OPERATING SYSTEMS

Input - Output Management and Security - 1,2

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

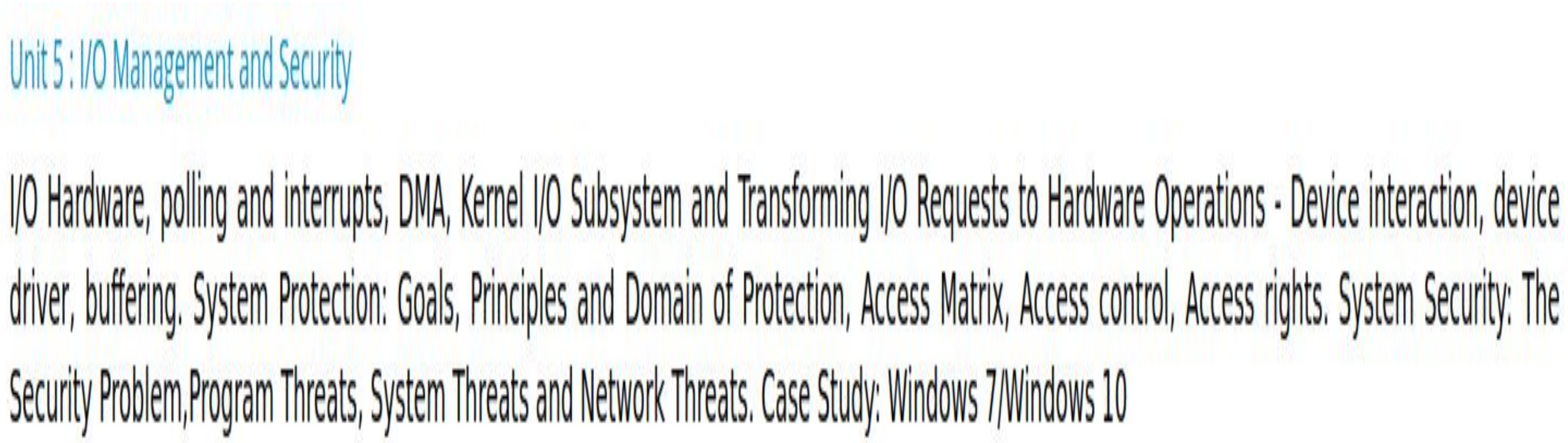
OPERATING SYSTEMS

I/O Hardware, Polling and Interrupts, DMA

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

OPERATING SYSTEMS

Course Syllabus - Unit 5

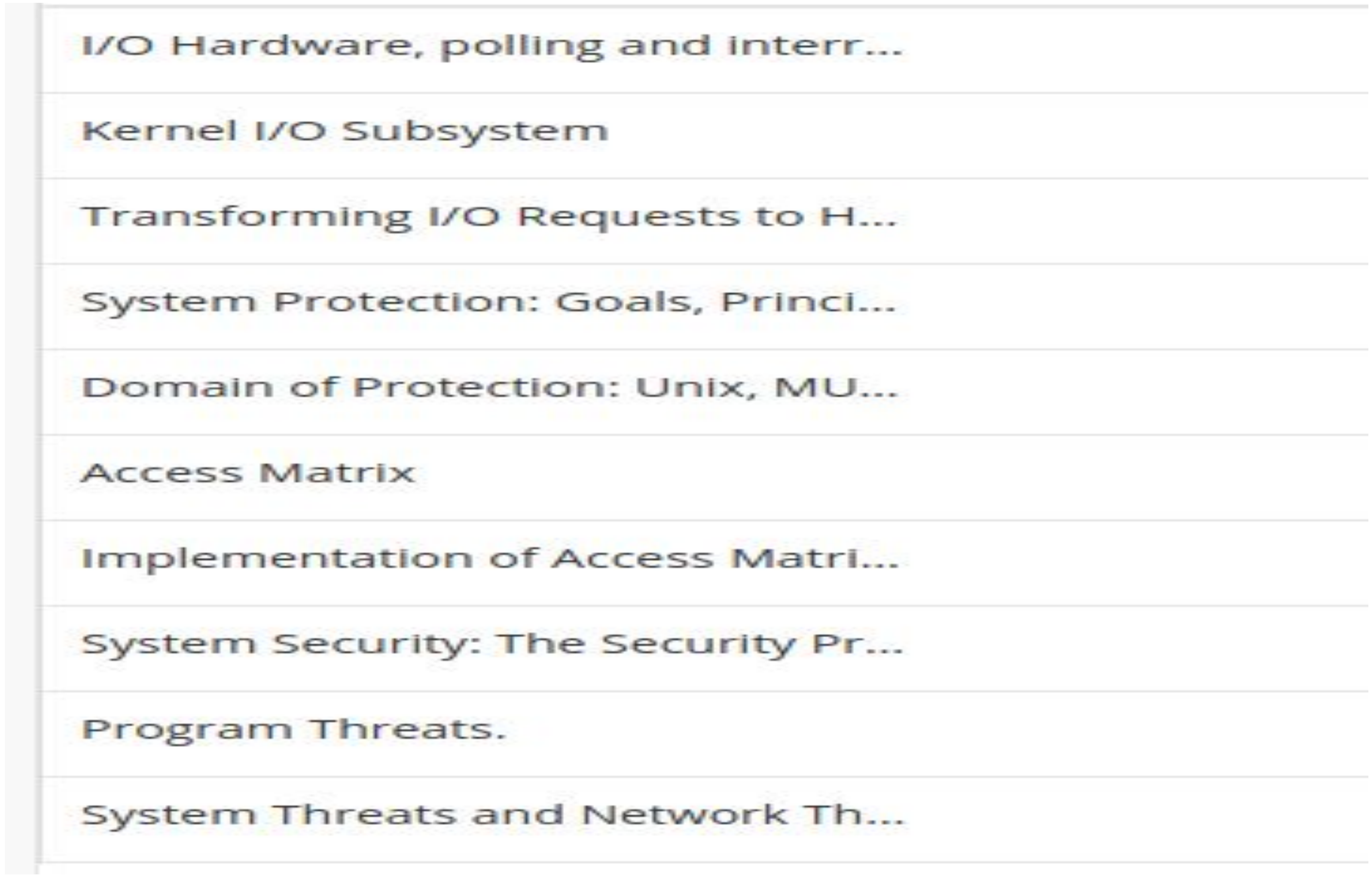


Unit 5 : I/O Management and Security

I/O Hardware, polling and interrupts, DMA, Kernel I/O Subsystem and Transforming I/O Requests to Hardware Operations - Device interaction, device driver, buffering. System Protection: Goals, Principles and Domain of Protection, Access Matrix, Access control, Access rights. System Security: The Security Problem,Program Threats, System Threats and Network Threats. Case Study: Windows 7/Windows 10

OPERATING SYSTEMS

Course Outline



I/O Hardware, polling and interr...

Kernel I/O Subsystem

Transforming I/O Requests to H...

System Protection: Goals, Princi...

Domain of Protection: Unix, MU...

Access Matrix

Implementation of Access Matri...

System Security: The Security Pr...

Program Threats.

System Threats and Network Th...

Topic Outline

- Overview
- I/O Hardware
- Polling
- Interrupts
- Direct Memory Access
- I/O Hardware Summary

Overview

- 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.
- The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices.
- The control of devices connected to the computer is a major concern of operating-system designers.
- Because I/O devices vary so widely in their function and speed , varied methods are needed to control them.
- These methods form the I/O subsystem of the kernel, which separates the rest of the kernel from the complexities of managing I/O devices.

Overview

- I/O -device technology exhibits two conflicting trends.
    - Standardization of software and hardware interfaces.
    - Increasingly broad variety of I/O devices.

- The basic I/O hardware elements, such as ports, buses, and device controllers, accommodate a wide variety of I/O devices.

- To encapsulate the details and variety of different devices, the kernel of an operating system is structured to use device-driver modules.

- The device drivers present a uniform device-access interface to the I/O subsystem, much as system calls provide a standard interface between the application and the operating system

I/O Hardware

- The general categories of storage devices (disks, tapes), transmission devices - network connections, Bluetooth, and human-interface devices - screen, keyboard, mouse, audio in and out

- Other Devices are more specialized, such as those involved in the steering of a jet.

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

- The device communicates with the machine via a connection point, or port—for example, a serial port.

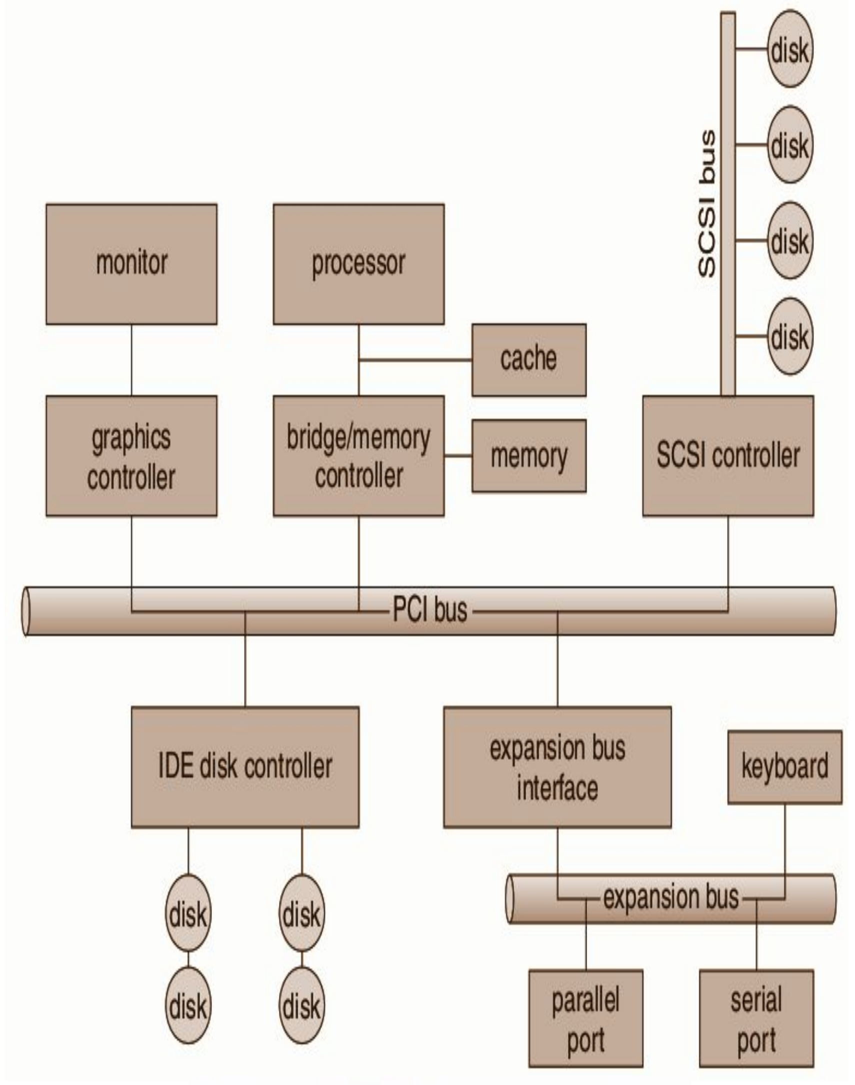
- If devices share a common set of wires, the connection is called a **Bus**. A bus is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires.

I/O Hardware

- When device A has a cable that plugs into device B, and device B has a cable that plugs into device C, and device C plugs into a port on the computer, this arrangement is called a **daisy chain**.

- A daisy chain usually operates as a **Bus**

- Buses are used widely in computer architecture and vary in their signaling methods, speed, throughput, and connection methods.



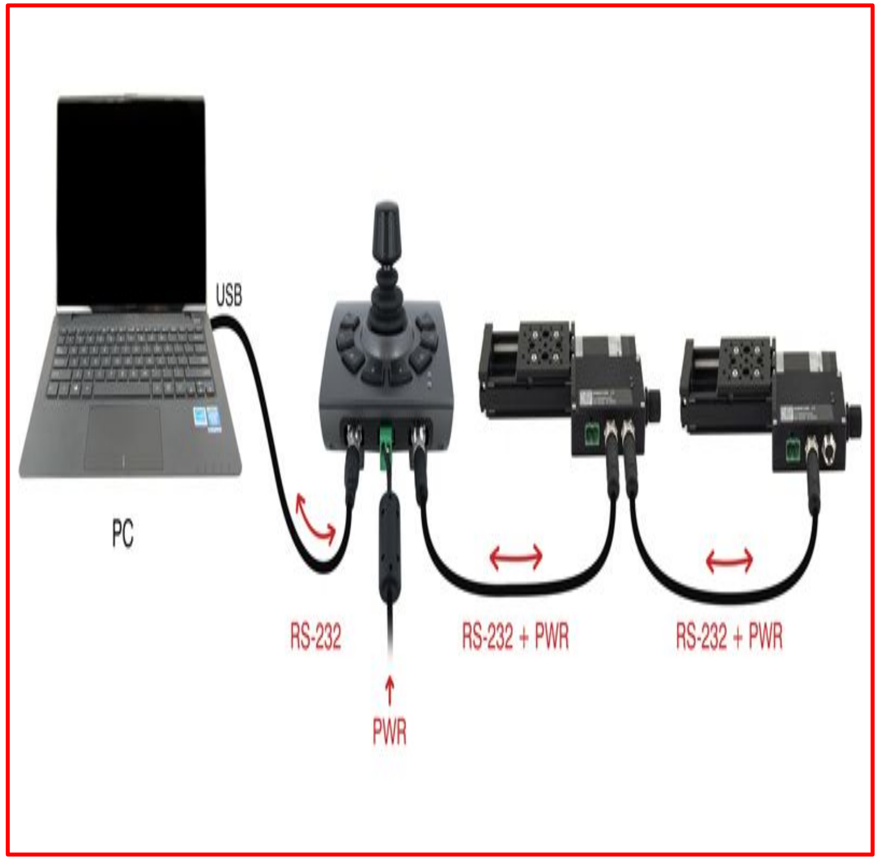
A typical PC bus structure.

I/O Hardware

- When device A has a cable that plugs into device B, and device B has a cable that plugs into device C, and device C plugs into a port on the computer, this arrangement is called a **daisy chain**.

- A daisy chain usually operates as a **Bus**

- Buses are used widely in computer architecture and vary in their signaling methods, speed, throughput, and connection methods.



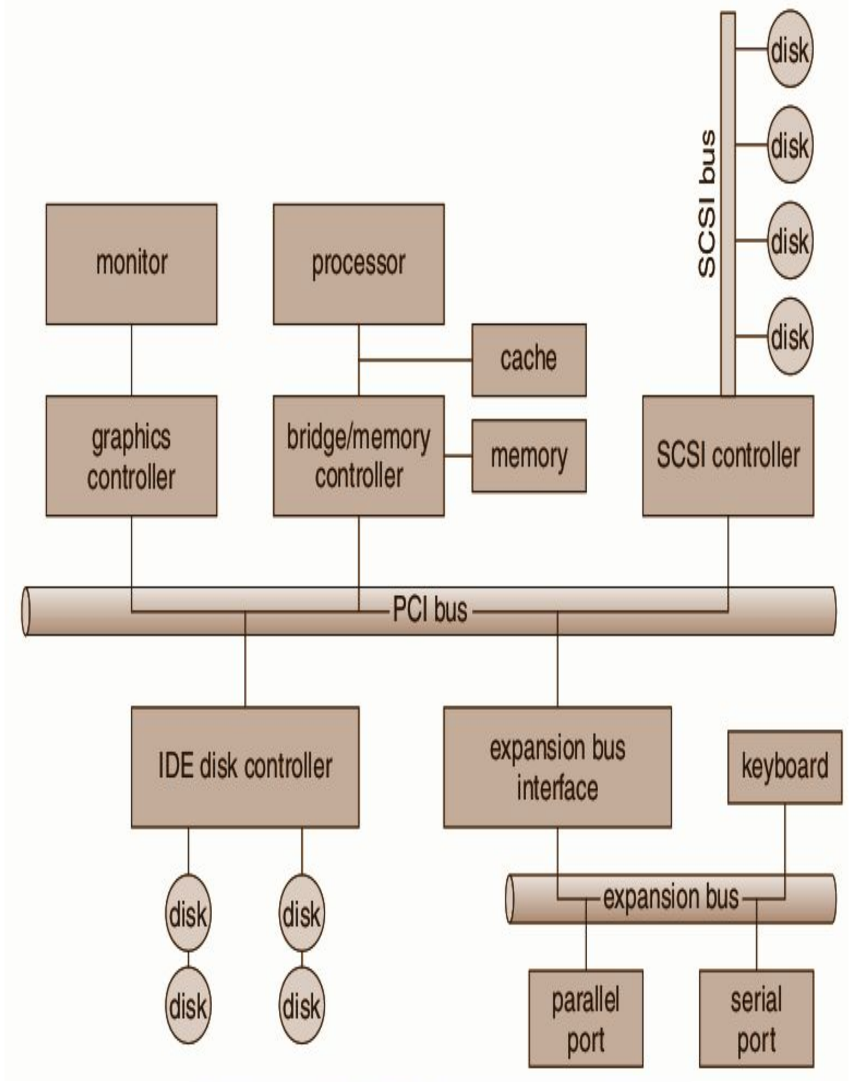
I/O Hardware

- **Peripheral Component Interconnect - PCI**

- In the figure, a PCI bus (the common PC system bus) connects the processor–memory subsystem to fast devices, and an expansion bus connects relatively slow devices, such as the keyboard and serial and USB ports.

- In the upper-right portion of the figure, four disks are connected together on a Small Computer System Interface (SCSI) bus plugged into a SCSI controller.

- Other common buses used to interconnect main parts of a computer include PCI Express (PCIe), with throughput of up to 16 GB per second, and HyperTransport, with throughput of up to 25 GB per second.



A typical PC bus structure.

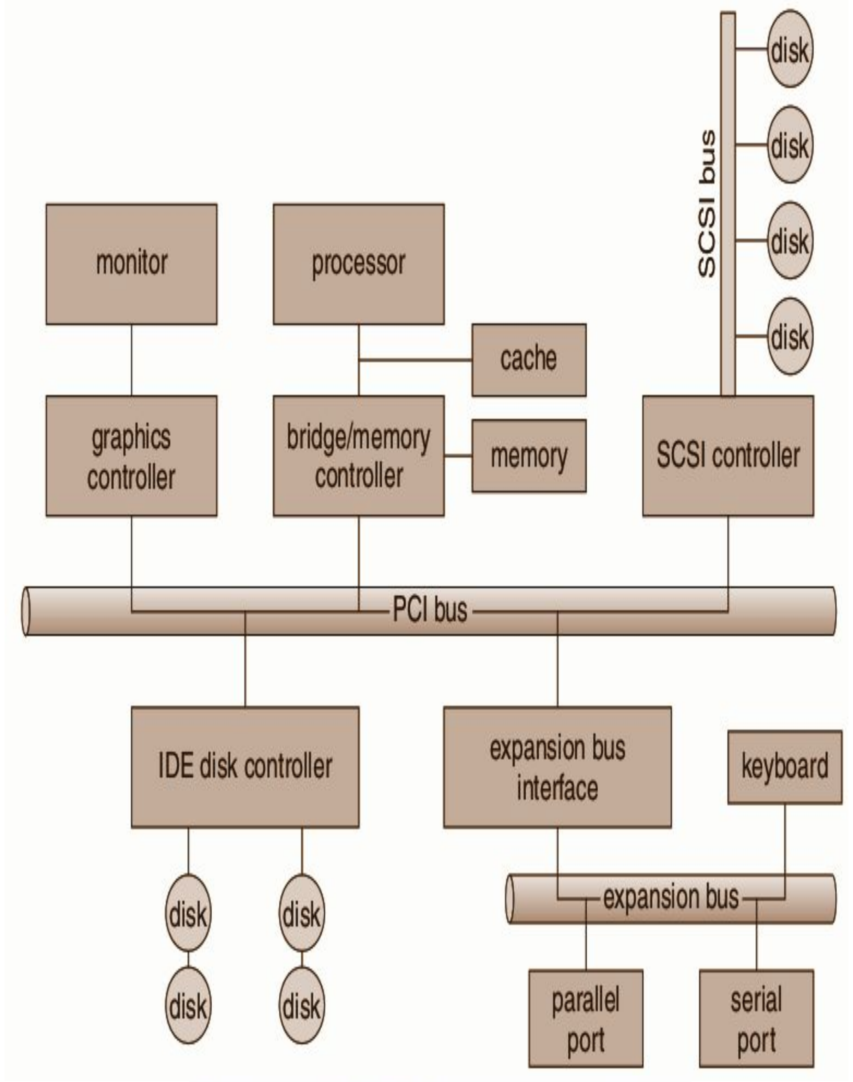
I/O Hardware

- A controller is a collection of electronics that can operate a port, a bus, or a device.

- A serial-port controller is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port.

- SCSI typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages. Some devices have their own built-in controllers

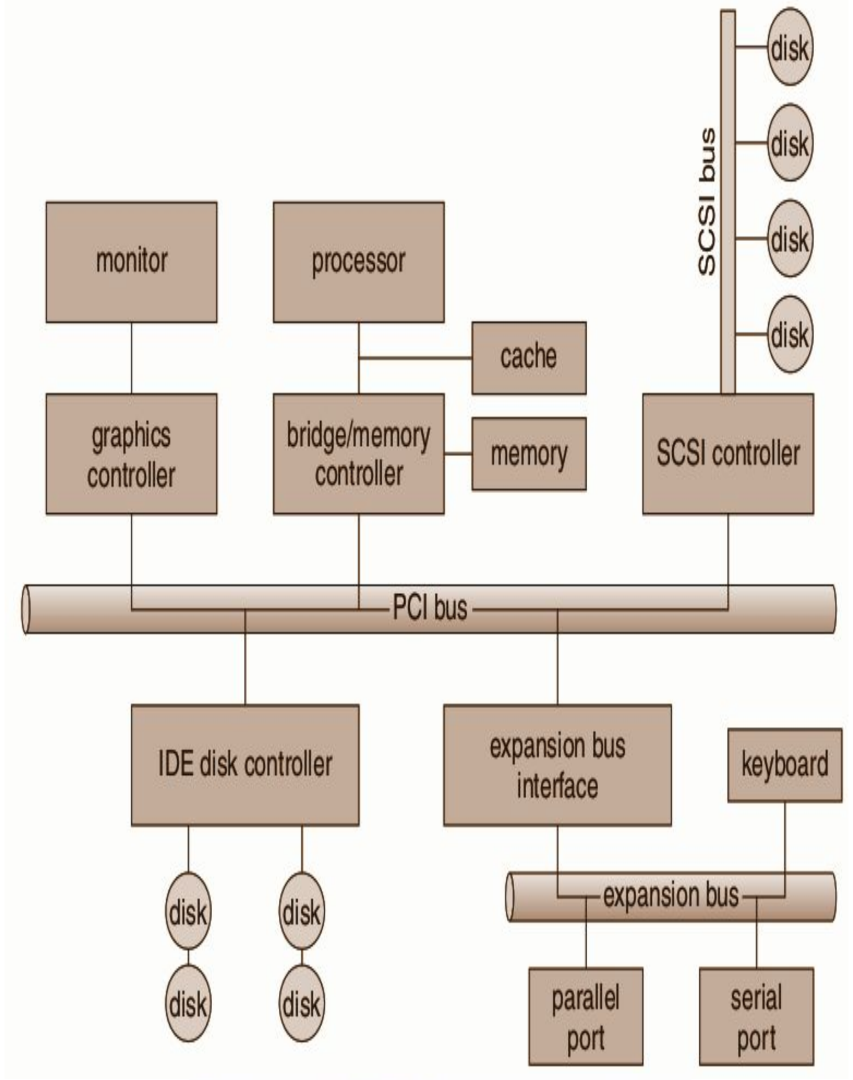
- Other board is the disk controller. It implements the disk side of the protocol for some kind of connection— SCSI or Serial Advanced Technology Attachment (SATA), for instance. It has microcode and a processor to do many tasks, such as bad-sector mapping, prefetching, buffering, and caching.



A typical PC bus structure.

I/O Hardware

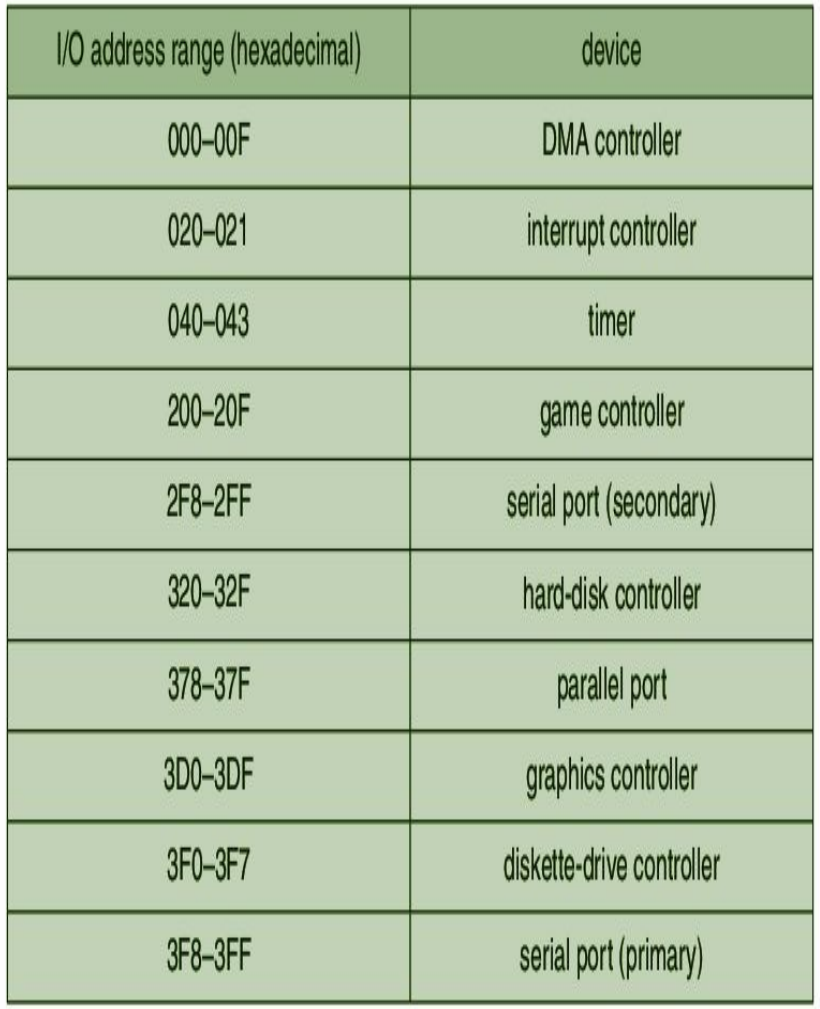
- **How can the processor give commands and data to a controller to accomplish an I/O transfer ?**
- The controller has one or more registers for data and control signals.
- The processor communicates with the controller by reading and writing bit patterns in these registers.
- One way in which this communication can occur is through the use of special I/O instructions that specify the transfer of a byte or word to an I/O port address.
- I/O can be performed using **Memory mapped I/O and I/O mapped I/O**



A typical PC bus structure.

I/O Hardware

- **An I/O port typically consists of four registers, called the Status, Control, Data-in, and Data-out 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 into for example Half duplex, Full Duplex

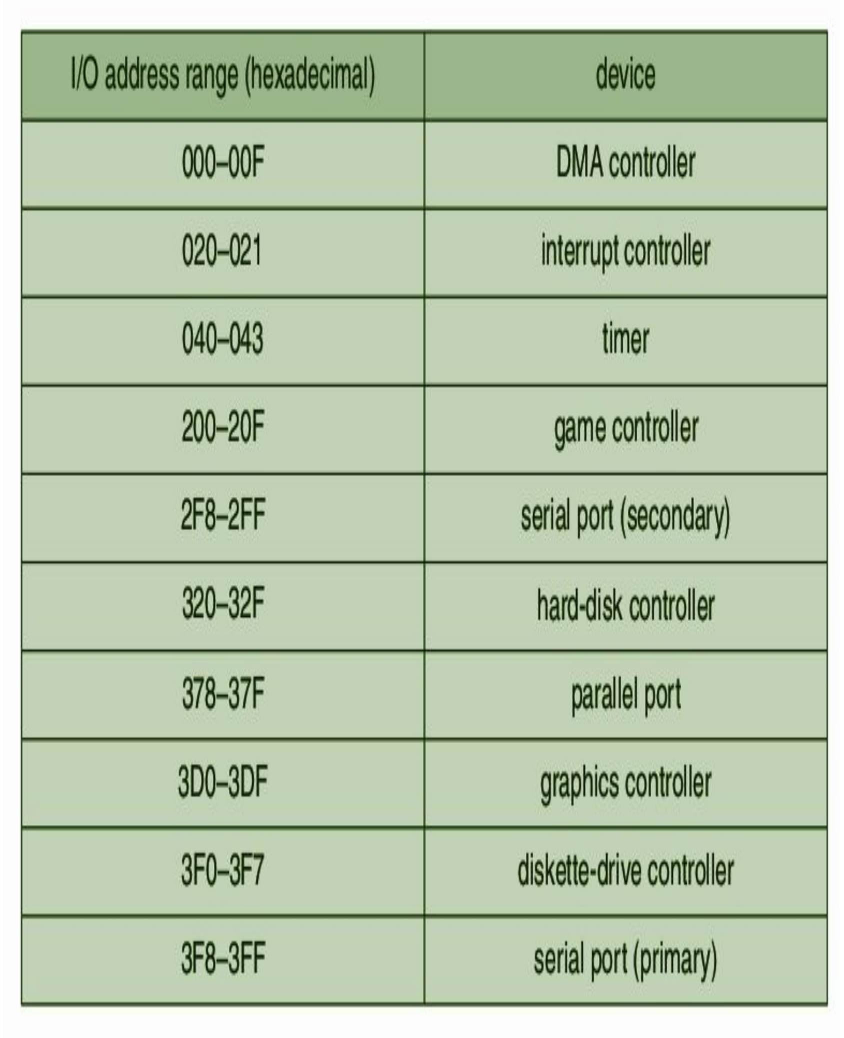


I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Device I/O port locations on PCs (partial).

I/O Hardware

- The data registers are typically 1 to 4 bytes in size.
- Some controllers have **FIFO chips** that can hold several bytes of input or output data to expand the capacity of the controller beyond the size of the data register.
- A FIFO chip can hold a small burst of data until the device or host is able to receive the data



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

Device I/O port locations on PCs (partial).

Polling

- The complete protocol for interaction between the host and a controller can be intricate, but the basic handshaking notion is simple.

- For example, the host writes output through a port, coordinating with the controller by handshaking as follows.
  - The host repeatedly reads the busy bit until that bit becomes clear.
  - The host sets the write bit in the command register and writes a byte into the data-out register.
  - The host sets the command-ready bit.
  - When the controller notices that the command-ready bit is set, it sets the busy bit.
  - The controller reads the command register and sees the write command. It reads the data-out register to get the byte and does the I/O to the device.
  - The controller clears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished.

Polling

- In first step, the host is **Busy-waiting or Polling**: it is in a loop, reading the status register over and over until the busy bit becomes clear.

- In many computer architectures, three CPU -instruction cycles are sufficient to poll a device:
  - read a device register,
  - logical anding to extract a status bit
  - branch if not zero.

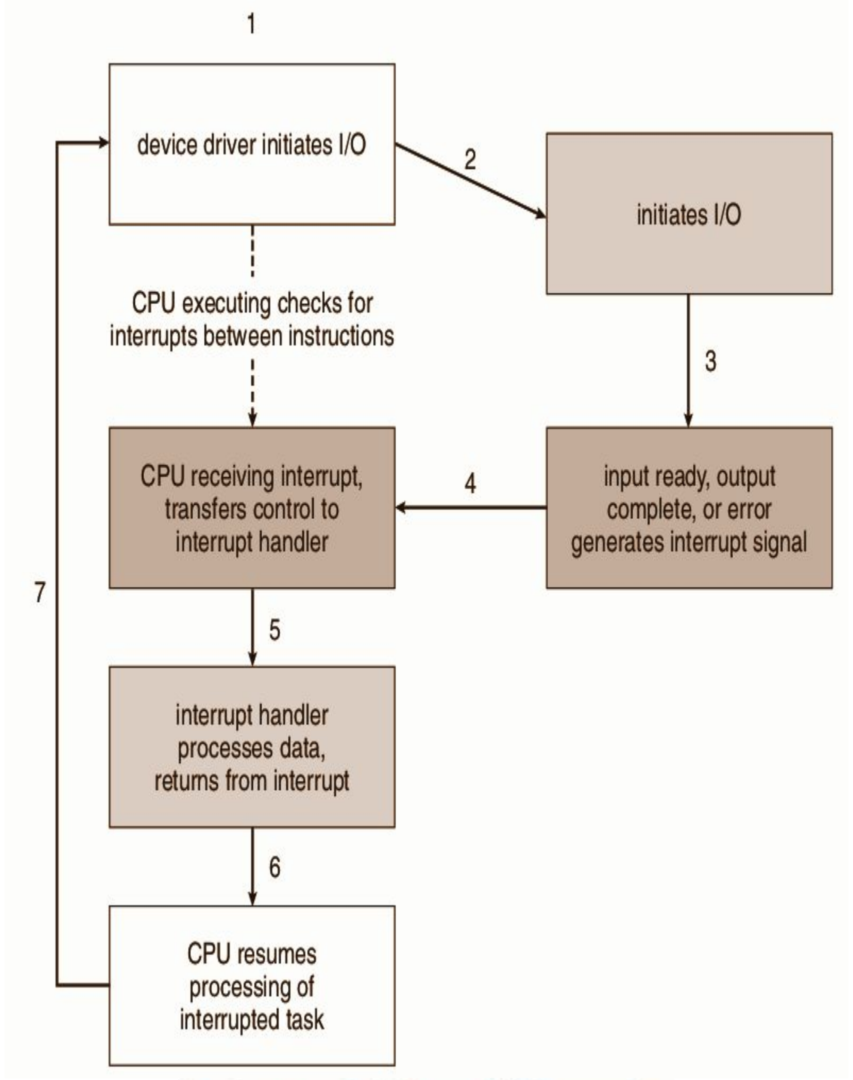
- But polling becomes inefficient when it is attempted repeatedly yet rarely finds a device ready for service, while other useful CPU processing remains undone

- In such instances, it may be more efficient to arrange for the hardware controller to notify the CPU when the device becomes ready for service, rather than to require the CPU to poll repeatedly for an I/O completion.

- The hardware mechanism that enables a device to notify the CPU is called an **Interrupt**.

Interrupts

- The basic interrupt mechanism works as follows.
- The CPU hardware has a wire called the interrupt-request line that the CPU senses after executing every instruction.
- When the CPU detects that a controller has asserted a signal on the interrupt-request line, the CPU performs a state save and jumps to the interrupt-handler routine at a fixed address in memory.
- The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt.

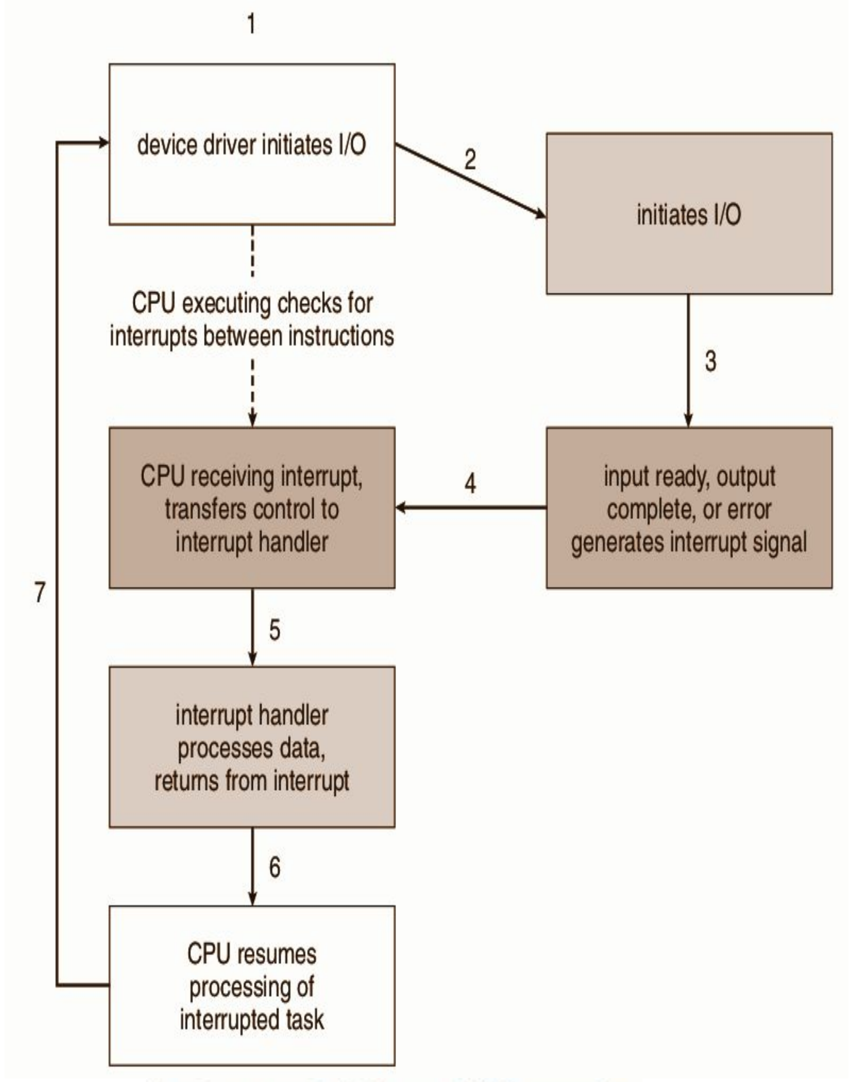

1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
5
interrupt handler processes data, returns from interrupt
6
CPU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

Interrupt-driven I/O cycle.

Interrupts

- We say that the device controller raises an interrupt by asserting a signal on the interrupt request line, the CPU catches the interrupt and dispatches it to the handler, and the handler clears the interrupt by servicing the device.

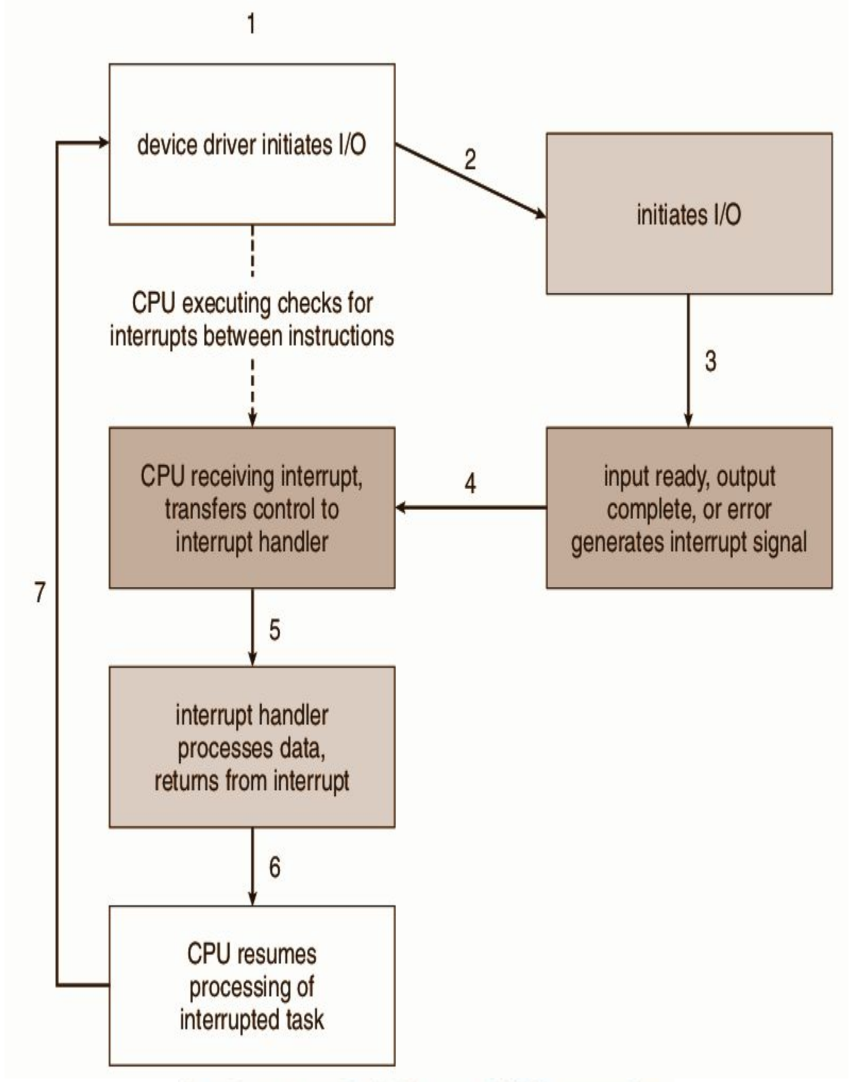
- Figure summarizes the interrupt-driven I/O cycle.


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
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task
7

Interrupt-driven I/O cycle.

Interrupts

- In a modern operating system, however, we need more sophisticated interrupt-handling features.
  - We need the ability to defer interrupt handling during critical processing.
  - We need an efficient way to dispatch to the proper interrupt handler for a device without first polling all the devices to see which one raised the interrupt.
  - We need multi level interrupts, so that the operating system can distinguish between high and low-priority interrupts and can respond with the appropriate degree of urgency.


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
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task
7

Interrupt-driven I/O cycle.

Interrupts

- In modern computer hardware, these three features are provided by the CPU and by the **interrupt-controller hardware**.
- Most CPUs have two interrupt request lines.
  - One is the **nonmaskable** interrupt, which is reserved for events such as unrecoverable memory errors.
  - The second interrupt line is **maskable**: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted.
- The maskable interrupt is used by device controllers to request service.
- The interrupt mechanism accepts an address—a number that selects a specific interrupt-handling routine from a small set.
- In most architectures, this address is an offset in a table called the **interrupt vector**. This vector contains the memory addresses of specialized interrupt handlers.

Interrupts

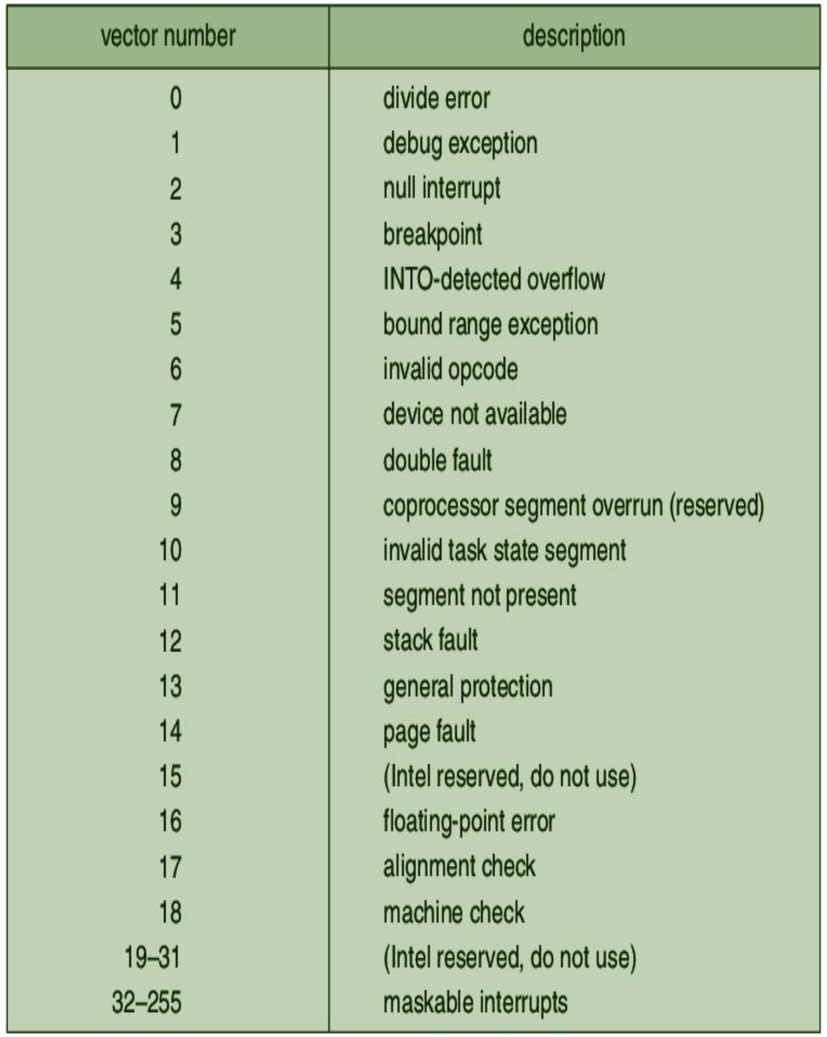
- A common way to solve the problem of multiple devices and Interrupt handlers is to use **interrupt chaining**, in which each element in the interrupt vector points to the head of a list of interrupt handlers.

- When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.

OPERATING SYSTEMS

Interrupts

- Figure illustrates the design of the interrupt vector for the Intel Pentium processor.
- The events from 0 to 31, which are nonmaskable, are used to signal various error conditions.
- The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.
- The interrupt mechanism also implements a system of interrupt priority levels.
- These levels enable the CPU to defer the handling of low-priority interrupts without masking all interrupts and makes it possible for a high- priority interrupt to preempt the execution of a low-priority interrupt.

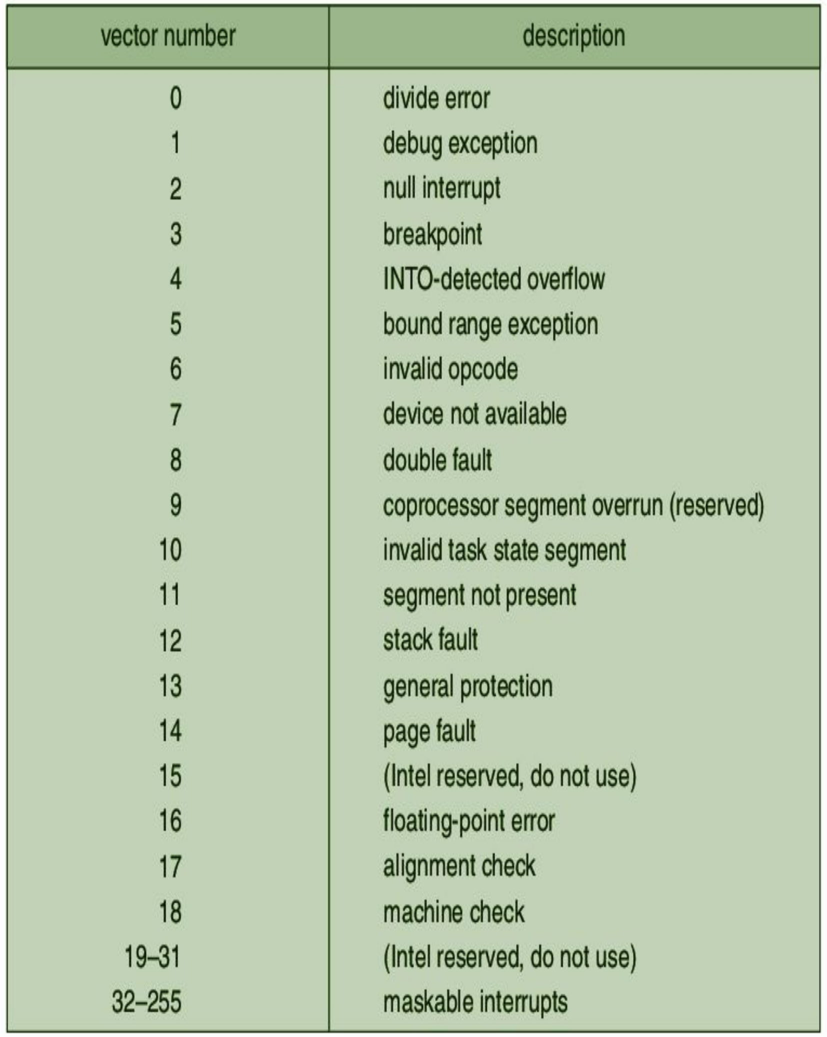


vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
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

Intel Pentium processor event-vector table.

Interrupts

- A modern operating system interacts with the interrupt mechanism in several ways.
- At boot time, the operating system probes the hardware buses to determine what devices are present and installs the corresponding interrupt handlers into the interrupt vector.
- During I/O , the various device controllers raise interrupts when they are ready for service. These interrupts signify that output has completed, or that input data are available, or that a failure has been detected.
- The interrupt mechanism is also used to handle a wide variety of exceptions, such as dividing by 0, accessing a protected or nonexistent memory address, or attempting to execute a privileged instruction from user mode.



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

Intel Pentium processor event-vector table.

Interrupts

- An operating system has other good uses for an efficient hardware and software mechanism that saves a small amount of processor state and then calls a privileged routine in the kernel.

- For example, many operating systems use the interrupt mechanism for virtual memory paging.

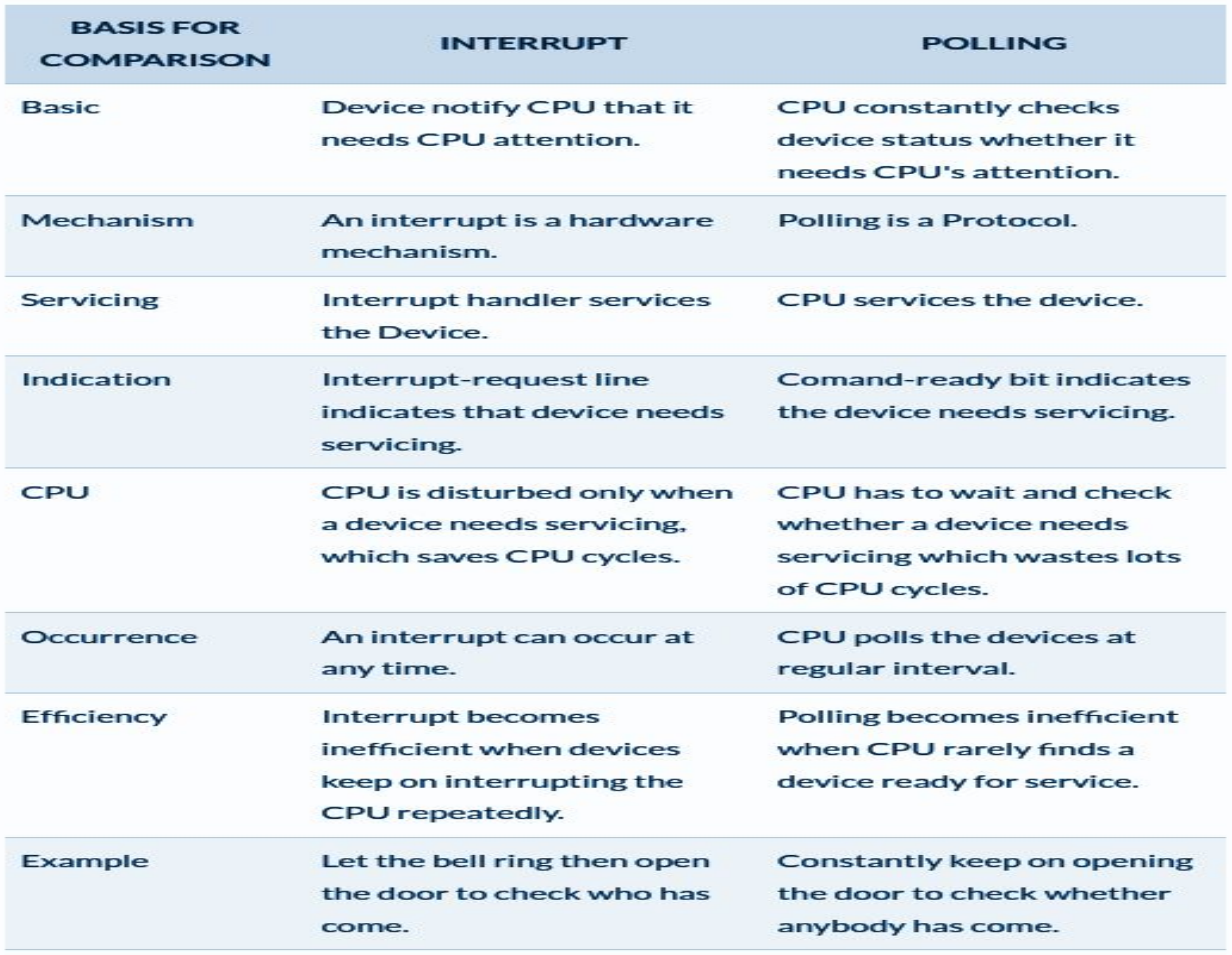
- A page fault is an exception that raises an interrupt.

- The interrupt suspends the current process and jumps to the page-fault handler in the kernel. This handler saves the state of the process, moves the process to the wait queue, performs page-cache management, schedules an I/O operation to fetch the page, schedules another process to resume execution, and then returns from the interrupt

Interrupts

- Another example is found in the implementation of system calls.
- Usually, a program uses library calls to issue system calls.
- The library routines check the arguments given by the application, build a data structure to convey the arguments to the kernel, and then execute a special instruction called a **software interrupt, or trap.**
- This instruction has an operand that identifies the desired kernel service. When a process executes the trap instruction, the interrupt hardware saves the state of the user code, switches to kernel mode, and dispatches to the kernel routine that implements the requested service.
- The trap is given a relatively low interrupt priority compared with those assigned to device interrupts—executing a system call on behalf of an application is less urgent than servicing a device controller before its FIFO queue overflows and loses data.

Interrupt Versus Polling



BASIS FOR COMPARISON	INTERRUPT	POLLING
Basic	Device notify CPU that it needs CPU attention.	CPU constantly checks device status whether it needs CPU's attention.
Mechanism	An interrupt is a hardware mechanism.	Polling is a Protocol.
Servicing	Interrupt handler services the Device.	CPU services the device.
Indication	Interrupt-request line indicates that device needs servicing.	Comand-ready bit indicates the device needs servicing.
CPU	CPU is disturbed only when a device needs servicing, which saves CPU cycles.	CPU has to wait and check whether a device needs servicing which wastes lots of CPU cycles.
Occurrence	An interrupt can occur at any time.	CPU polls the devices at regular interval.
Efficiency	Interrupt becomes inefficient when devices keep on interrupting the CPU repeatedly.	Polling becomes inefficient when CPU rarely finds a device ready for service.
Example	Let the bell ring then open the door to check who has come.	Constantly keep on opening the door to check whether anybody has come.

Direct Memory Access

- 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 one byte at a time —a process termed programmed I/O (PIO).

- Many computers avoid burdening the main CPU with PIO by offloading some of this work to a special-purpose processor called a **Direct Memory Access** (DMA) controller.

- To initiate a DMA transfer, the host writes a DMA command block into memory.

- This block contains 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 all modern computers, from smartphones to mainframes.

Direct Memory Access

- Direct memory access (DMA) is a **mode of data transfer** between the memory and I/O devices. This happens **without the involvement** of the processor. We have two other methods of data transfer, **Programmed I/O** and **Interrupt driven I/O**. Let's revise each along with their drawbacks.

- In **programmed I/O**, the processor keeps on scanning whether any device is ready for data transfer. If an I/O device is ready, the processor **fully dedicates** itself in transferring the data between I/O and memory. It transfers data at a **high rate, but it can't get involved in any other activity** during data transfer. This is the major **drawback** of programmed I/O.

- In **Interrupt driven I/O**, whenever the device is ready for data transfer, then it raises an **interrupt to processor**. Processor completes executing its ongoing instruction and saves its current state. It then switches to data transfer which causes a **delay**. Here, the processor doesn't keep scanning for peripherals ready for data transfer. But, it is **fully involved** in the data transfer process. So, it is also not an effective way of data transfer

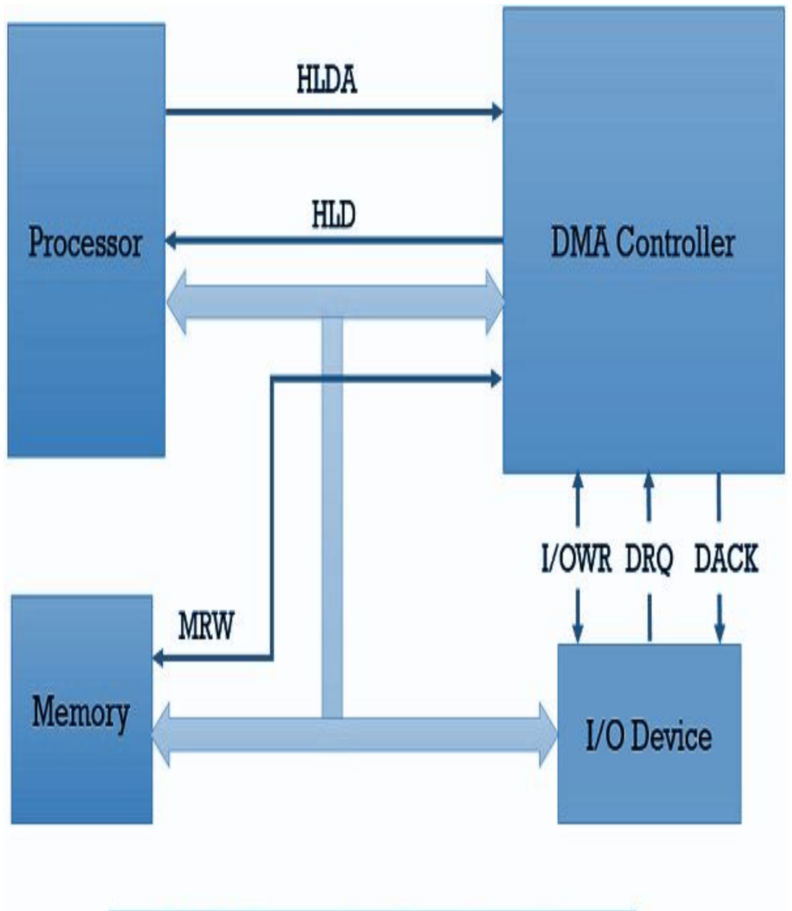
- The above two modes of data transfer are not useful for transferring a large block of data. But, the DMA controller completes this task at a faster rate and is also effective for transfer of large data block.

Direct Memory Access

- The DMA controller transfers the data in three modes:
    - **Burst Mode:** Here, once the DMA controller gains the charge of the system bus, then it releases the system bus only after completion of data transfer. Till then the CPU has to wait for the system buses.
    - **Cycle Stealing Mode:** In this mode, the DMA controller forces the CPU to stop its operation and relinquish the control over the bus for a short term to DMA controller. After the transfer of every byte, the DMA controller releases the bus and then again requests for the system bus. In this way, the DMA controller steals the clock cycle for transferring every byte.
    - **Transparent Mode:** Here, the DMA controller takes the charge of system bus only if the processor does not require the system bus.

Direct Memory Access

- DMA controller is a hardware unit that allows I/O devices to access memory directly without the participation of the processor.
  - Whenever an I/O device wants to transfer the data to or from memory, it sends the DMA request (**DRQ**) to the DMA controller. DMA controller accepts this DRQ and asks the CPU to hold for a few clock cycles by sending it the Hold request (**HLD**).
  - CPU receives the Hold request (HLD) from DMA controller and relinquishes the bus and sends the Hold acknowledgement (**HLDA**) to DMA controller.
  - After receiving the Hold acknowledgement (HLDA), DMA controller acknowledges I/O device **(DACK)** that the data transfer can be performed and DMA controller takes the charge of the system bus and transfers the data to or from memory.
  - When the data transfer is accomplished, the DMA raise an **interrupt** to let know the processor that the task of data transfer is finished and the processor can take control over the bus again and start processing where it has left.



DMA Controller Data Transfer

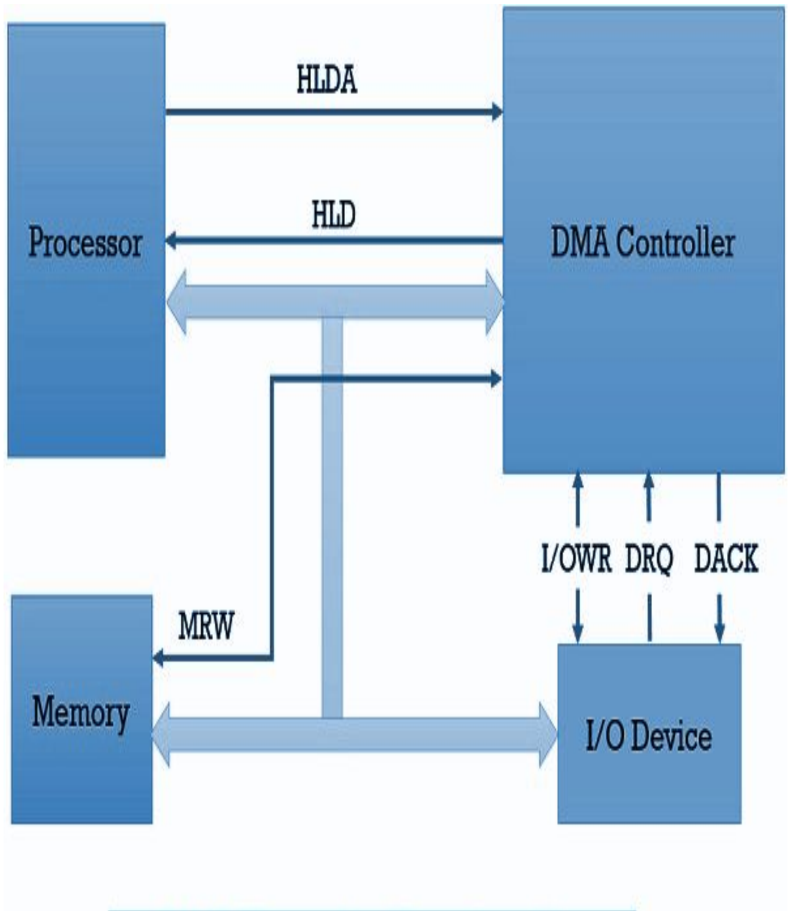
Direct Memory Access

Advantages:

- Transferring the data without the involvement of the processor will speed up the read-write task.
- DMA reduces the clock cycle requires to read or write a block of data.
- Implementing DMA also reduces the overhead of the processor.

Disadvantages

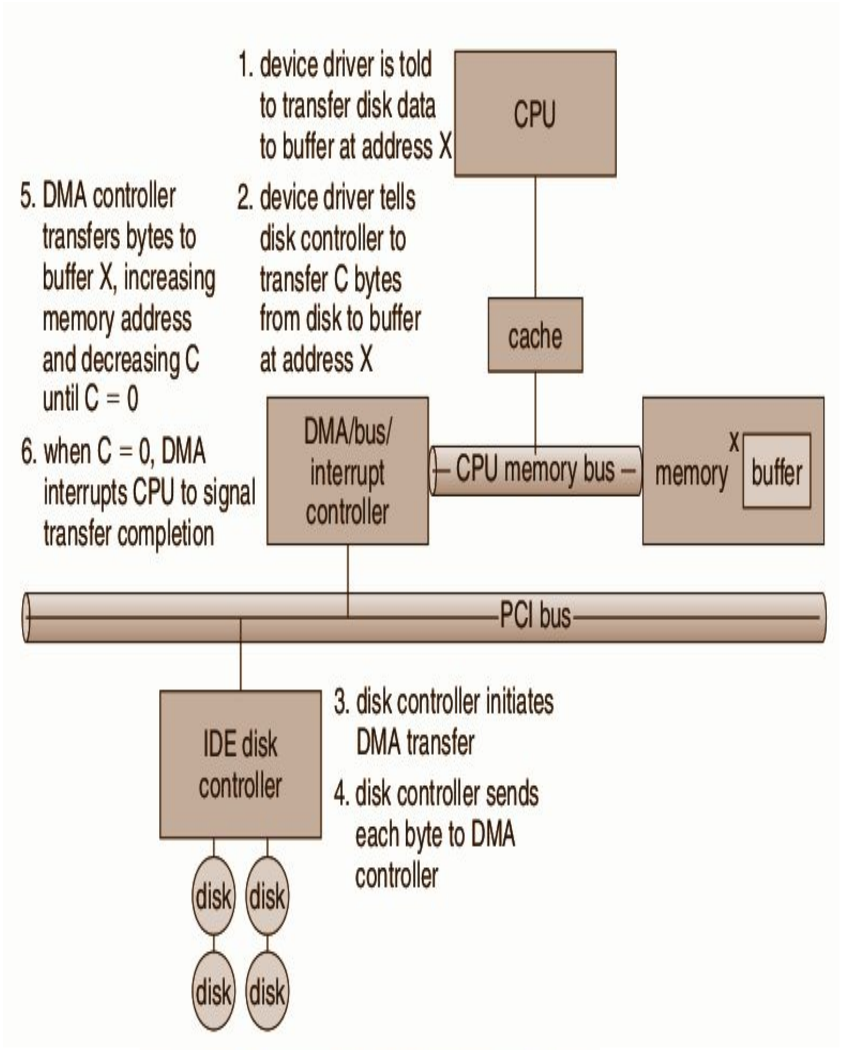
- As it is a hardware unit, it would cost to implement a DMA controller in the system.
- Cache coherence problem can occur while using DMA controller.



DMA Controller Data Transfer

Direct Memory Access

- When the entire transfer is finished, the DMA controller interrupts the CPU .
- Although this **cycle stealing** can slow down the CPU computation, offloading the data-transfer work to a DMA controller generally improves the total system performance.
- Some computer architectures use physical memory addresses for DMA , but others perform direct virtual memory access (DVMA), using virtual addresses that undergo translation to physical addresses.
- The trend in general-purpose operating systems is to protect memory and devices so that the system can try to guard against erroneous or malicious applications.


1. device driver is told to transfer disk data to buffer at address X
CPU
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
cache
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
DMA/bus/ interrupt controller
CPU memory bus
memory
X
buffer
PCI bus
IDE disk controller
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
disk
disk
disk
disk

Steps in a DMA transfer.

Direct Memory Access - Summary

- DMA is an abbreviation of direct memory access.
- DMA is a method of data transfer between main memory and peripheral devices.
- The hardware unit that controls the DMA transfer is a DMA controller.
- DMA controller transfers the data to and from memory without the participation of the processor.
- The processor provides the start address and the word count of the data block which is transferred to or from memory to the DMA controller and frees the bus for DMA controller to transfer the block of data.
- DMA controller transfers the data block at the faster rate as data is directly accessed by I/O devices and is not required to pass through the processor which save the clock cycles.
- DMA controller transfers the block of data to and from memory in three modes burst mode, cycle steal mode and transparent mode.
- DMA can be configured in various ways it can be a part of individual I/O devices, or all the peripherals attached to the system may share the same DMA controller.
- The DMA controller is a convenient mode of data transfer. It is preferred over the programmed I/O and Interrupt-driven I/O mode of data transfer

I/O Hardware Summary

- A bus
- A controller
- An I/O port and its registers
- The handshaking relationship between the host and a device controller
- The execution of this handshaking in a polling loop or via interrupts
- The offloading of this work to a DMA controller for large transfers

Topics Uncovered

- Overview
- I/O Hardware
- Polling
- Interrupts
- Direct Memory Access
- I/O Hardware Summary

Topic Outline

- Transforming I/O Requests to Hardware Operations

Transforming I/O Requests to Hardware Operations

- Users request data using file names, which must ultimately be mapped to specific blocks of data from a specific device managed by a specific device driver.
- DOS uses the colon separator to specify a particular device (e.g. C:, LPT:, etc.)
- UNIX uses a mount table to map filename prefixes (e.g. /usr) to specific mounted devices.
- Where multiple entries in the mount table match different prefixes of the filename the one that matches the longest prefix is chosen. (e.g. /usr/home instead of /usr where both exist in the mount table and both match the desired file.)

Transforming I/O Requests to Hardware Operations

- UNIX uses special device files, usually located in /dev, to represent and access physical devices directly.
  - Each device file has a major and minor number associated with it, stored and displayed where the file size would normally go.
  - The major number is an index into a table of device drivers, and indicates which device driver handles this device. (E.g. the disk drive handler.)
  - The minor number is a parameter passed to the device driver, and indicates which specific device is to be accessed, out of the many which may be handled by a particular device driver. (e.g. a particular disk drive or partition.)
- A series of lookup tables and mappings makes the access of different devices flexible, and somewhat transparent to users.

Transforming I/O Requests to Hardware Operations

- Modern operating systems gain significant flexibility from the multiple stages of lookup tables in the path between a request and a physical device controller.

- The mechanisms that pass requests between applications and drivers are general.

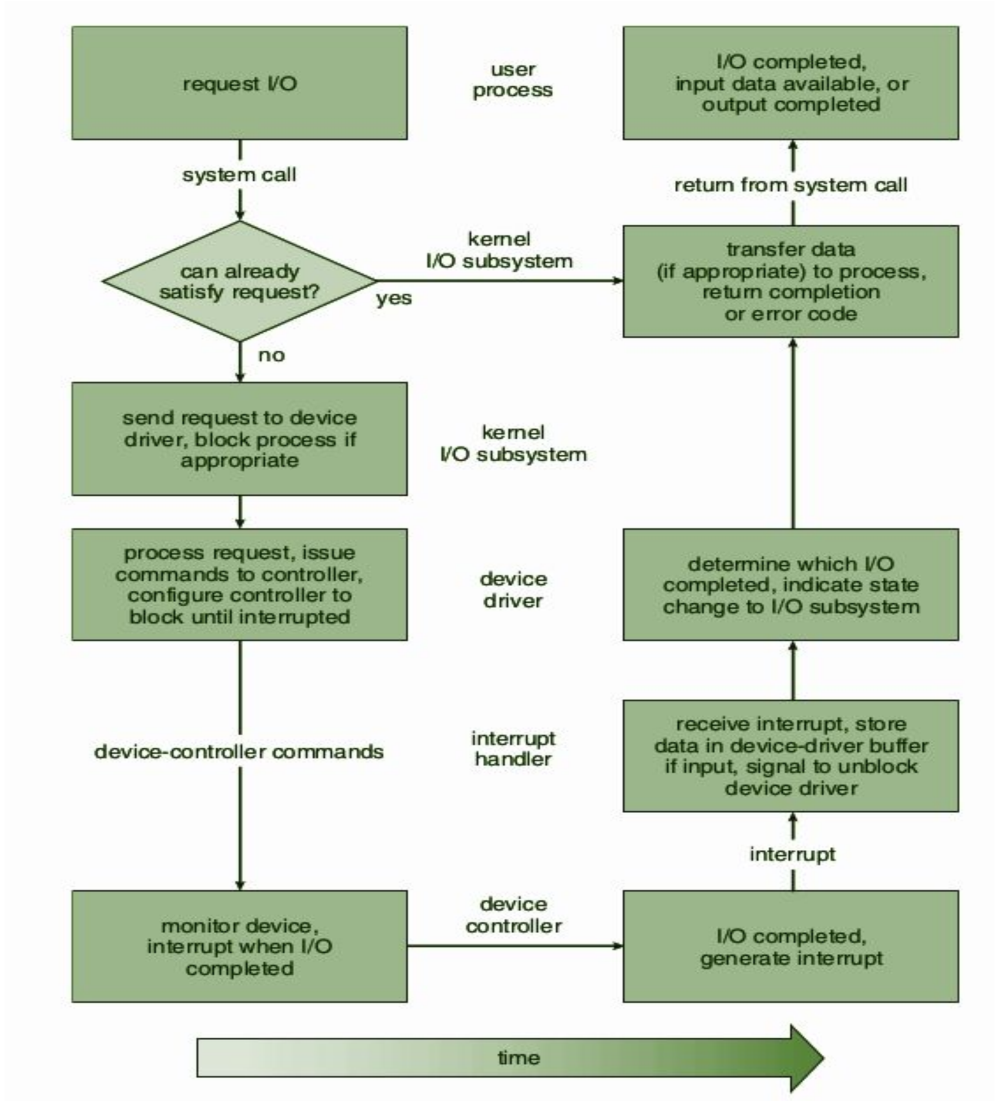
- Thus, we can introduce new devices and drivers into a computer without recompiling the kernel. In fact, some operating systems have the ability to load device drivers on demand.

- At boot time, the system first probes the hardware buses to determine what devices are present.

- It then loads in the necessary drivers, either immediately or when first required by an I/O request.

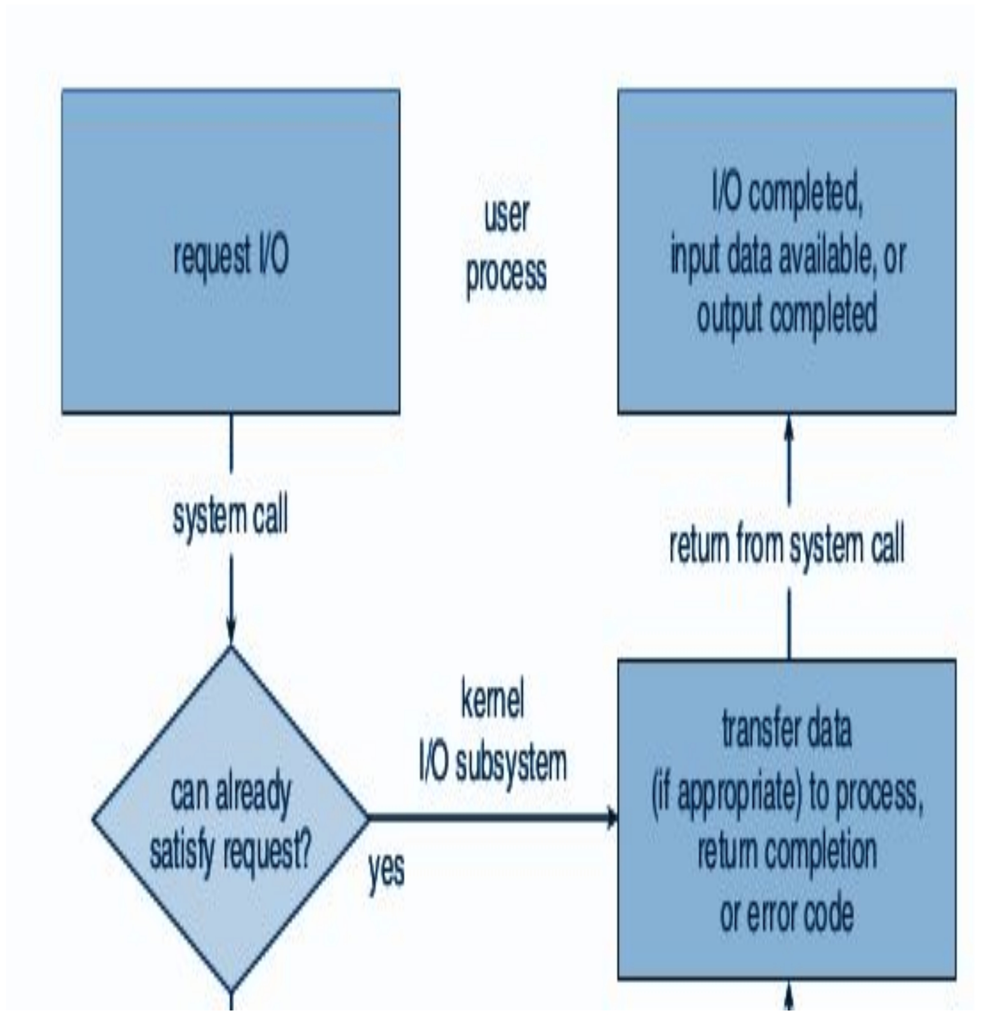
Transforming I/O Requests to Hardware Operations

- The figure suggests that an I/O operation requires a great many steps that together consume a tremendous number of CPU cycles.
- A **blocking read** will wait until there is data available (or a timeout, if any, expires), and then returns from the function call.
- A **non-blocking read** will (or at least should) always return immediately, but it might not return any data, if none is available at the moment


request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
kernel I/O subsystem
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
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time

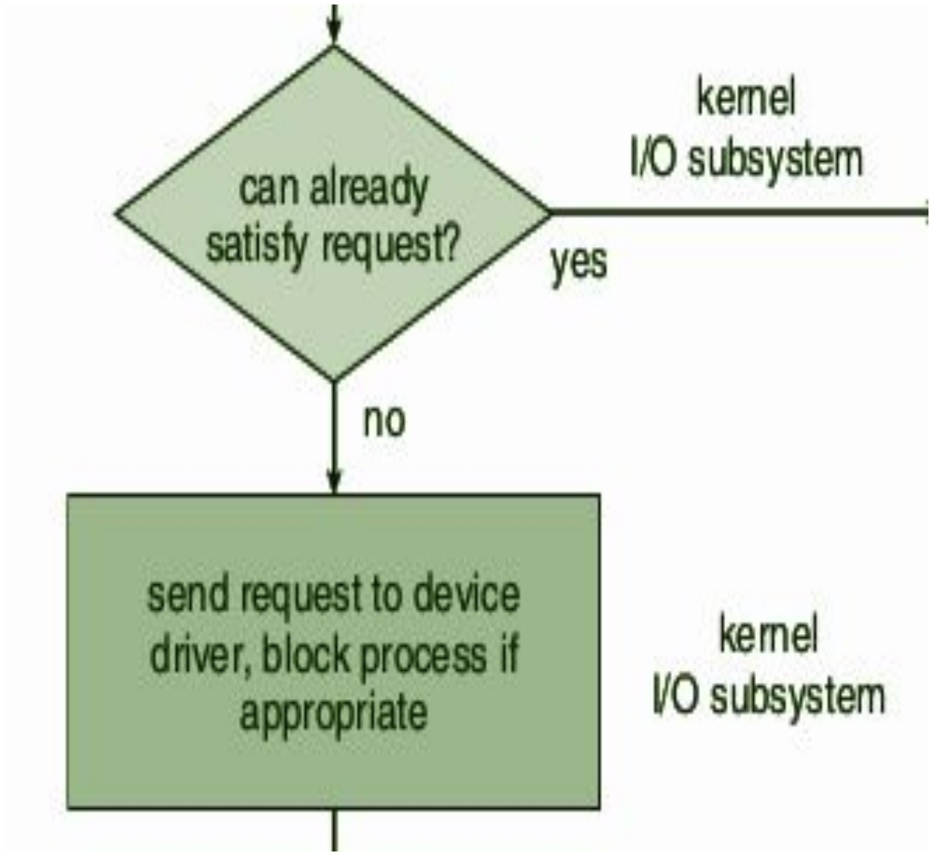
Transforming I/O Requests to Hardware Operations

- A process issues a blocking read() system call to a file descriptor of a file that has been opened previously.
- The system-call code in the kernel checks the parameters for correctness.
- In the case of input, if the data are already available in the buffer cache, the data are returned to the process, and the I/O request is completed.


request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
yes
kernel I/O subsystem
transfer data (if appropriate) to process, return completion or error code

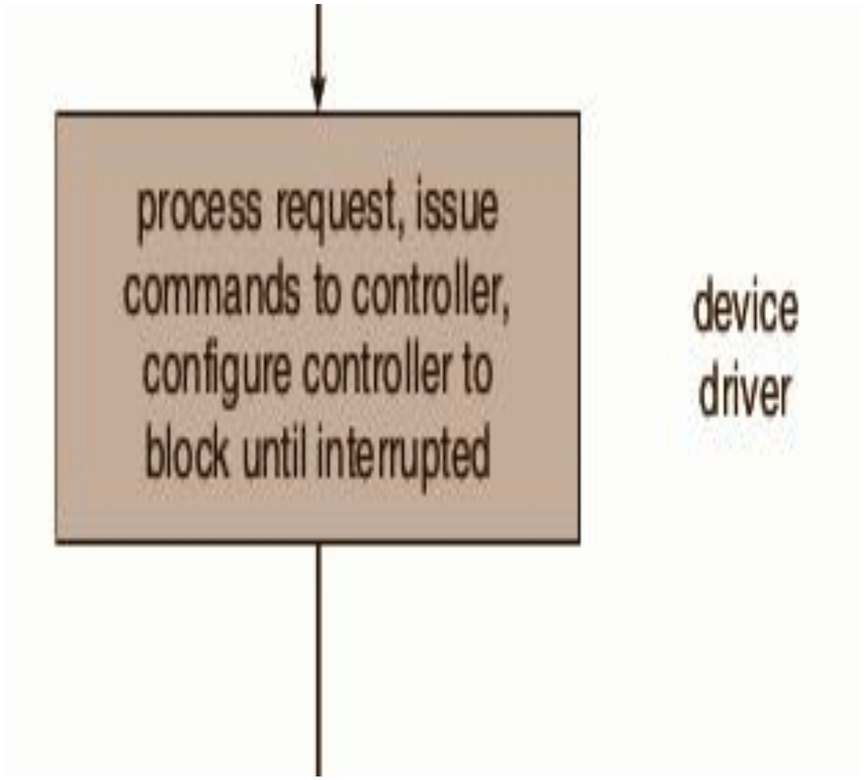
Transforming I/O Requests to Hardware Operations

- Otherwise, a physical I/O must be performed.
- The process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled.
- Eventually, the I/O subsystem sends the request to the device driver.
- Depending on the operating system, the request is sent via a subroutine call or an in-kernel message.


can already satisfy request?
yes
kernel I/O subsystem
no
send request to device driver, block process if appropriate
kernel I/O subsystem

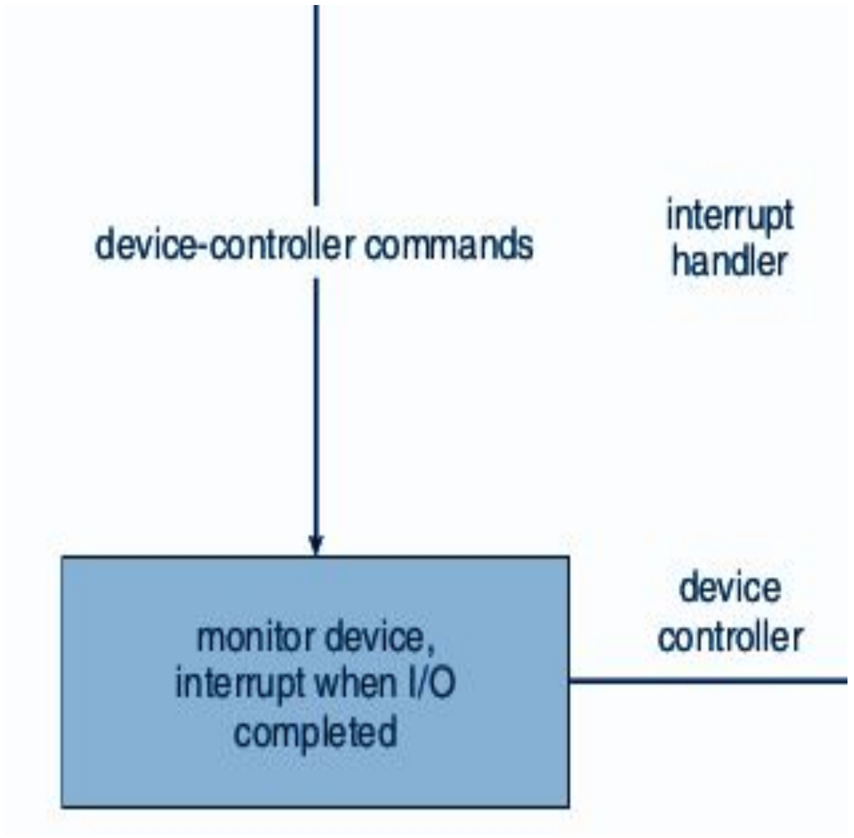
Transforming I/O Requests to Hardware Operations

- The device driver allocates kernel buffer space to receive the data and schedules the I/O .
- Eventually, the driver sends commands to the device controller by writing into the device-control registers.


process request, issue commands to controller, configure controller to block until interrupted
device driver

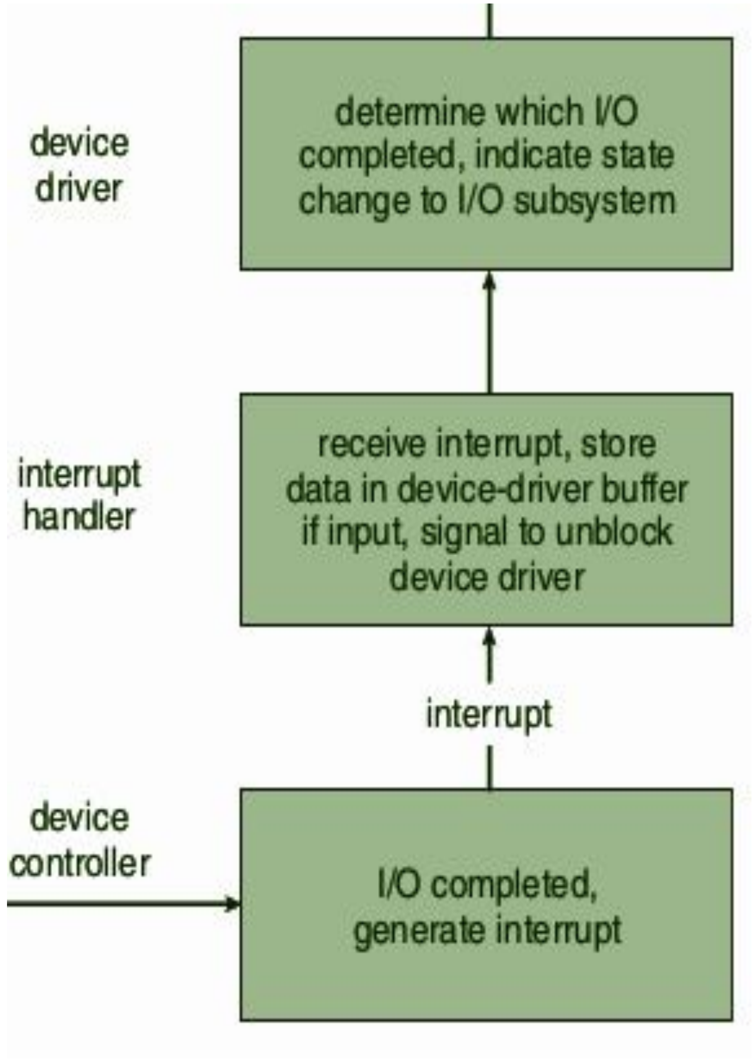
Transforming I/O Requests to Hardware Operations

- The device controller operates the device hardware to perform the data transfer.
- The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory.
- We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.


device-controller commands
interrupt handler
monitor device, interrupt when I/O completed
device controller

Transforming I/O Requests to Hardware Operations

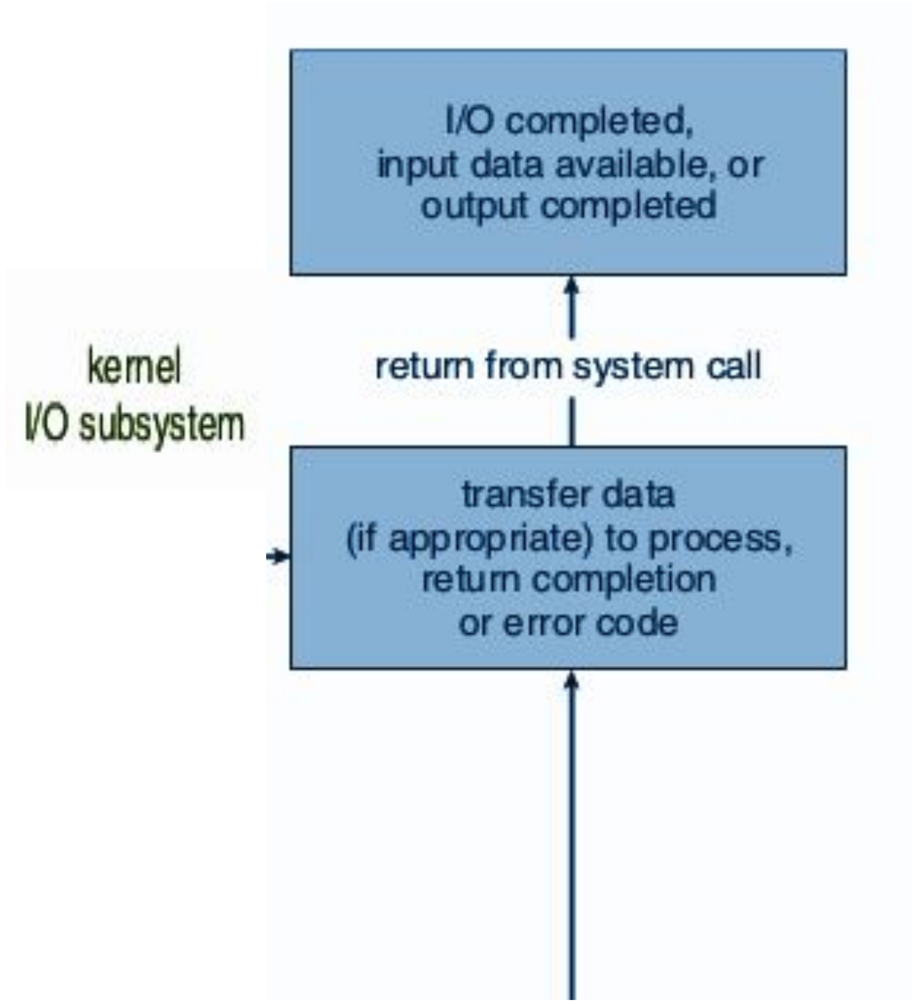
- The kernel transfers data or return codes to the address space of the requesting process and moves the process from the wait queue back to the ready queue.


device driver
determine which I/O completed, indicate state change to I/O subsystem
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
device controller
I/O completed, generate interrupt

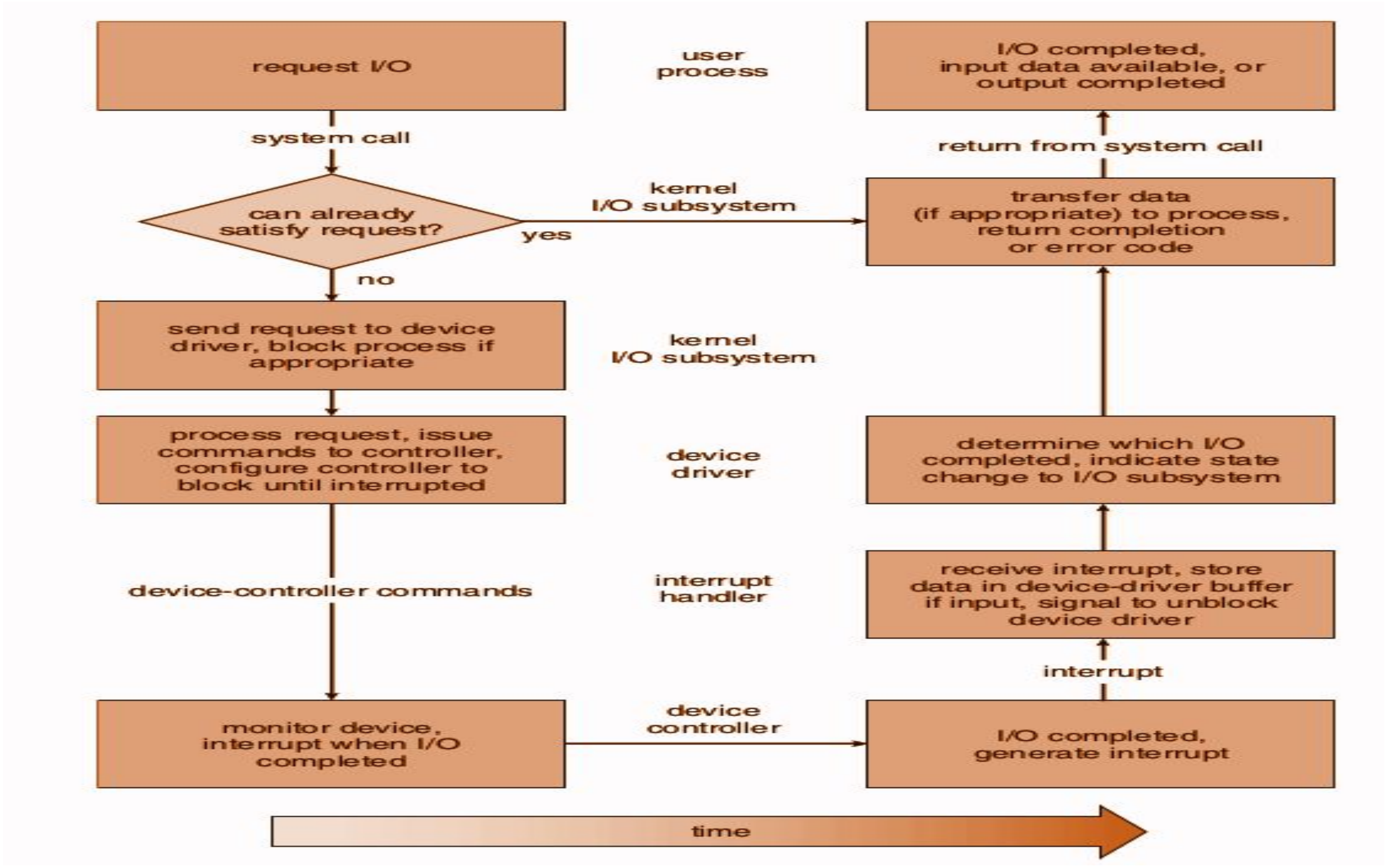
Transforming I/O Requests to Hardware Operations

- Moving the process to the ready queue unblocks the process.

- When the scheduler assigns the process to the CPU , the process resumes execution at the completion of the system call.


I/O completed, input data available, or output completed
return from system call
kernel I/O subsystem
transfer data (if appropriate) to process, return completion or error code

Transforming I/O Requests to Hardware Operations


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

Topics Uncovered

- Transforming I/O Requests to Hardware Operations

PES UNIVERSITY ONLINE

THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

nitin.pujari@pes.edu

For Course Deliverables by the Anchor Faculty click on www.pesuacademy.com