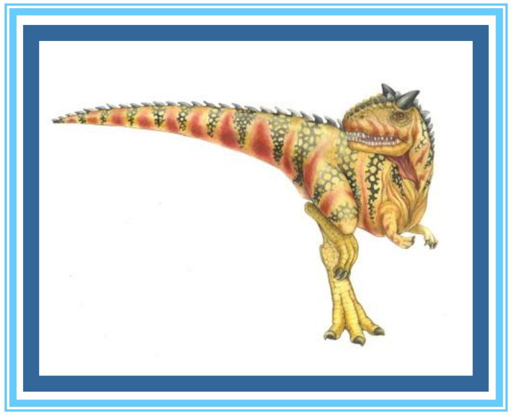
Xi'an Jiaotong-Liverpool University
西交利物浦大學

CPT104 - Operating Systems Concepts

I/O Systems



I/O Systems

- Overview
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Streams
- Performance

Overview

- The control of devices connected to the computer is a major concern of operating-system designers
- I/O devices vary so widely in their function and speed
- Various methods to control them
- Performance management
- New types of devices frequent
- To encapsulate different devices, the kernel of an OS is structured to use *device-driver modules*.
- The **device drivers** present a uniform *device access interface* to the I/O subsystem (provide a standard interface between the application and the operating system).

Overview

The I/O devices are classified:

Human-readable and **machine-readable**: The human-readable devices are mouse, keyboard, and so on, and the machine-readable devices are sensors, controllers, disks, etc.

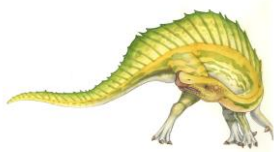
Transfer of data

- *character-oriented device* - accepts and delivers the data as a stream of characters/bytes
- *block-oriented device* – accepts and delivers the data as fixed-size blocks

Type of access

- *sequential device* such as a tape drive.
- *random access device*, such as a disk.

Network device - to send or receive data on a network.



I/O Hardware

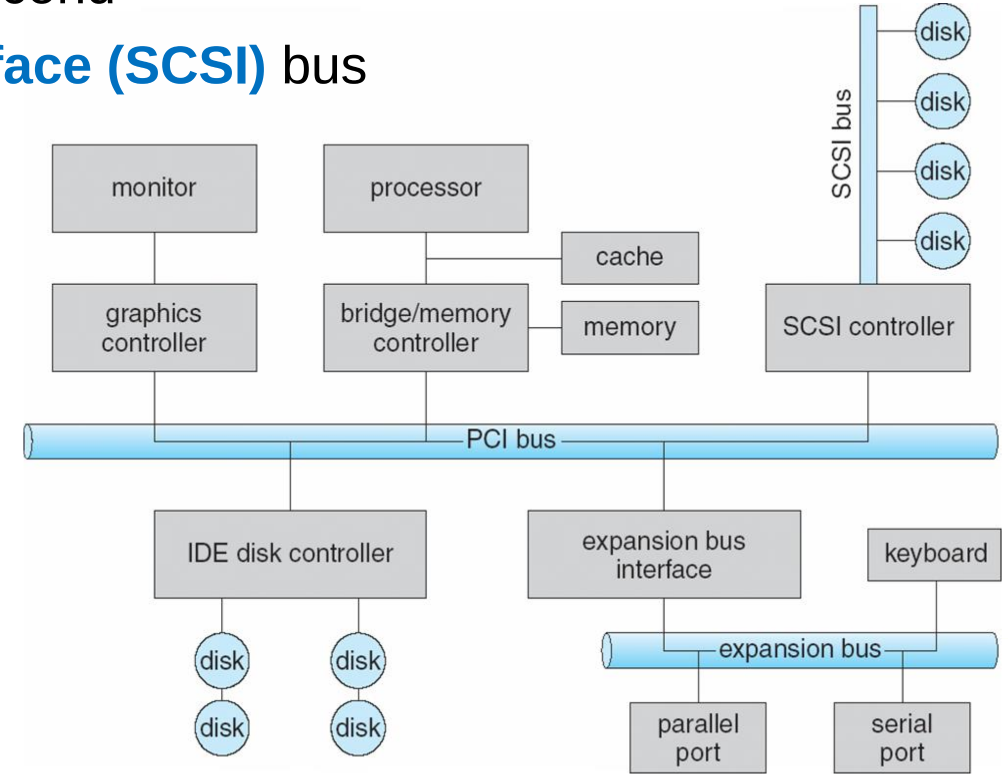
A device communicates with a computer system by sending signals over a cable or even through the air.

- **Port** – connection point for device
- **Bus** - a set of wires and a defined protocol that specifies a set of messages that can be sent on the wires
  - **PCI** bus common in PCs and servers, *PCI Express* (**PCIe**)
  - **expansion bus** connects relatively slow devices
- **Controller** (**host adapter**) – is an electronic component that allows computer systems communication with devices.
  - circuit board and/or integrated circuit adapter

Controller

Typical PC Bus Structure

- **PCI** bus - connects the processor- memory subsystem to fast devices
- **expansion** bus - connects relatively slow devices (keyboard and serial and USB ports)
- **PCI Express (PCIe)** - 16 GB per second
- **HyperTransport** - 25 GB per second
- **Small Computer System Interface (SCSI)** bus
- **daisy-chain** bus (not shown)



How can the processor give commands and data to a controller to accomplish an I/O transfer?

- Controller has ***registers*** (a small and temporary storage unit) that are used for communicating with the CPU.
- By ***writing*** into these registers, the operating system can command the device to deliver data, accept data, etc.
- By ***reading*** from these registers, the operating system can learn what the device's state is, if it is prepared to accept a new command, etc.
- Devices have a **data buffer** that the operating system can read and write.

There are 4 registers (1 to 4 bytes in size) :

- **data-in** register
- **data-out** register

$\Big\}$ - data registers to pass data to the device or get data from.

- **status** register - can be read to see the current status of the device.
- **control** register - to tell the device to perform a certain task.

How the CPU communicates with the control registers and with the device data buffers?

Two alternatives exist:

Port-mapped I/O (PMIO)

- each control register is assigned an **I/O port number** (8- or 16-bit integer).
- the set of all the I/O ports form the **I/O port space**, that ordinary user programs cannot access it (only the operating system can).
- access by special **I/O instruction** (e.g. IN, OUT, MOV)

Memory-mapped I/O - maps all the control registers into the memory space

- each control register is assigned a unique memory address
- more efficient for large memory I/O (e.g. graphic card)
- vulnerable to accidental modification, error

I/O communication techniques

There are three fundamentally different ways that I/O can be performed (techniques to interact with a device and control the transfer with a device):

- ❑ **Polling (Programmed I/O)**
- ❑ **Interrupt-driven I/O**
- ❑ **Direct Memory Access**

1. Polling (Programmed I/O)

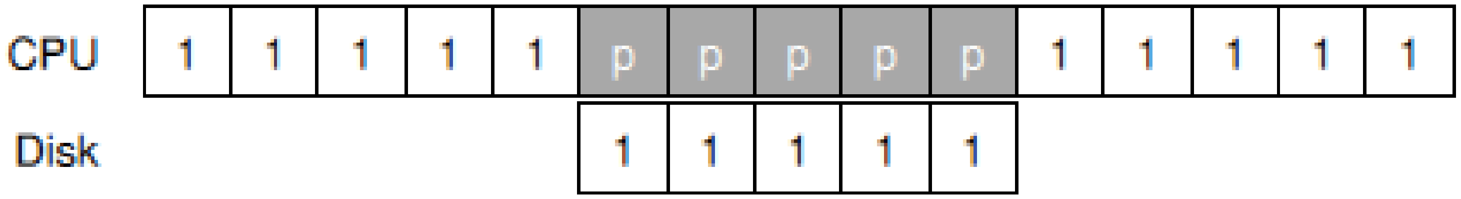
The **data transfer** is **initiated by the instructions written in a computer program.**

CPU executes a **busy-wait loop** – periodically **checking status of the device** (to see if it is time for the next I/O operation).

CPU stays in a loop until the I/O device indicates that it is ready for data transfer.

Polling can be very fast and efficient, if both the device and the controller are fast and if there is significant data to transfer.

Disadvantage: It keeps the processor busy needlessly and leads to wastage of the CPU cycles



2. Interrupt-driven I/O

- to reduce the processor waiting time $\Rightarrow$ a **hardware mechanism = interrupt.**
- the CPU has an **interrupt-request line** that is sensed after every instruction.
- **Interrupts** allow devices to notify the CPU when they have data to transfer or when an operation is complete.
- The CPU transfers control to the **interrupt handler.**

Most CPUs have **two** interrupt-request lines:

Non-maskable - for critical error conditions.

Maskable - used by device controllers to request / the CPU can temporarily ignore during critical processing.

How does the processor know when the I/O is complete?

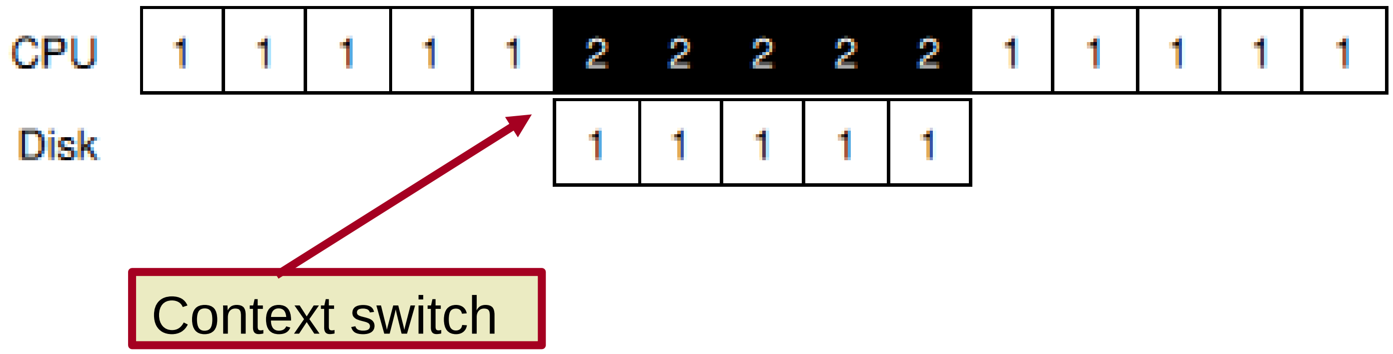
Through an **interrupt mechanism** (see next *Interrupt-driven I/O Cycle*).

- when the operation is complete, the device controller generates an interrupt to the processor.

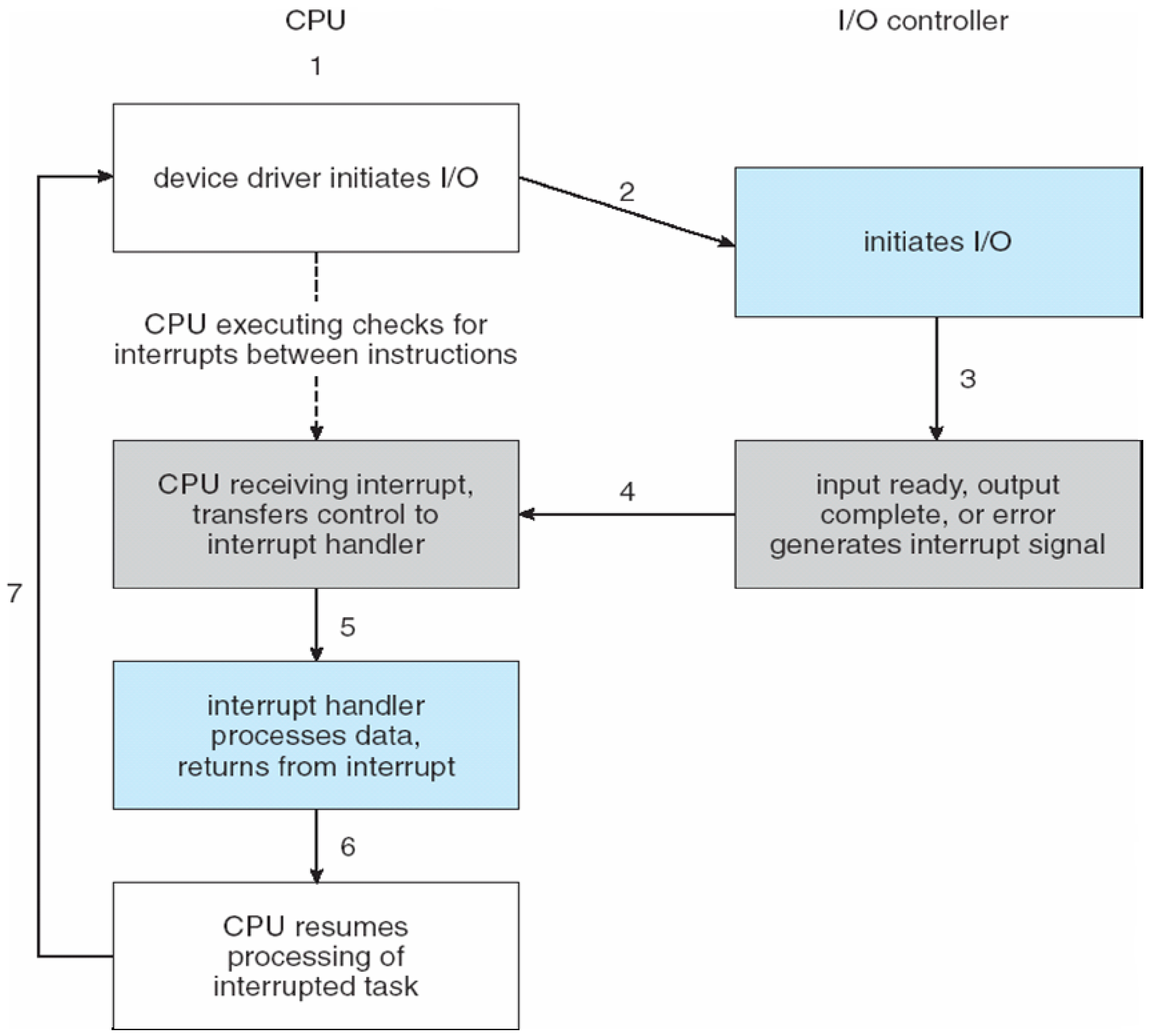
- **NB**: the processor checks for the interrupt after every instruction cycle.

- after detecting an interrupt, the processor will perform a *context switch*, by executing the **Interrupt Service Routine**.

- **Interrupt handler** receives interrupts the processor then performs the data transfer for the I/O operation.



Interrupt-Driven I/O Cycle


CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

3. Direct Memory Access

DMA = Direct Memory Access

- when the data are large, interrupt driven I/O is not efficient.
- instead of reading one character at a time through the processor, a **block of characters is read at a time**.
- bypasses CPU to transfer data directly between I/O device and memory
- DMA hardware generates an interrupt when the I/O transaction is complete
- requires **DMA controller**
- version that is aware of virtual addresses can be even more efficient - **Direct Virtual Memory Access DVMA**

Direct Memory Access

To *read* or *write* a *block*, the processor sends the command to the DMA controller.

The processor passes the following information to the DMA controller:

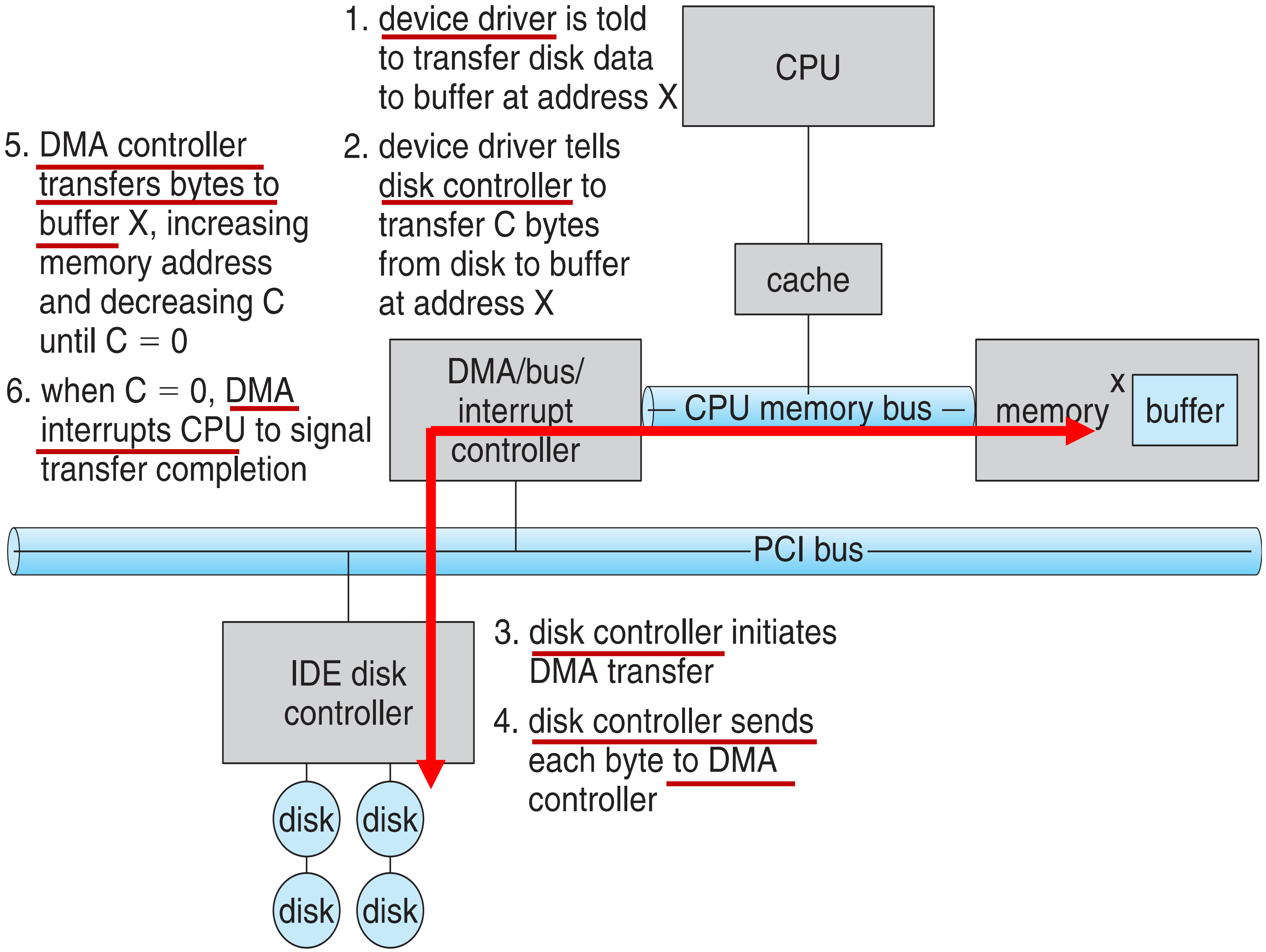
• The type of request (*read* or *write*)

• The address of the I/O device to which I/O operation is to be carried out

• The start address of the memory, where the data need to be written or read from, along with the total number of words.

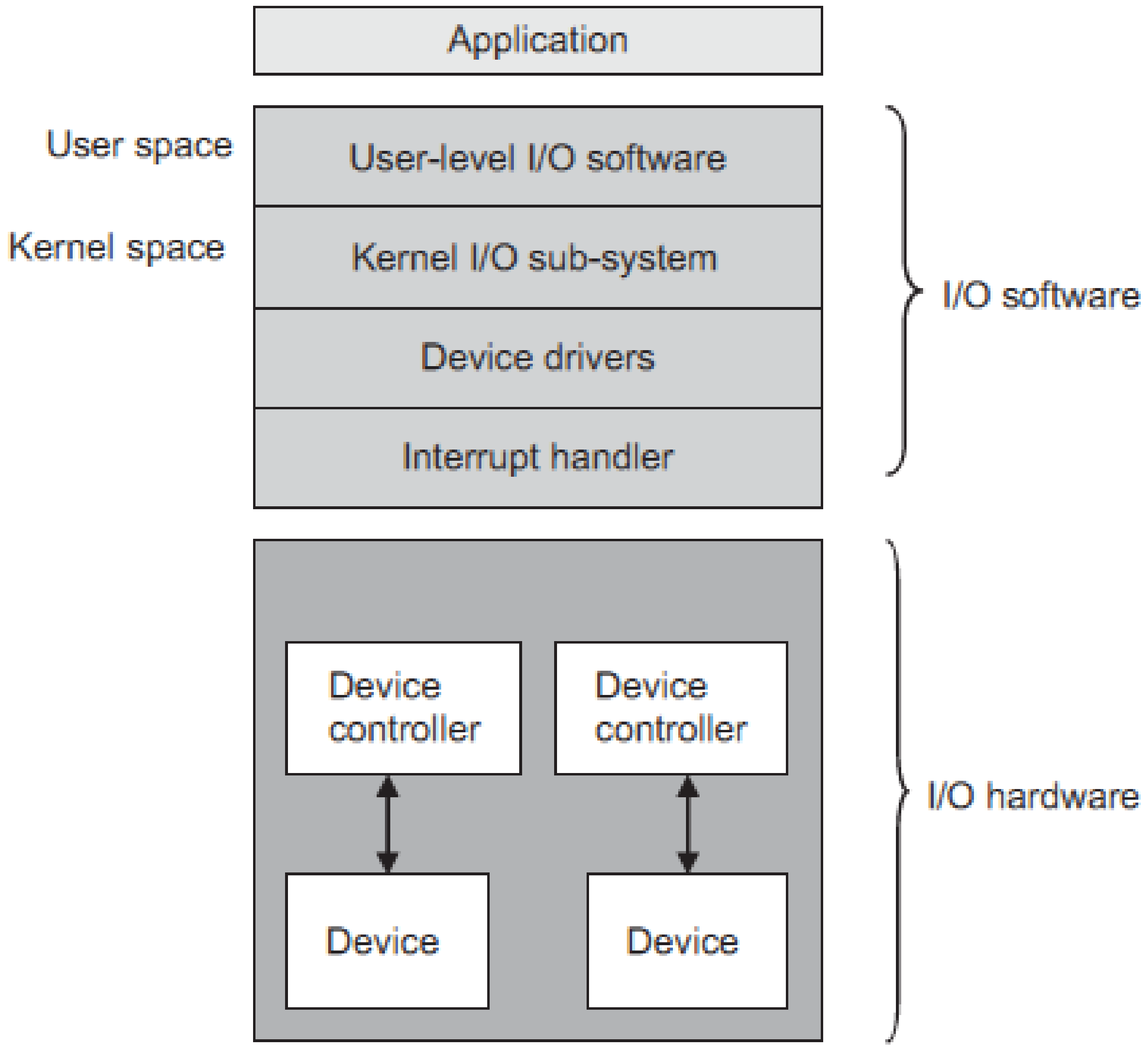
• The DMA controller then copies this address and the word count to its registers.

Six-step Process to Perform DMA Transfer



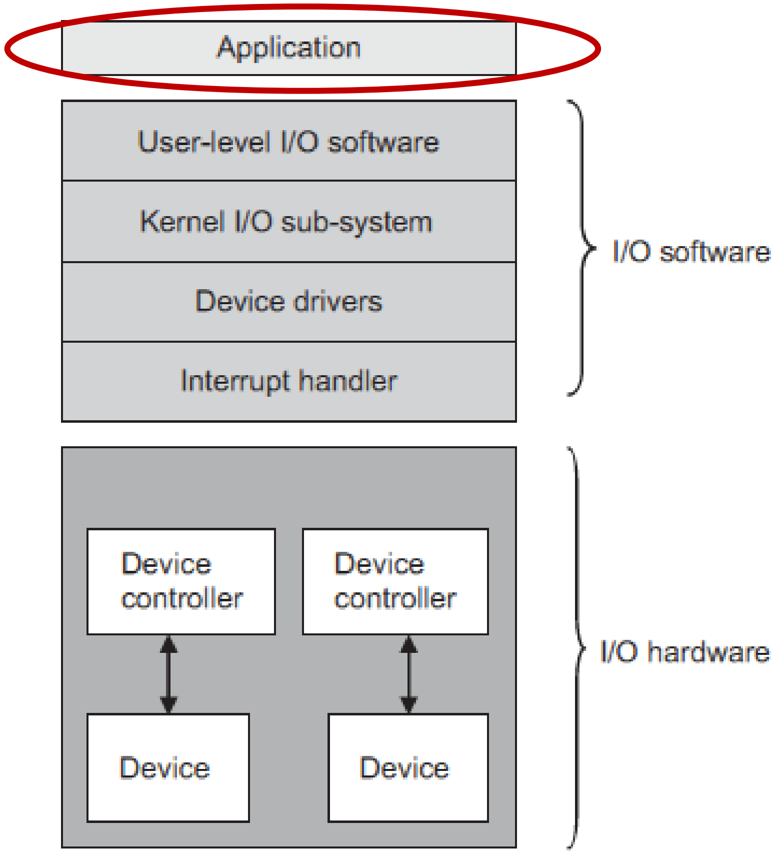
1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion

Layered structure of I/O

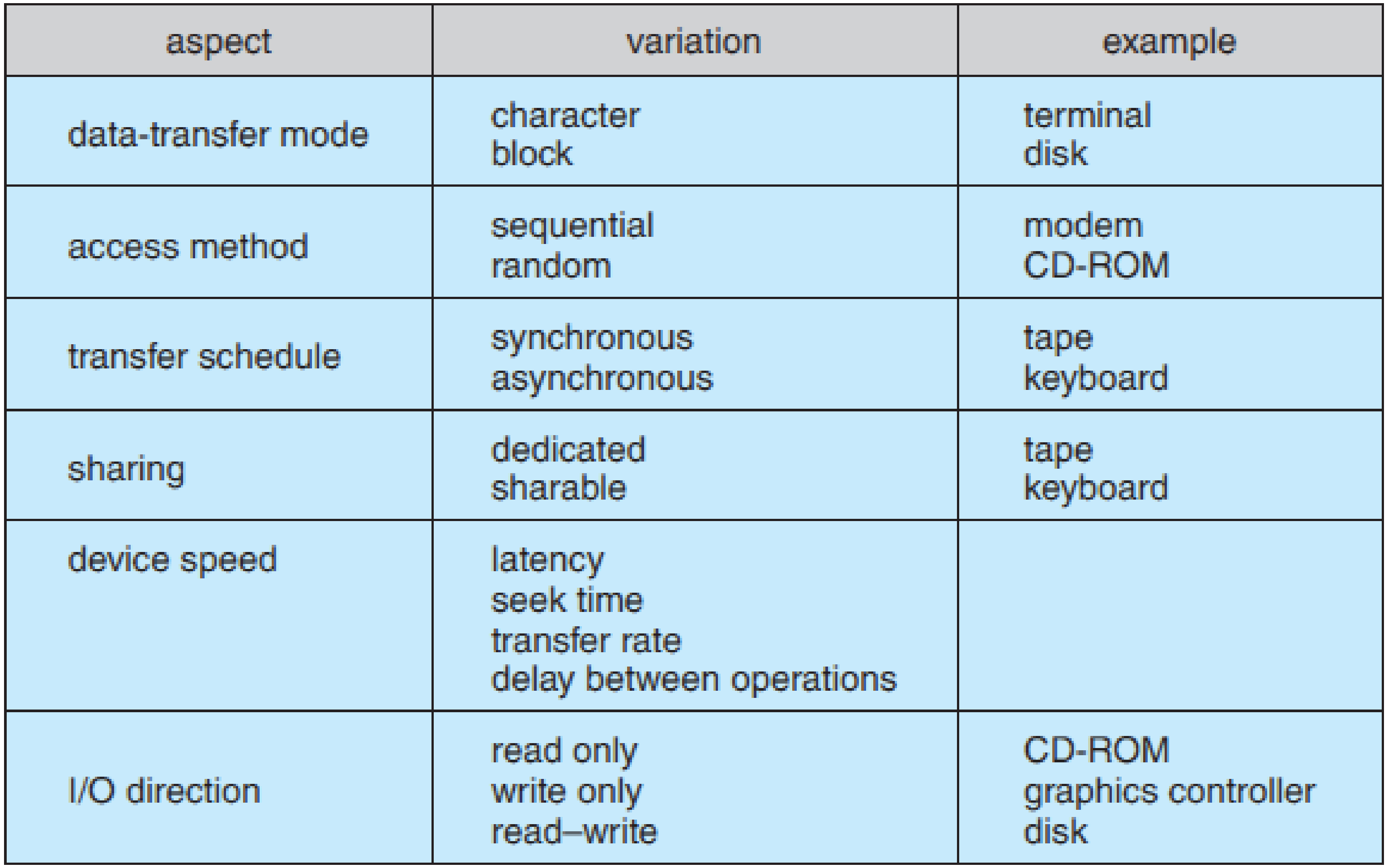


1. Application I/O Interface

- User application access to a wide variety of different devices
- Devices vary in many dimensions:
  - **Character-stream** or **block**
  - **Network Devices**
  - **Clocks and Timers**
  - **Blocking and Non-blocking I/O**
  - **Vectored I/O** (new)



Characteristics 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

Block and Character Devices

- Block devices are accessed a block at a time
  - include **disk drives** and **block-oriented devices**
  - commands include **read, write, seek**
    - *Raw I/O* (accessing blocks on a hard drive directly),
    - *Direct I/O* (uses the normal *filesystem* access)
    - *Memory-mapped file I/O*.
- Character devices are accessed one byte at a time
  - include **keyboards**, **mice, serial ports**
  - commands include **`get(), put()`**
  - supported by higher-level library routines.

Network Devices

- Varying enough from block and character to have own interface
- *Linux*, *Unix*, *Windows* and many others include **socket interface**
  - *socket* acts like a cable or pipeline connecting two networked entities.
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

Clocks and Timers

- Three types of time services are commonly needed in modern systems:
  - Get the current time of day.
  - Get the elapsed time since a previous event.
  - Set a timer to trigger event X at time T.
- A **Programmable Interrupt Timer** (PIT) is a hardware counter that generates an output signal when it reaches a programmed count.

Blocking and Non-blocking I/O

- Blocking – **process/app is suspended** (move it in the waiting queue) **until I/O completed**
  - most I/O requests are considered blocking requests, meaning that control does not return to the application until the I/O is complete.
- Non-blocking - **the I/O request returns immediately**, **whether the requested I/O operation has** (completely) **occurred or not**.

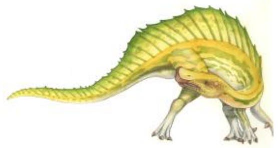
  Example:
  - a user interface that receives keyboard and mouse input while processing and displaying data on the screen.
  - a video application that reads frames from a file on disk while simultaneously decompressing and displaying the output on the display.
  - .

Vectored I/O

- known as **scatter/gather I/O**

Scatter/gather refers to **the process of gathering data from, or scattering data into**, the given **set of buffers**.

- read from or write to multiple buffers at once
- allows one system call to perform multiple I/O operations
- For example, Unix **readve()** accepts a vector of multiple buffers to read into or write from.



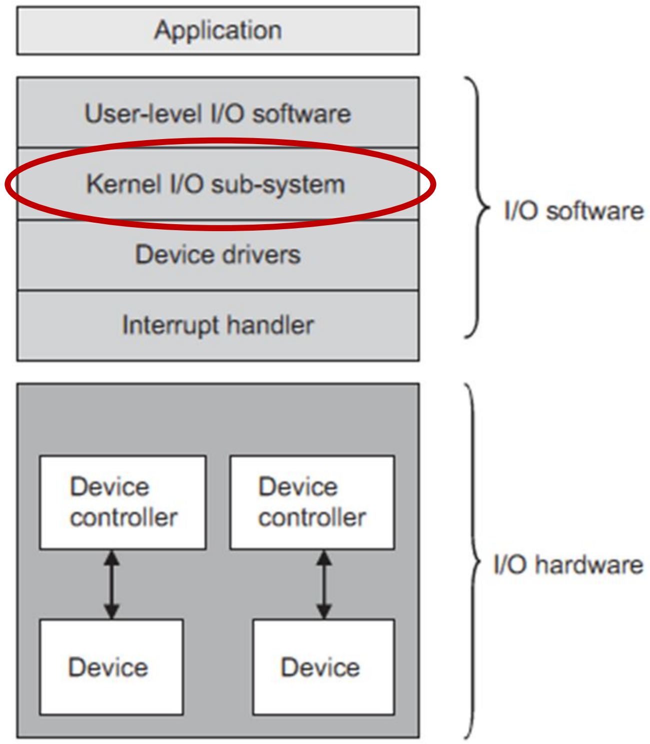
2. Kernel I/O Subsystem

This part is provided in the kernel space.

The user interacts with this layer to access any device.

There are different functions:

- Uniform Interface
- Scheduling
- Buffering
- Caching
- Spooling and Device reservation
- I/O protection
- Error Handling

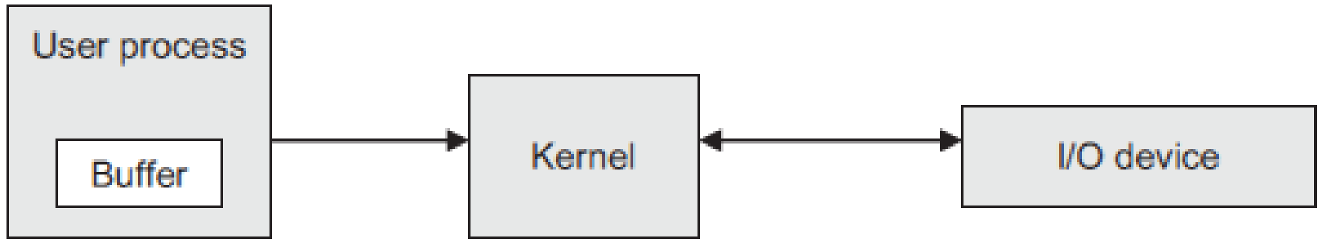


Kernel I/O Subsystem - Functions

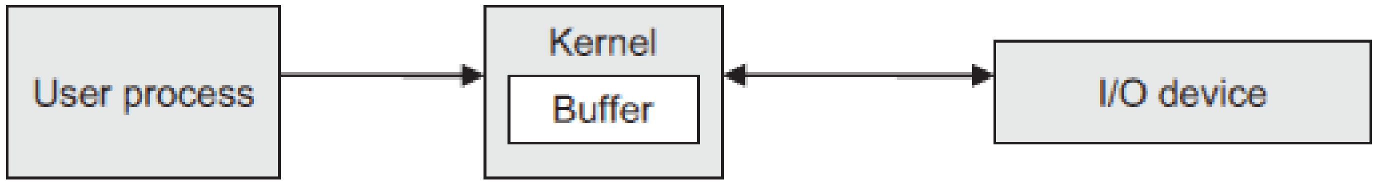
- **Uniform Interface** - makes uniform the interface, such that **all the drivers' interface through a common interface.**
- **Scheduling**
  - Some I/O request **ordering via per-device queue**
  - Some OSs try **fairness**
- **Buffering** - **store data while transferring between devices**
  - a buffer is an area where the data, being read or written, are copied in it, so that the operation on the device can be performed with its own speed. *(see the next slide)*
    - ***Single buffering***
      - kernel and user buffers
      - varying sizes
    - ***Double buffering*** – two copies of the data
      - permits one set of data to be used while another is collected.

Buffering

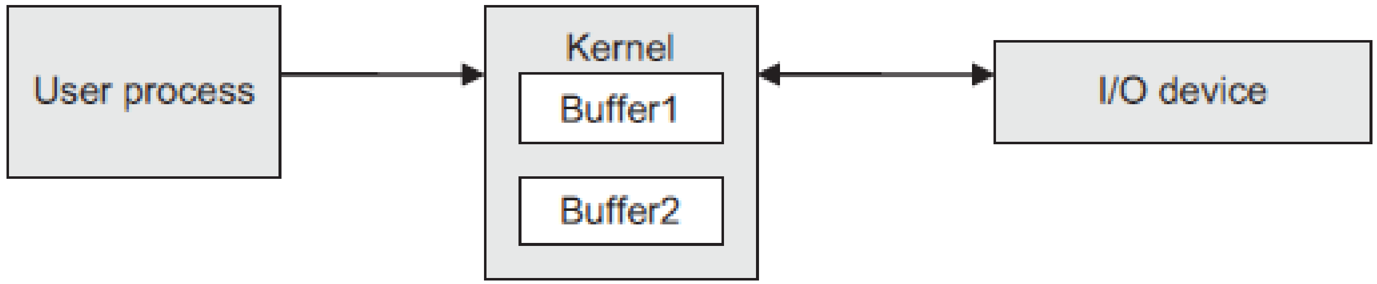
Buffering in user space



Buffering in kernel space



Double buffering



Kernel I/O Subsystem - Functions

- Caching - region of **fast memory holding copy of data**
    - involves keeping a copy of data in a faster-access location than where the data is normally stored.
    - *buffering* and *caching* use often the same storage space
- Spooling and Device reservation – (SPOOL = *Simultaneous Peripheral Operations On-Line)* = **buffers data for devices** that cannot support interleaved data streams.
    - device can serve **only one request at a time**
    - ***spool queues*** can be *general* or *specific*.

Kernel I/O Subsystem - Functions

- **I/O protection** - provides **exclusive access** to a device
  - All I/O instructions defined to be privileged
  - I/O must be performed via system calls that must be performed in kernel mode.
  - Memory-mapped and I/O port must be protected.
- **Error Handling** – the I/O functions, when performed, may sometimes result in **errors**.
  - **Transient Errors** - temporary reasons that cause any I/O processing to fail (e.g. buffers overflow)
  - **Permanent Errors** - due to the failure of any device or wrong I/O request (e.g. disk crash).

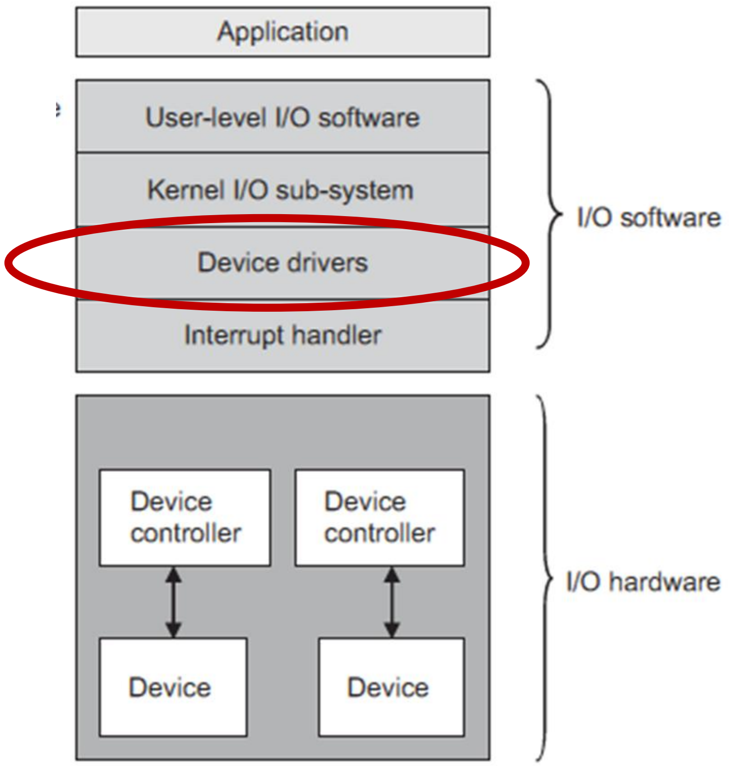
3. Device Driver

Piece of software a **device driver** **knows in detail how a device works.**

The functions of a device driver are to:

- **accept** the I/O requests **from the kernel** I/O sub-system.

- **control** the I/O operation.

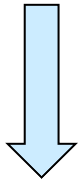


4. Interrupt Handler

The device driver **communicates** with the device controllers, and then the device, with the help of the **interrupt-handling mechanism.**

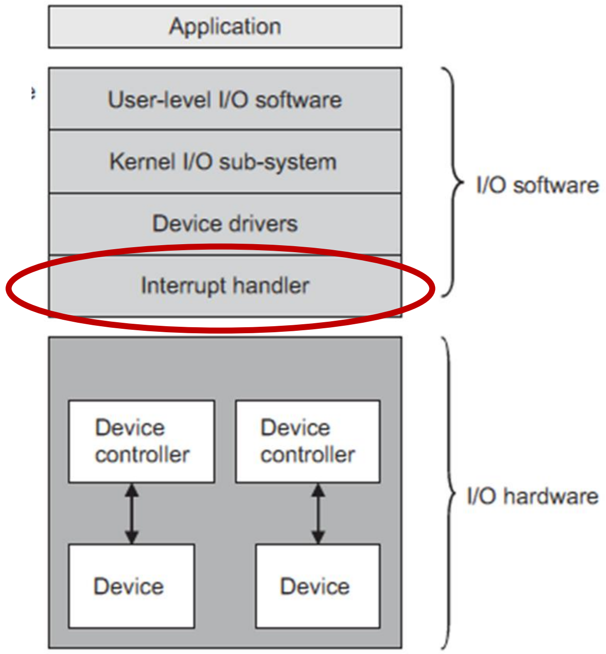
The **Interrupt Service Routine** (ISR) is executed in order to handle a specific interrupt for an I/O operation.

Transforming I/O Requests to Hardware Operations



Life Cycle of an I/O Request

see the next slide and the explanation in LMO

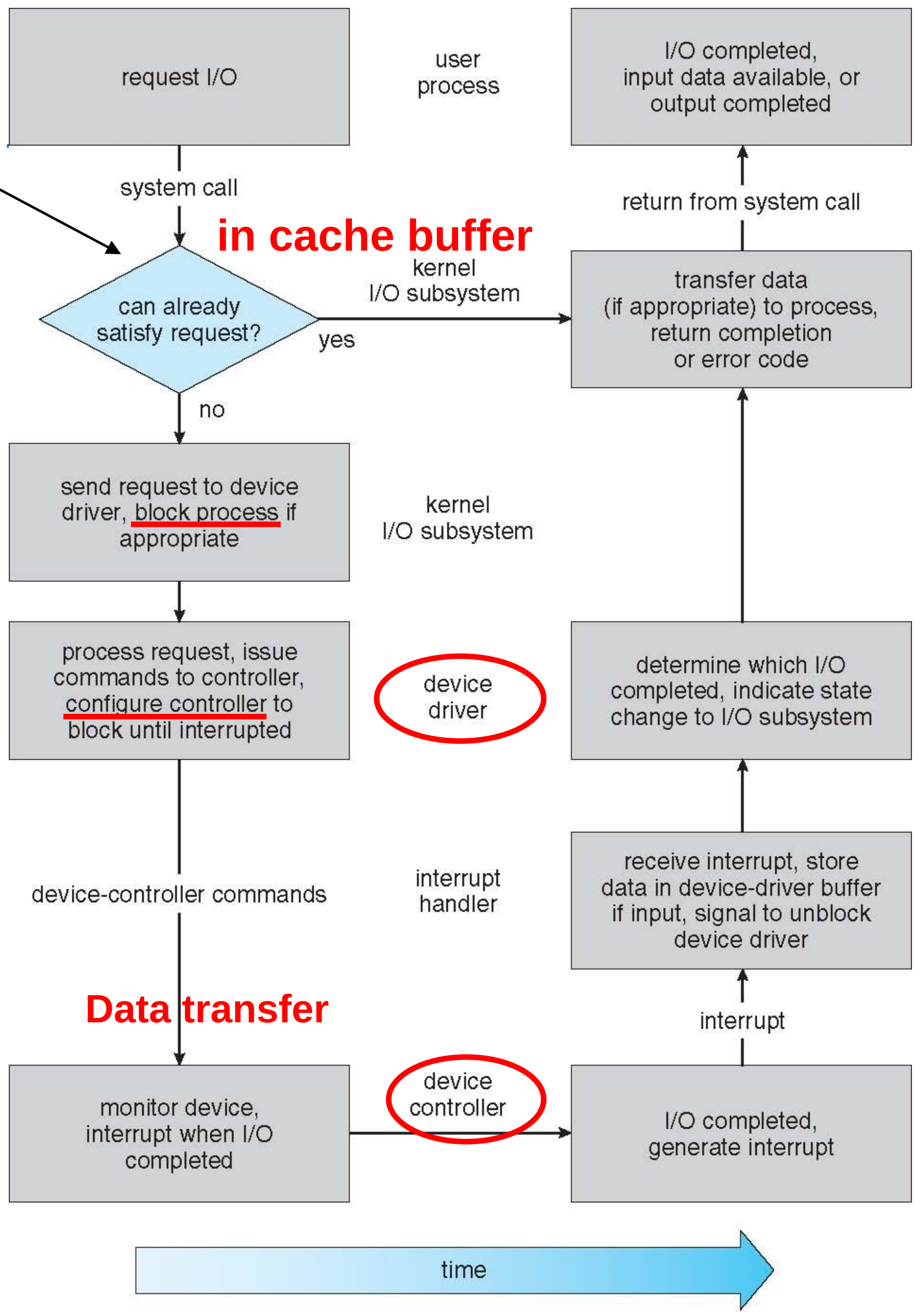


Check buffer cache

Otherwise, a **physical I/O needs** to be performed

Move process from *run queue* to *wait queue*

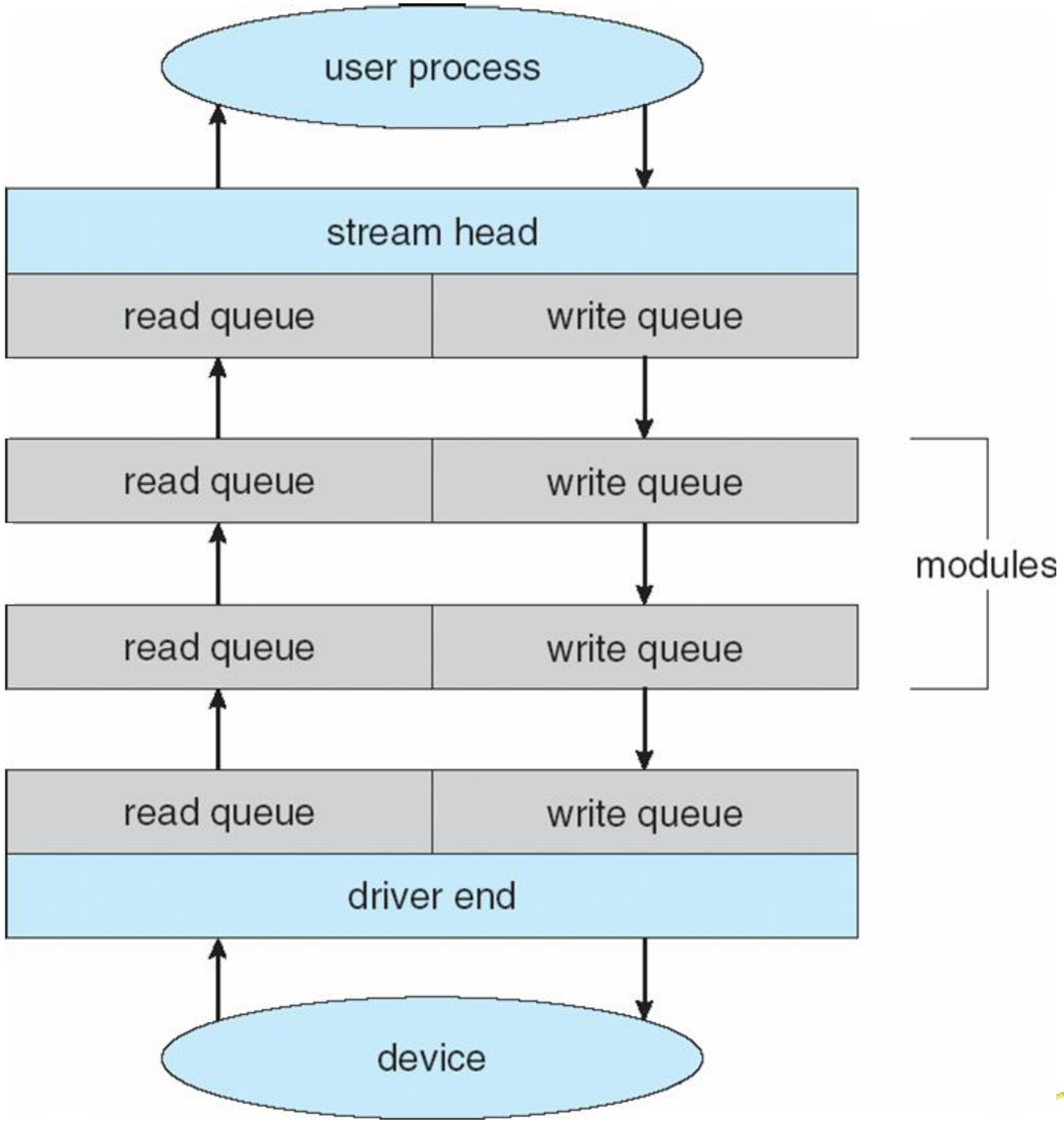
Allocate kernel buffer
Schedule I/O


request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
in cache buffer
kernel I/O subsystem
can already satisfy request?
yes
transfer data (if appropriate) to process, return completion or error code
no
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
determine which I/O completed, indicate state change to I/O subsystem
device-controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
Data transfer
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time

STREAMS

- **STREAM** – a **full-duplex communication channel between a user-level process and a device** in Unix System V and beyond
- A STREAM consists of:
  - The user process interacts with the **stream head**. User processes communicate with the stream head using either ***read( )*** and ***write( )***
  - The device driver interacts with the **device end**.
  - zero or more **stream modules** between
- Each module contains a **read queue** and a **write queue**
- Message passing is used to communicate between queues
  - **Flow control** can be optionally supported. Without flow control, data is passed along as soon as it is ready.

The STREAMS Structure



Improving Performance

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Use smarter hardware devices
- Balance CPU, memory, bus, and I/O performance for highest throughput

End of Lecture

- **Summary**
  - Overview
  - I/O Hardware
  - Application I/O Interface
  - Kernel I/O Subsystem
  - Streams
  - Performance

- **Reading**
  - Textbook 9th edition, **chapter 13 of the module textbook**