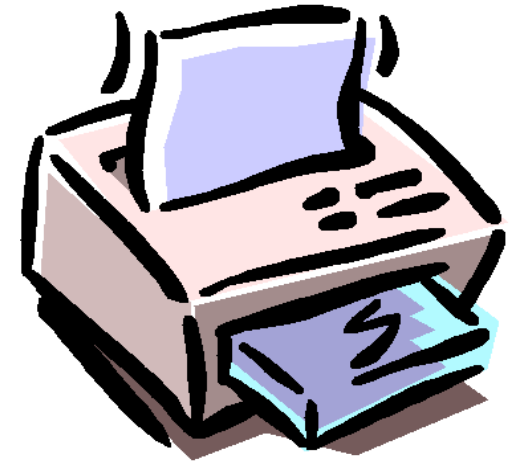
- When an application issues a blocking system call, the execution of the application is suspended.
 - The application is moved from the operating system's ready queue to a wait queue.
 - After the system call completes, the application is moved back to the ready queue.
 - Easy to understand



Nonblocking I/O semantic

- A nonblocking call does not halt the execution of the application for an extended time.
- The call returns immediately
 - Either the call returns the available data (may be none) along with a return value indicated how many bytes were transferred.
 - E.g. user interface that receives keyboard and mouse input while processing and displaying data on the screen.
 - The completion of the I/O at some future time is communicated to the application,
 - either through the setting of some variable in the address space of the application,
 - or through the triggering of a signal or software interrupt
 - or a call-back routine that is executed outside the linear control flow of the application.
- In either case, after the return from the call, the application continues to execute its code.

Spooling



- A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams.
 - Although a printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together.
 - The operating system solves this problem by intercepting all output to the printer.
 - Each application's output is spooled to a separate disk file.
 - When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer.
 - The spooling system copies the queued spool files to the printer one at a time.
 - In some operating systems, spooling is managed by a system daemon process.

Error handling

- An operating system that uses protected memory can guard against many kinds of hardware and application errors, so that a complete system failure is not the usual result of each minor mechanical glitch.
 - Devices and I/O transfers can fail in many ways, either for transient reasons, such as a network becoming overloaded, or for “permanent” reasons, such as a disk controller becoming defective.
 - Operating systems can often compensate effectively for transient failures.
 - For instance, a disk read() failure results in a read() retry, and a network send() error results in a resend(), if the protocol so specifies.
 - Unfortunately, if an important component experiences a permanent failure, the operating system is unlikely to recover.

I/O system in OS: Summary

- **How to access I/O devices in HW**
 - **Device controllers and device drivers**
- **How to interact with I/O devices?**
 - **Poll based vs. Interrupt based I/O**
 - **CPU checks if I/O is complete**
 - **An interrupt is generated when I/O is complete**
 - **Programmed vs. DMA based I/O**
 - **Data is transferred to/from CPU**
 - **DMA controller transfers data from device buffer to main memory without CPU intervention**
- **How are I/O devices categorized?**
 - **Character vs. Block**
 - **Streams of chars (e.g. printer, modem)**
 - **Units of blocks (e.g. disks)**
- **Caching and Buffering**
- **Blocking and non-blocking semantics**
- **Other issues**
 - **Error handling,**