I/O Systems
Operating Systems – EDA093/DIT401

Vincenzo Gulisano
vincenzo.gulisano@chalmers.se

UNIVERSITY OF
GOTHENBURG

Reading instructions

- Chapter 5.1 to 5.3

  (extra facultative reading: 13.1-13.7 from Silberschatz Operating System Concepts)

Agenda

- Introduction
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Performance

Agenda

- Introduction
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Performance

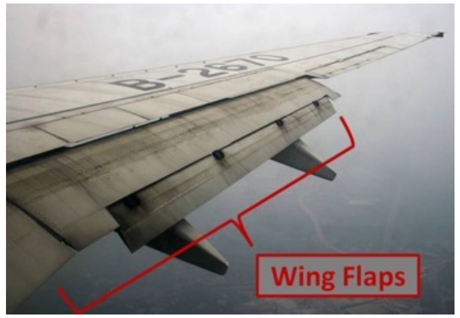
Introduction

- Main jobs of a computer: I/O & processing.
- Role of OS: manage & control I/O operations & I/O devices.

OBJECTIVES:

- Explore structure of an OS's I/O subsystem
- Principles & complexities of I/O hardware
- Explain the performance aspects of I/O hardware & software.

Introduction



I/O challenge: control as good as possible the devices of a given architecture

Why is it a challenge?

1. **Variability** of I/O devices (in their function & speed)
   Varied methods are needed to control them.
2. **Increasing standardization** of software & hardware vs **Increasing types** of devices
3. Different details & oddities of devices

How does the OS address the challenge?

- dedicated I/O subsystem in the kernel
- **device drivers** that provide a **device access** interface to the I/O subsystem

Agenda

- Introduction
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Performance

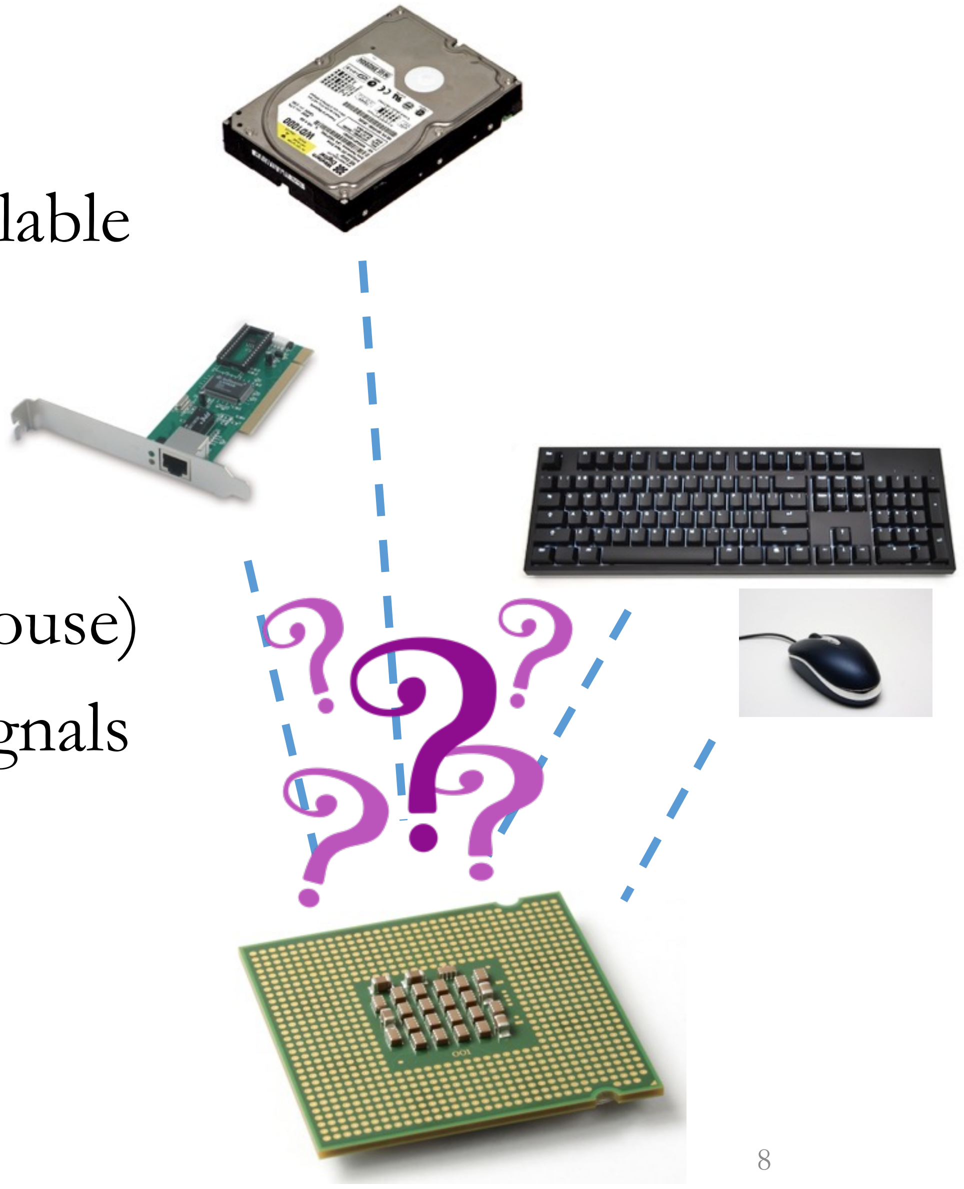
I/O Hardware

We have different types of hardware usually available in a computer:

- Storage devices (disks)
- Transmission devices (network cards)
- Human-interface devices (screen, keyboard, mouse)

All of them exchange information by sending signals (via cable or air)

How do they communicate with the CPU, then?



I/O Hardware

Common concepts:

- **Port**: the connection point (e.g., serial port)
- **Bus**: a set of wires & a protocol that specifies a set of messages that can be sent on the wires.
  - Messages conveyed by patterns of electrical voltages with different timings.
  - Buses vary in signaling methods, speed, throughput & connection methods.
- When device A plugs to device B and device B plugs to device C... We say A,B,C,… form a **daisy chain**
  - A **daisy chain** also operates as a bus

Typical PC bus structure

Important concepts:

- **PCI bus**: connects the processor-memory subsystem to the fast devices.
- **Expansion bus**: connects relatively slow devices.
- **Controller**: a collection of electronics that can operate a port, a bus or a device.

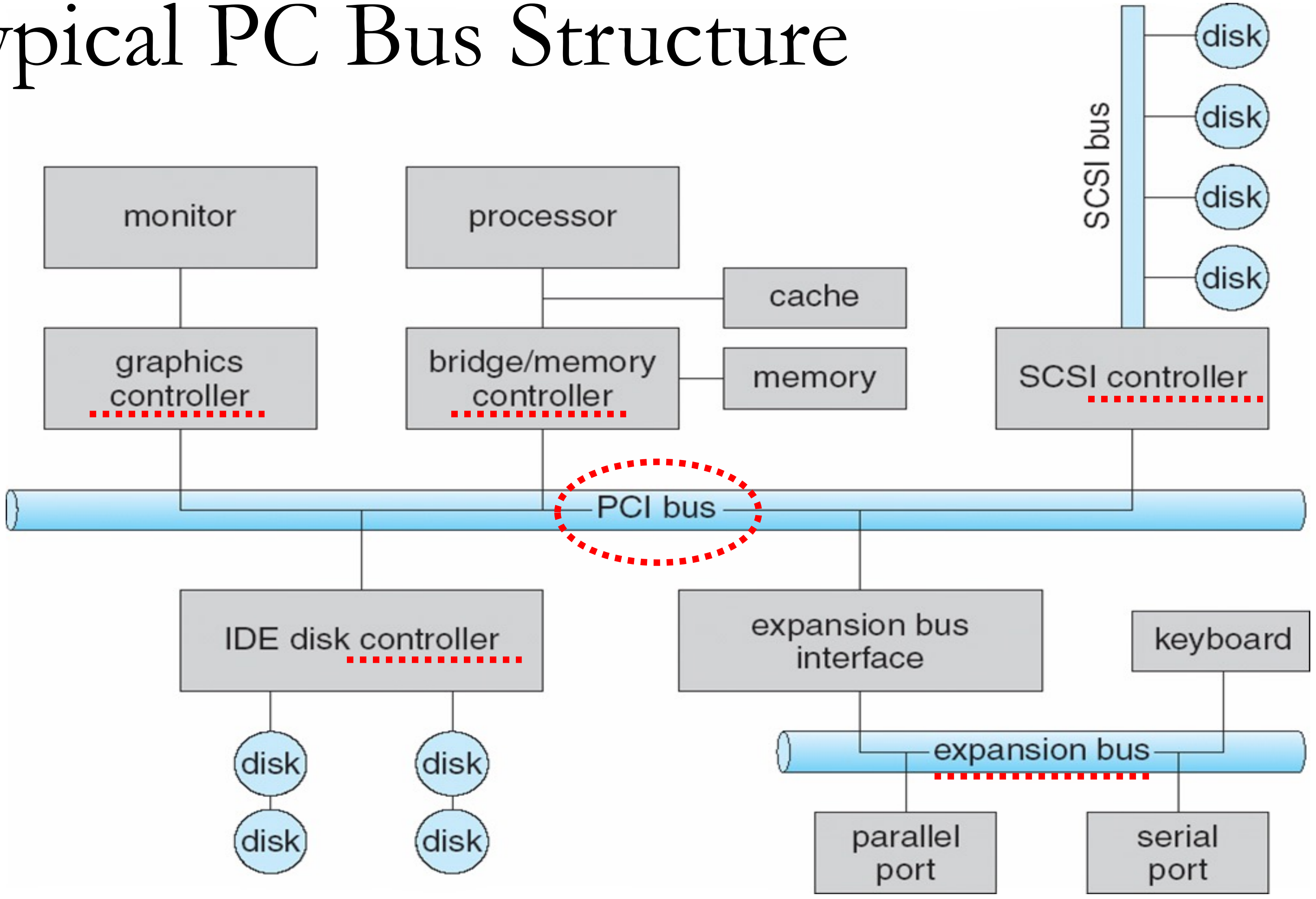
  Example 1: Serial-port controller

  - A single chip (on the host) that controls the signals on the wires of a serial port.

  Example 2: SCSI controller

  - Contains a processor, a microcode, private memory to process the SCSI protocol messages (e.g., disk drive a circuit board).

A typical PC Bus Structure



I/O Hardware

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

1st Way of Communication

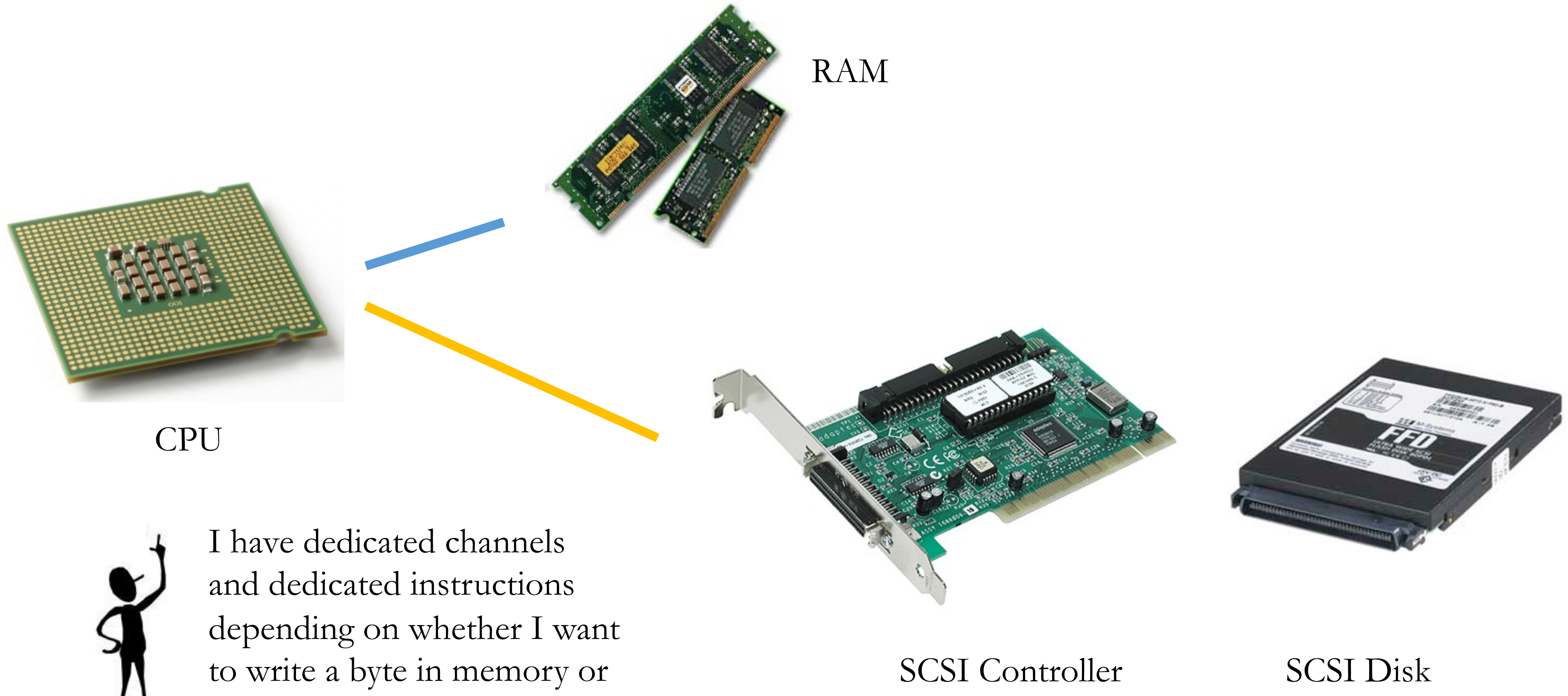
- Processor reads & writes bit patterns in the controllers' registers.
  - Use of special I/O instructions that specify the transfer of a byte or word to an I/O port address.
  - I/O instruction triggers bus lines to select the proper device & move bits into or out of a device register.

2nd Way of Communication: Memory mapped I/O

- Device-control registers are mapped into the address space of the processor.
- CPU executes I/O requests using the standard data transfer instructions to read and write the device-control registers.

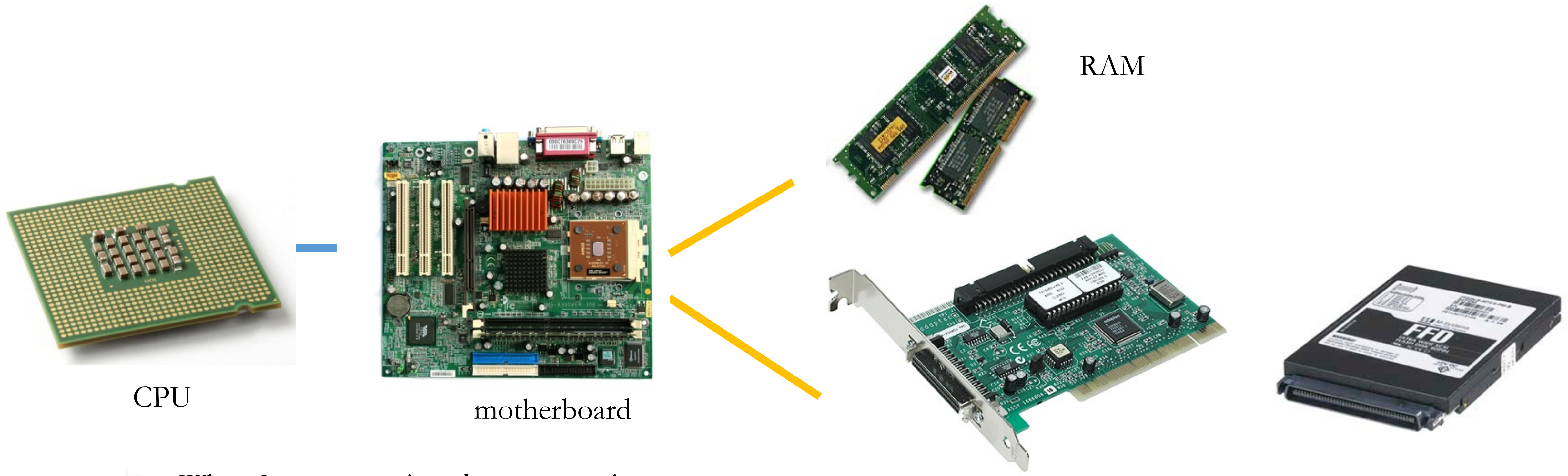
→ Some systems use both ways of communication.

Processor reads & writes bit patterns in the controllers’ registers



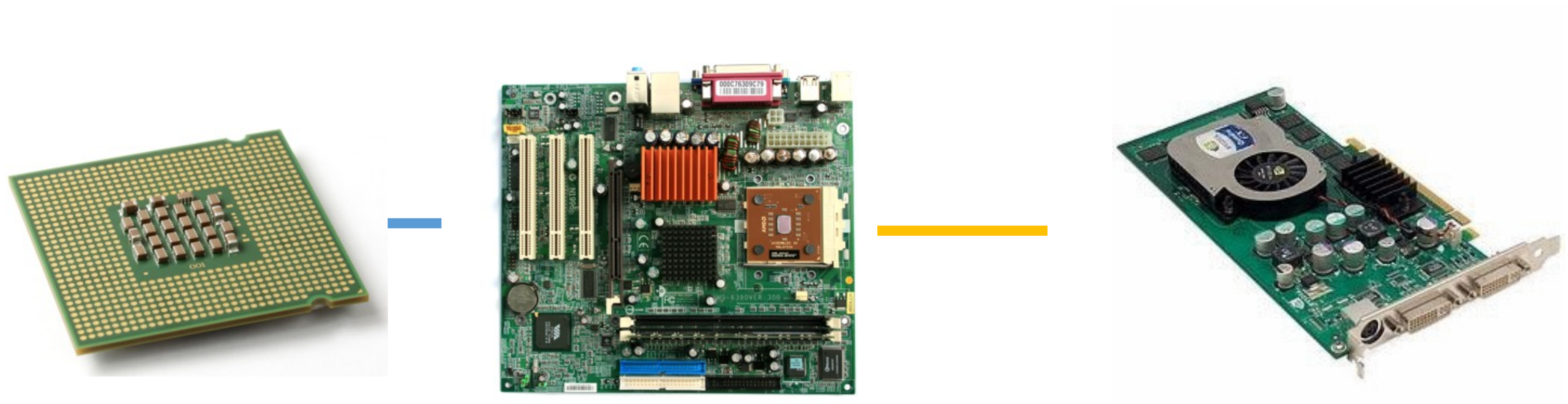
I have dedicated channels
and dedicated instructions
depending on whether I want
to write a byte in memory or
in a certain controller

Memory mapped I/O



When I want to write a byte to a register of a certain controller I use a given address, but I do not really know (or care) how to differentiate a write to memory from a write to a controller

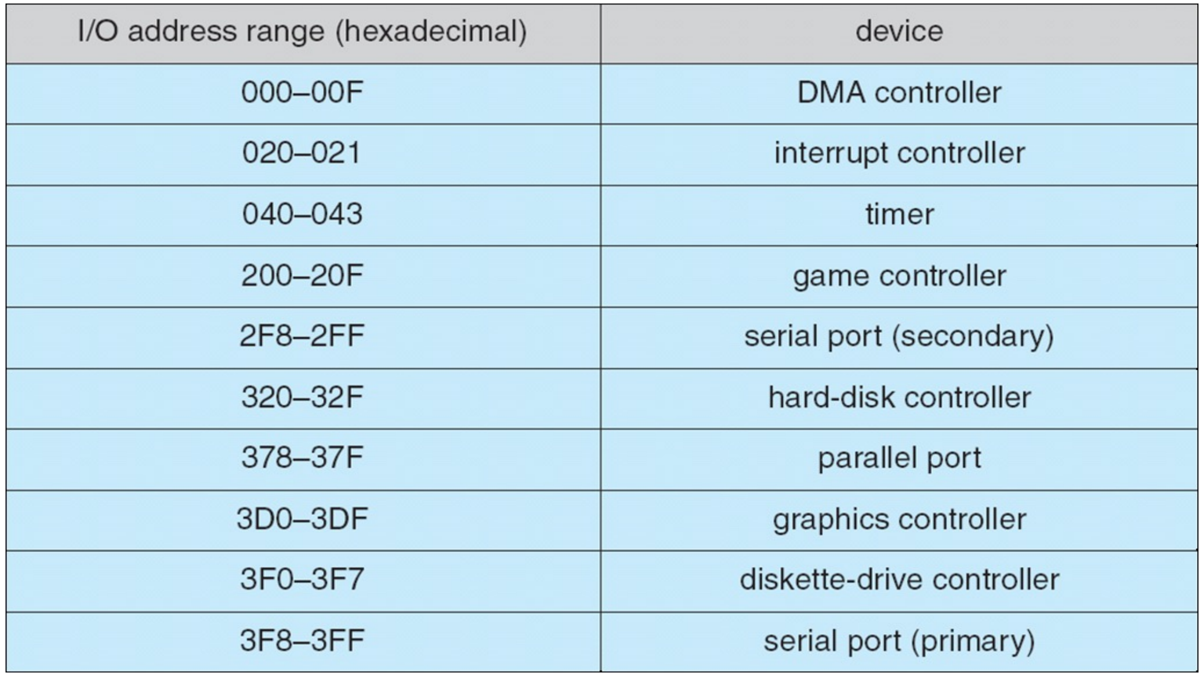
Example of Memory mapped I/O: graphics controller



Graphics controller: has large memory-mapped region to hold screen contents.
Process **sends output** to the screen by writing data into the memory-mapped region.
Controller generates **screen image** based on the contents of the memory.

Device I/O Port Locations on PCs

Usual I/O port addresses for PCs



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

I/O Hardware

I/O port typically consists of:

- Data-in register: is read by the host to get input (e.g. 1 to 4 bytes in size)
- Data-out register: is written by the host to send output
- Status register: contains bits that can be read by the host. Indicates states such as:
  - whether the current command has completed
  - whether a byte is available to be read
  - whether a device error has occurred
- Control register: to start a command or to change the mode of a device.

Example 1: a certain bit in the control register of a serial port chooses between full-duplex & half-duplex communication.

Example 2: another bit sets the word length to 7 or 8 bits, Other bits select one of the speeds supported by the serial port

Interaction between host and controller

- Complete interaction protocol between the host & a controller can be intricate.
  - **busy-waiting / polling:** the host is continuously trying to get access to the device via the controller
  - **interrupt:** the controller alerts the host when the device is available

Polling – Handshaking procedure

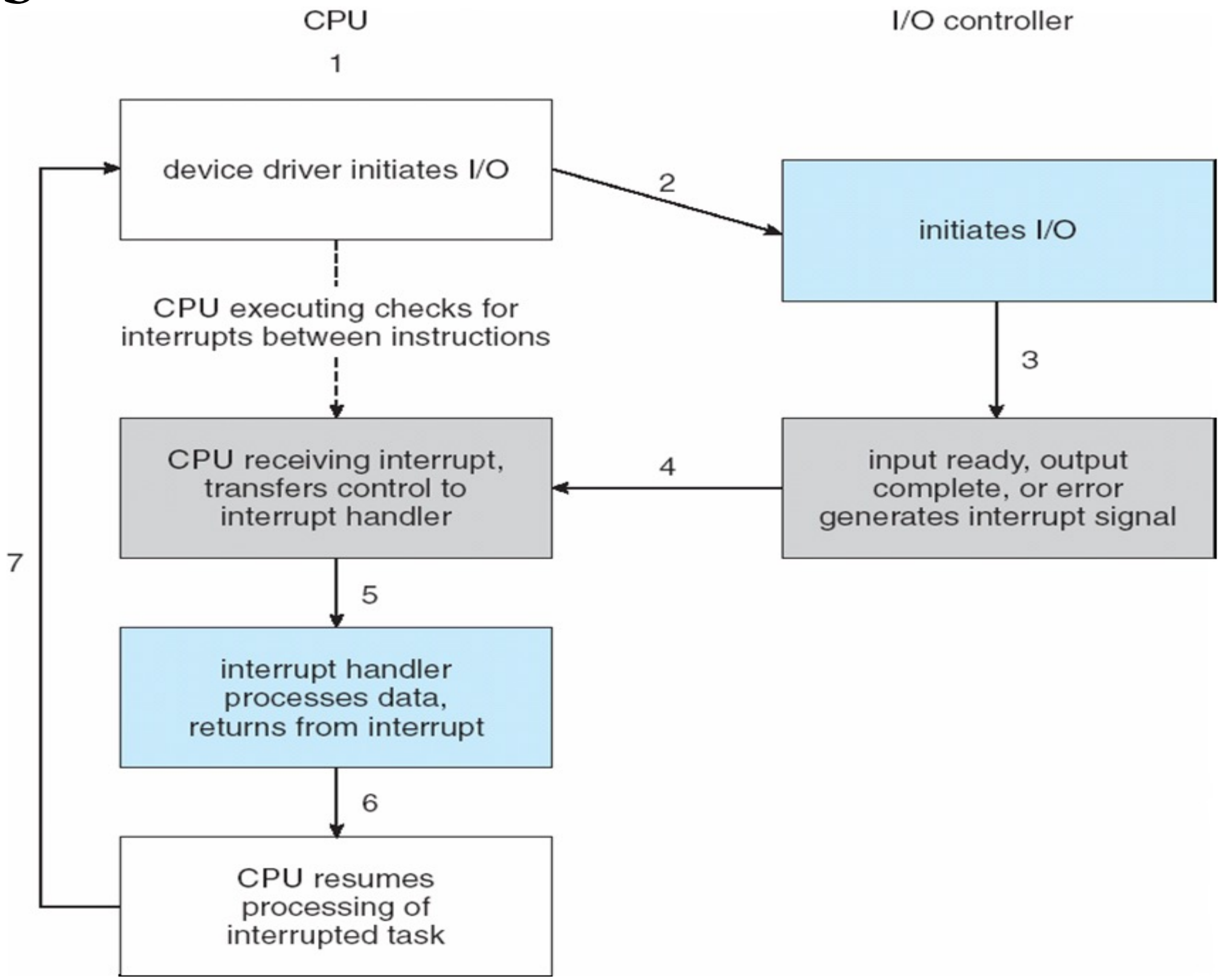
- 2 bits are used to coordinate the producer-consumer relationship between the controller & the host.
- Controller indicates its **state** through the busy bit in the status register.
- Host **signals** its wishes via the command-ready bit in the command register.
  Command ready bit: if it is set a command is available for the controller
- **busy-waiting / polling**: it is a loop, reading the status register over and over until the busy bit becomes clear.
  Reasonable: If device controller & device are fast
  Inefficient: attempted repeatedly but device is busy.
- **In such cases**: hardware controller should notify the CPU when the device becomes **ready** for service → this hardware mechanism is called interrupt.

Interrupts

Basic mechanism

1. CPU hardware has a wire called interrupt request line
2. CPU senses the interrupt request line after executing every instruction.
3. If yes, CPU performs a save state & jumps to the interrupt handler routine at a fixed address in memory.
4. Interrupt handler (IH): What was the cause of the interrupt?
5. IH performs the necessary processing, performs state restore
6. … executes a return from interrupt instruction → return CPU to the execution state prior to the interrupt

Interrupts


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

Interrupts - Dictionary

- The device controller raises an interrupt
- ....by asserting a signal on the interrupt request line
- CPU catches the interrupt
- CPU dispatches it to the interrupt handler
- Handler clears the interrupt by servicing the device

Interrupts

Sophisticated Interrupt Handling

In a modern OS: we need sophisticated interrupt handling features

- Ability to **defer interrupt handling** during critical processing
- Efficient way to locate the proper interrupt handler for a device without asking all devices to see which one raised the interrupt
- Multi-level interrupts so that the OS can distinguish between **high** & **low level priority** interrupts

In modern OS these provided by **CPU** & **interrupt controller hardware**

Interrupts

Two interrupt request lines:

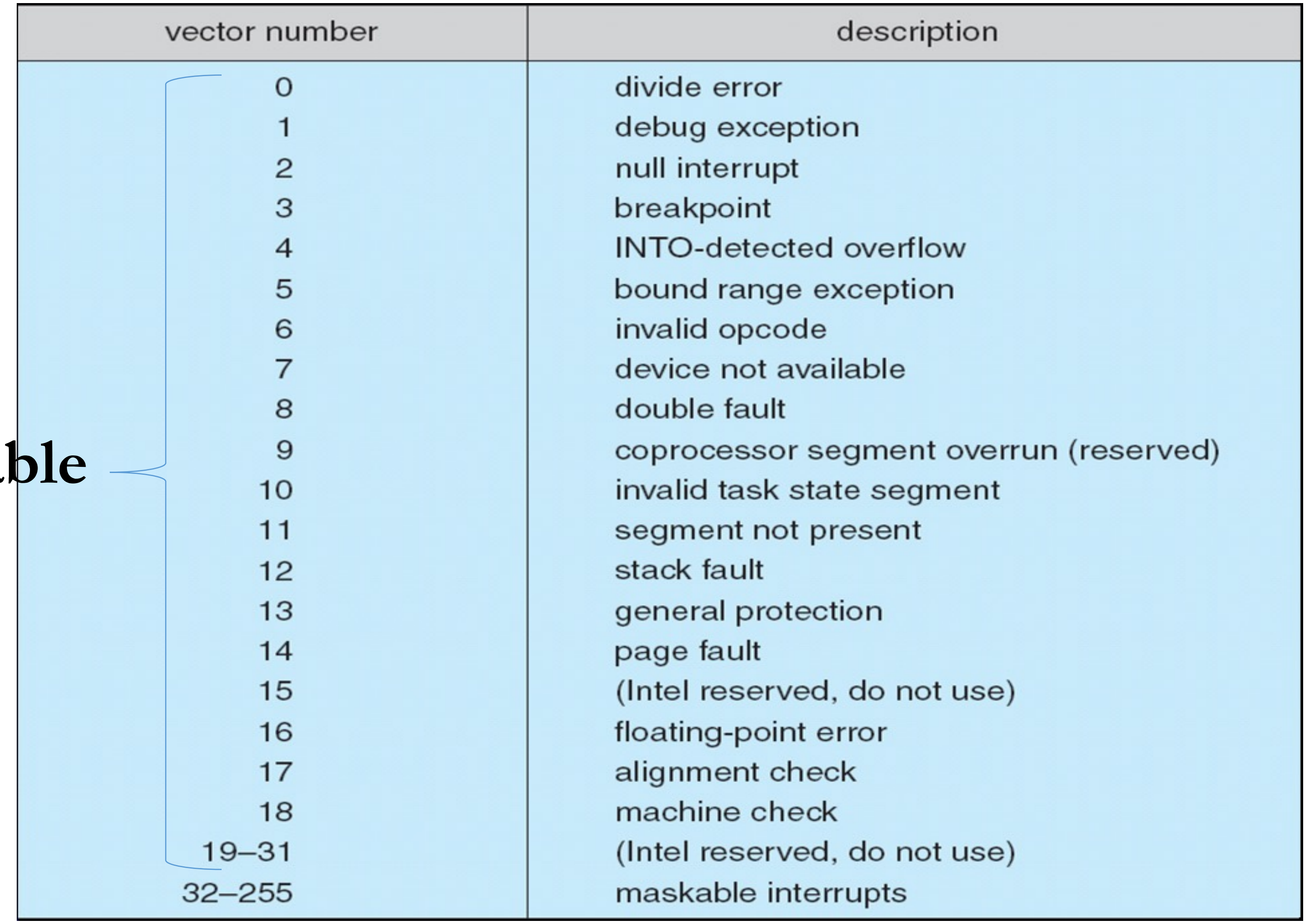
- **Nonmaskable interrupt**: reserved for events such as unrecoverable memory errors.
- **Maskable interrupt**: can be turned off by CPU before the execution of critical instruction requests.

Interrupt mechanism accepts an address: a **number** that selects a **specific interrupt-handling routine** from a small set.

→This address is an **offset** in a table called interrupt vector

This vector contains the **memory addresses** of **specialized interrupt handlers**.

Interrupts – Intel Pentium processor event-vector table



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

nonmaskable

Interrupts

OS & interrupts interaction

- A modern OS interacts with the interrupt mechanism in **several ways**.
- At boot time, **probes** the buses to see which devices are present and **installs the corresponding interrupt handler**.
- During I/O, device controllers **raise interrupts** when they are ready for service. These interrupts may mean:
    - output has completed
    - input data are available
    - or a failure has been detected

Direct Memory Access

Direct Memory Access

- Devices with large transfer e.g. disk drive.
  Not a good idea to use an expensive **general-purpose** processor to watch status bits & feed data into a controller register → **programmed I/O** process.
- Use instead DMA controller
- Bypasses CPU to transfer data directly between I/O device & memory

DMA transfer

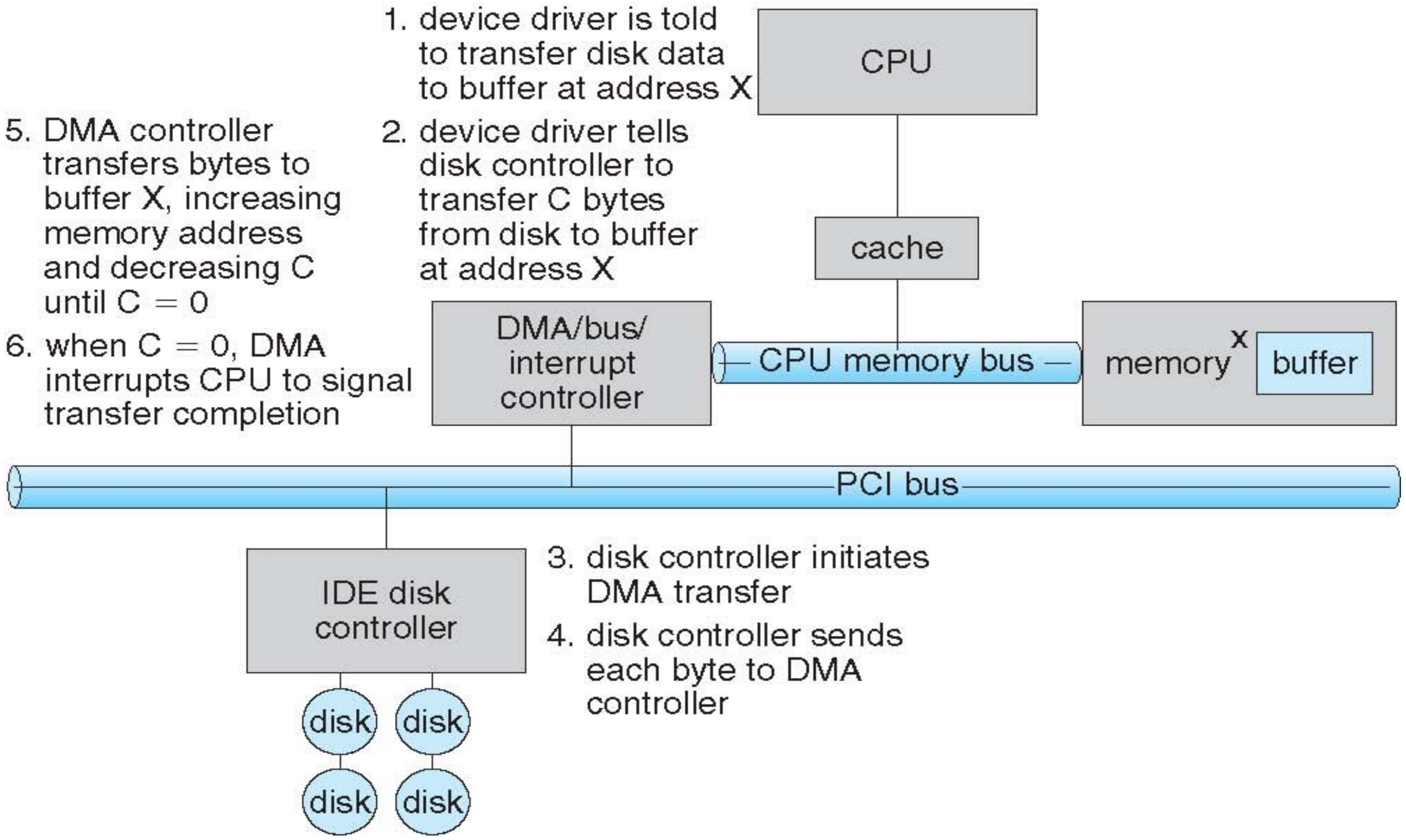
- To initiate a DMA transfer, host writes a DMA command block into memory.
- Block contains:
  - a pointer to the source of the transfer
  - a pointer to the destination of the transfer
  - a count of the number of bytes to be transferred.

Direct Memory Access

Handshaking between DMA controller & device controller

- Performed via a pair of wires: DMA request & DMA-acknowledge.
- A word of data available for transfer → device controller places a signal on the DMA-request wire
- DMA controller:
  - places the **desired memory address** on the memory-address wires.
  - places a signal on the DMA-acknowledge wire.
- Device controller receives the `DMA-acknowledge` signal & transfers the word of data to memory & removes the `DMA-request` signal.

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
CPU
cache
CPU memory bus
memory
x
buffer
DMA/bus/ interrupt controller
PCI bus
IDE disk controller
disk
disk
disk
disk

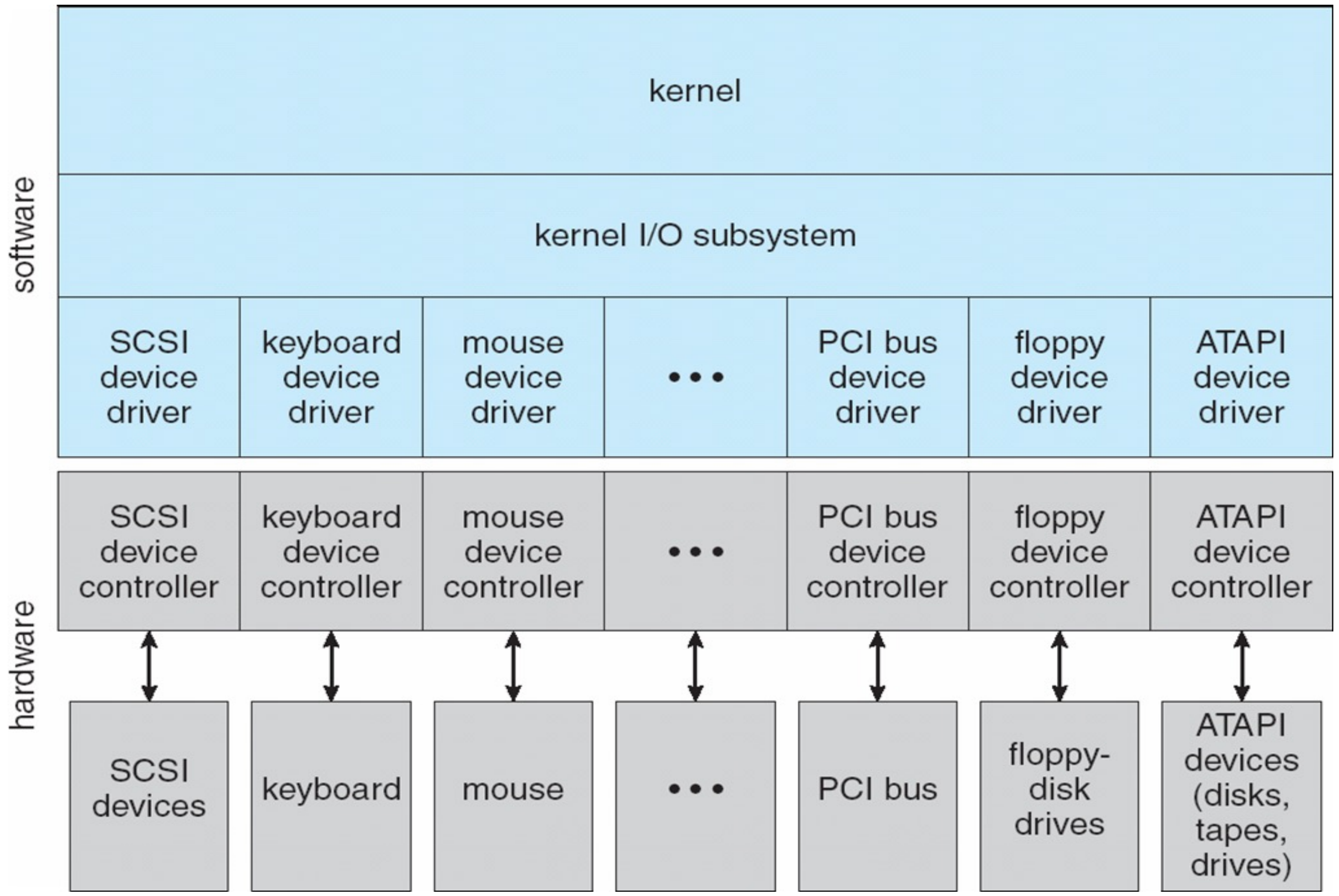
Agenda

- Introduction
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Performance

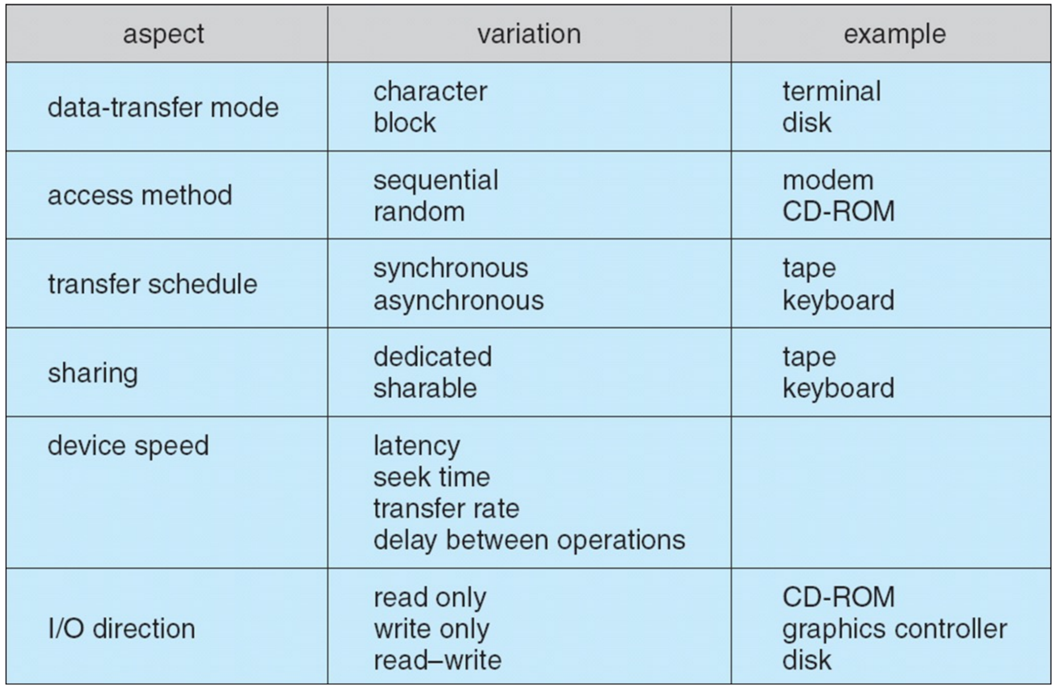
Application I/O Interface

- **Goals:**
  - **Abstract away** the detailed differences in I/O devices by identifying a few general kinds.
  - Encapsulate device behaviors in general
- Device-driver layer **hides differences** among I/O controllers from kernel.
- Make I/O subsystem independent of the hardware:
  - Simplifies the job of the OS.
  - Benefits the hardware manufacturers
  - Attach peripherals without waiting for new support code
  - **BUT**: each type of OS has its own standards for the device-driver interface.

Application I/O Interface



Application I/O Interface



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

Devices vary in many dimensions:

- **Character-stream** or **block**: bytes one by one or blocks of bytes
- **Sequential** or **random access**: fixed order or random
- **Synchronous** or **asynchronous**: data transfers with predictable response times or not
- **Sharable** or **dedicated**: used concurrently by several processes or not
- **Speed of operation**: speeds range form a few bytes to gigabytes per sec
- **Read-write**, **read only** or **write only**: some both directions, others not

Block and Character Devices

Block devices include disk drives

Block-device interface all necessary aspects for accessing disk drives & other block-oriented devices

- Commands include read(), write(), seek()
- Memory-mapped file access possible

Character devices

- Include keyboards, mouse, serial ports
- Commands include get(), put()
- Libraries layered on top allow line editing

Network Devices

Network Devices

- Network I/O differs significantly form disks, the interface is not `read(), write()` and `seek()`
- Unix & Windows include socket interface
- System calls in the socket interface enable an application to
  - Create a socket
  - Connect to a remote socket

Socket Interface

- Connect local socket to a remote address
- Listen remote application to plug into the local socket
- Send & receive packets over the connection
- `select()` → return which sockets have a packet waiting to be received

Clocks & Timers

- Give current time.
- Give elapsed time.
- Set a timer to **trigger** operation X at time T.
- **Programmable hardware interval timer**: hardware to measure elapsed time & trigger operations
- Example: wait a certain amount of time & then generate an interrupt **Cancel** operations preceding **too slowly**
- **Usual interrupt rate**: between 18 & 60 ticks per sec.
- Hardware clock constructed from a high-frequency counter.

Blocking & Nonblocking I/O

Blocking

- Process suspended until I/O completed
- Moved from run queue to wait queue
- Later moved back to run queue & resumes
- Easy to use & understand
- Insufficient for some needs

Nonblocking

- Example: a user interface that receives keyboard & mouse input while processing & displaying data on screen.
- Implemented via multi-threading (a thread does the blocking part, another processes results)
- Returns quickly with a return value indicating how many bytes were transferred.

Blocking & Nonblocking I/O

Asynchronous

- Process runs while I/O executes
- Returns immediately without waiting for I/O to complete
- Difficult to use
- I/O subsystem signals process when I/O completed

Agenda

- Introduction
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Performance

Kernel I/O subsystem

- Provides many services related to I/O, including:
  - I/O scheduling
  - Buffering / Caching / Spooling
  - Error handling
  - I/O protection

I/O Scheduling

Goal: schedule I/O requests in a **good order** before executing them

- **Improve** overall performance
- Share device access **fairly** among processes
- **Reduce** the average waiting time for I/O to complete

- Some OS maintain a wait queue of requests for each device
- Some OS try **fairness**

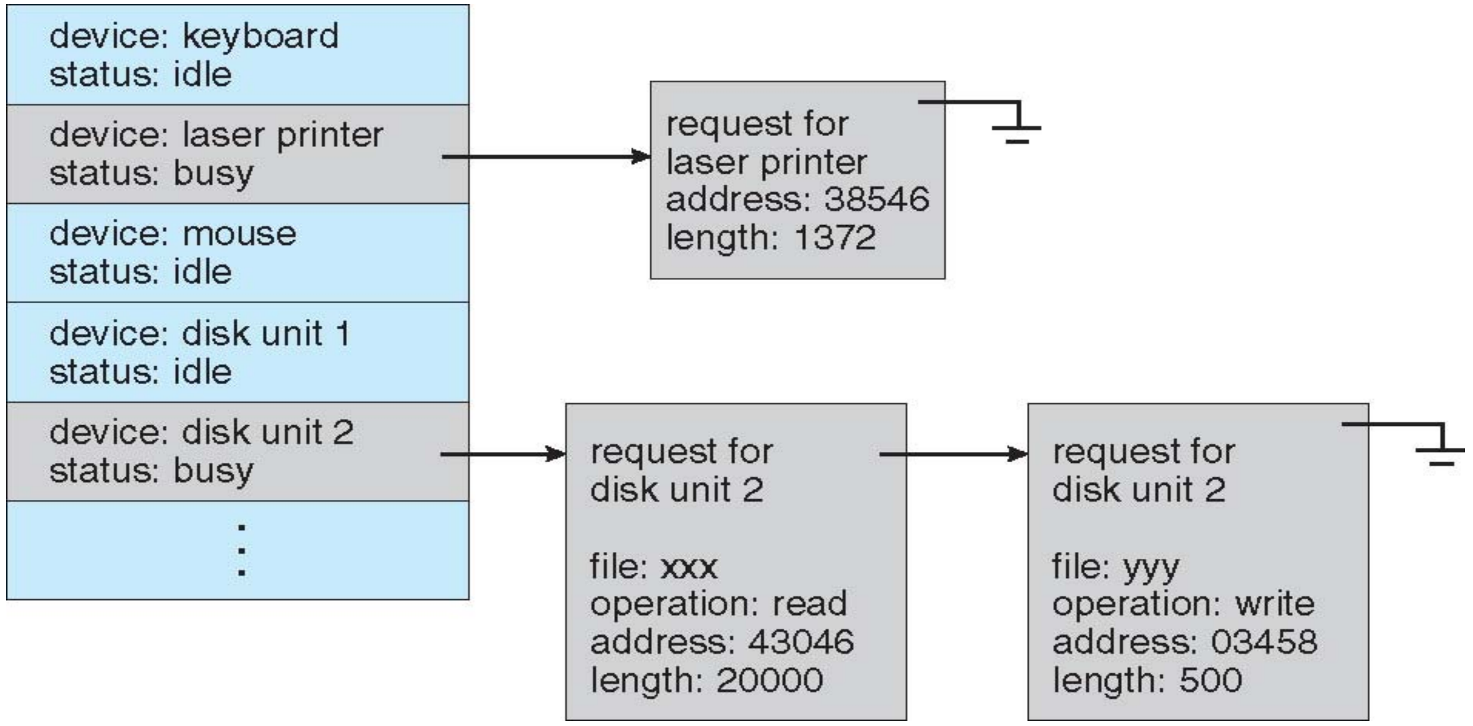
Example

- Application 1 needs a block near the end of a disk
- Application 2 needs a block near the beginning
- Application 3 needs a block near the middle of the disk

Best Scheduling: 2,3,1

Device status table

- When a kernel supports **asynchronous** I/O and might schedule requests in different orders, it must be able to keep track of many I/O requests



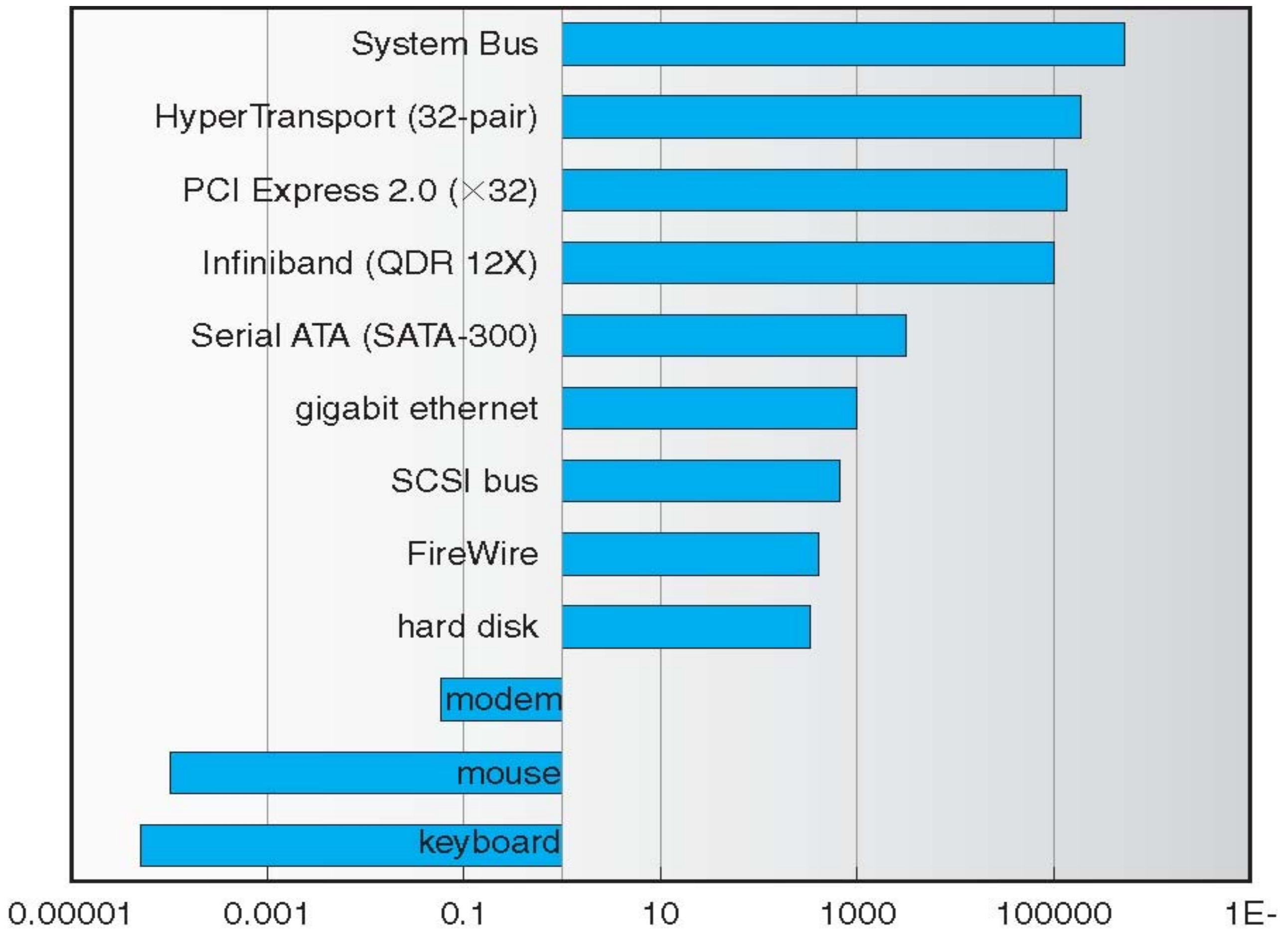
Buffering

- **Another way to improve performance** → Use storage space in main memory or on disk via buffering, caching, spooling

Buffering

- Buffer: memory area that stores data being transferred between two devices or between a device & an application.
- Example:
  - File received via modem for storage on the hard disk.
  - Buffer created in main memory to accumulate the bytes received from the modem.
  - Buffer full → write data on the disk
  - Disk write **not instantaneous**: two buffers are needed → Double buffering

Buffering – Differences in device speeds



Enormous differences in device speeds for typical hardware!

Sun Enterprise 6000 device-transfer rates (logarithmic)

Caching – Spooling

Caching

- Cache: holds a copy on faster storage of an item that resides elsewhere
- Easier access → Key to performance

Spooling

- Hold output for a device
- If device can serve only one request at a time (e.g. printing)
  Devices that cannot accept interleaved data streams.

Error Handling

<u>Error Handling</u>

- Devices and I/O transfers can fail in many ways.
  - **Transient reasons**: a network becomes overloaded
  - **Permanent reasons**: a disk controller becomes defective
- OS can recover from disk read, device unavailable, transient write failures
- Most return an error number or code when I/O request fails
- System error logs hold problem reports

I/O Protection

- **Necessary**: User access may attempt to disrupt normal operation via illegal I/O instructions.
- All I/O instructions defined to be **privileged** → Users cannot issue them directly
- To do I/O a user program **executes** a system call
- OS **checks** if request is **valid**
- If yes it is executed.
- Memory mapped & I/O port memory locations must be **protected**

Agenda

- Introduction
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Performance

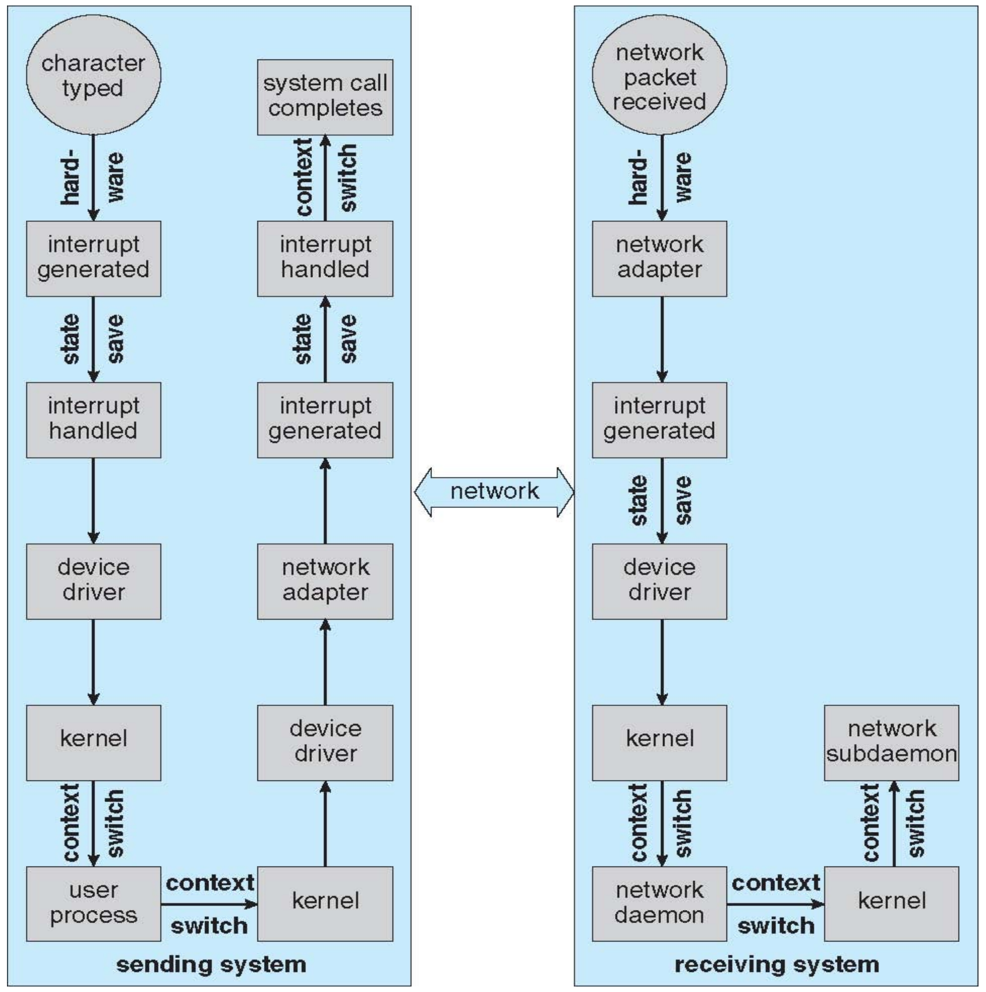
Performance

I/O has major impact in system performance:

Why?

- Places heavy demands on CPU to execute device driver
- Schedule processes fairly & efficiently
- Context switches due to interrupts
- …

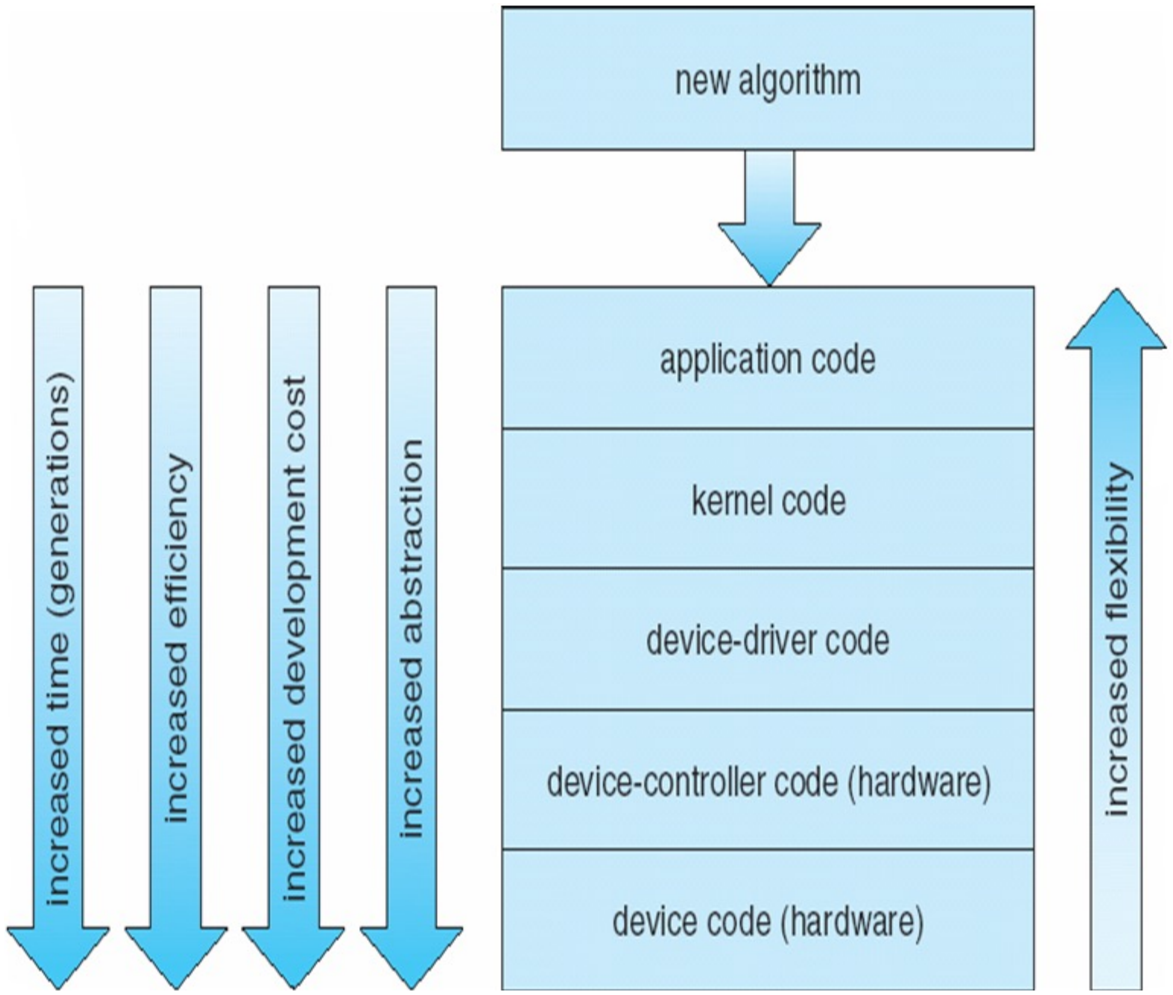
Example of performance: remote login



How to improve performance?

- Reduce number of context switches
- Reduce number of times data must be copied in memory (between device & application)
- Reduce frequency of interrupts
  Use large transfers, smart controllers
- Use DMA
- Balance CPU, memory, bus and I/O performance for highest throughput

Performance


new algorithm
application code
kernel code
device-driver code
device-controller code (hardware)
device code (hardware)
increased time (generations)
increased efficiency
increased development cost
increased abstraction
increased flexibility