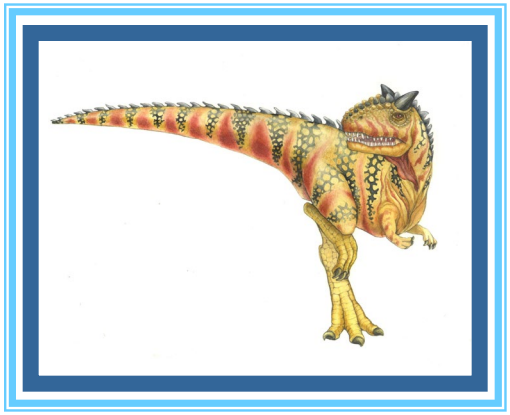
I/O System management



Agenda

- Overview
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- Performance

Objectives

- Explore the structure of an operating system's I/O subsystem
- Discuss the principles of I/O hardware and its complexity
- Provide details of the performance aspects of I/O hardware and software

Overview

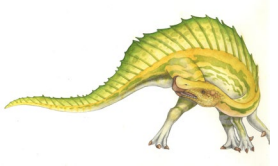
- The control of **I/O devices** connected to the computer is a significant concern of OS designers.
  - Because **I/O devices** vary so widely in their function and speed (consider a *mouse*, a *hard disk*, or an *optical disc*), so different methods are needed to control them.
- Such methods form the **I/O subsystem of the kernel**, which separates the rest of the kernel from the complexities of managing **I/O devices**.
- I/O-device technology exhibits **two conflicting trends**:
  - increasing standardization of SW and HW interfaces.
  - increasingly wide variety of I/O devices.

Overview

- Data behavior and structure of new I/O devices are so. Hence it is challenging to incorporate them into an OS.
- The essential I/O hardware elements, such as ***ports***, ***buses***, and ***device controllers***, accommodate various **I/O devices**.
- To encapsulate the details of different I/O 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 of the kernel**:
  - *Based on **system calls** generated by each **I/O device driver**, the OS provides a standard interface between the device and the application.*

Overview

- **I/O management** is a major section of an OS design, and it should handle the following:
  - *I/O operation of the device*
  - *I/O devices vary greatly*
  - *Various methods to control them*
  - *Performance management*
  - *New types of devices frequent*
  - *Present uniform device-access interface to I/O subsystem*
- ***Ports***, ***busses***, and ***device controllers*** connect to various devices
- **Device drivers** encapsulate each device in details



I/O Hardware

- Computers operate with a variety of **I/O devices,** such as:
  - **Storage**: disk, Flash drive, etc.
  - **Transmission**: network, Bluetooth, etc.
  - **Human interface**: keyboard, mouse, display, audio in and out, etc.
- **Signals from I/O devices interface with a computer through:**
  - **Port** – connection point for the device, e.g. USB, HDMI, etc.
  - **Bus** – group of signal conduction cables, operate based on a **daisy chain** or **shared direct access**
    - **PCI** bus connects fast devices to the **processor-memory** system
    - **Expansion bus-** connects relatively slow devices such as keyboard, serial ports, etc
    - **SCSI** – the bus that supports ***hard disks***

I/O Hardware

- An **I/O device controller** (or a **host adapter**) is a collection of electronics that can operate a ***port***, a ***bus***, or a ***device***.
- For example, a **serial-port controller** (or a USB port controller) is a single chip (or portion of a chip) in the computer that controls the signals on the **serial port**.
- A typical **PC** (**Personal Computer**) **bus structure** appears in **Figure 13.1.**

A Typical PC Bus Structure

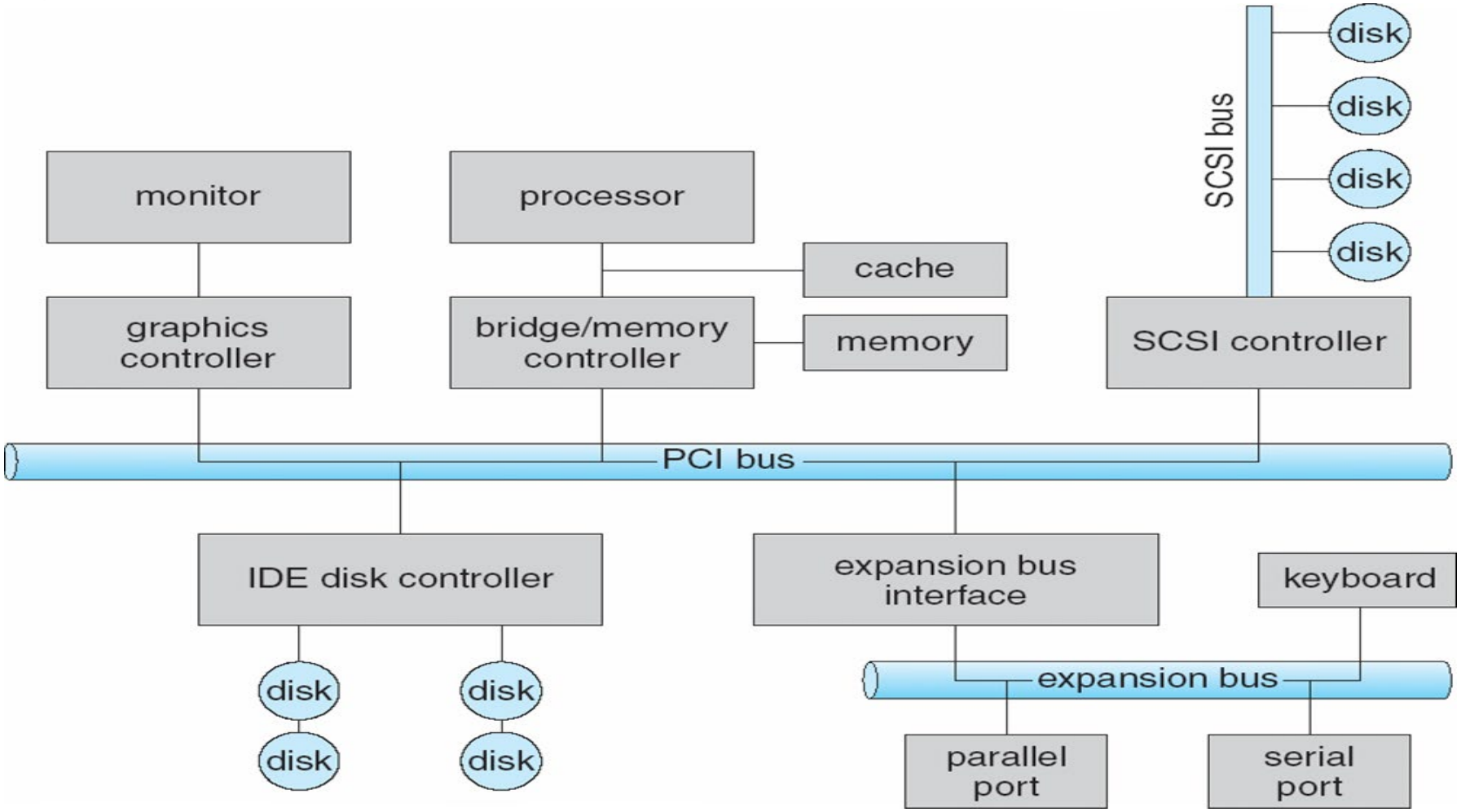


Figure 13.1 A typical PC bus structure.

I/O Hardware

- The **SCSI bus controller** has a complex protocol and is often implemented as a separate circuit board (or a **host adapter)**.
- **How can the CPU give commands and data to an I/O controller to transfer data?**
  - The ***controller*** has one or more **registers** for handling ***data*** and ***control*** signals.
  - The CPU communicates with the **controller** by reading the **control register** for putting the **I/O command** and finding the **status** of the device, and then writing a bit patterns into the **data register** for an **I/O write**
    - This communication is through **I/O instructions** that specify the transfer of a byte or word to an **I/O port address.**
  - The **I/O instruction** triggers bus lines to select the proper devices and to move bits into or out of a **device register**.

I/O Hardware

- The **device controller** can support **memory-mapped I/O** operation:
  - In this case, **the device-control registers** are mapped into the ***system's memory address space***.
- Sometimes, the CPU executes **direct I/O instructions** (**programmed I/O**) for **I/O operations** by reading or writing the **device-control registers** and transferring data from/to **I/O devices**.
- Some systems use **I/O instructions** and **memory-mapped** techniques for **I/O device** support.
- **Figure 3.2** shows the **I/O port addresses** for PCs (personal computers).

Device I/O Port Locations on PCs (partial)

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)

Figure 13.2 Device I/O port locations on PCs (partial).

I/O Hardware -Summary

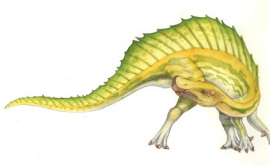
- **I/O devices** usually have **registers** where **device drivers** place *commands*, *addresses*, and *data to write* or *read data from* registers after command execution
  - *data-in register*, *data-out register*, *status register*, *control register*
  - Typically 1-4 bytes, or FIFO buffer
- Devices have addresses used by
  - **Direct I/O** (also called **Programmed I/O operations**)
  - **Memory-mapped I/O**

I/O Registers

- 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 the host can read.
    - These bits indicate **status** states, such as whether the current command has been completed, whether a byte can be read from the **data-in register**, and whether a **device error** has occurred.
  - The **host can write the control register** to start a **command** or change a device's mode.
    - For instance, a particular bit in the **control register** of a serial port chooses between **full-duplex** and **half-duplex** communication, another bit *enables parity checking*, etc.

Polling: Programmed I/O or Busy waiting I/O

- **Polling** is a technique to identify the status of an I/O device in **programmed I/O** operations.
- The **I/O controller** indicates its device state or status through the **busy bit** in the **status register**.
- The controller **sets** (1) the **busy bit** when the device is **busy** and **clears** (0) the **busy bit** when it is **ready** to accept the command.



Polling: Programmed I/O or Busy waiting I/O

- Assume a **host system** performs an **I/O write to a device** through a **port** coordinates with its **I/O controller** as follows:
  1. Read the **busy bit** from the **status register** of the I/O controller until it is 0 (0 = ready) is called **I/O *polling***.
  2. If it is **ready**, the host sets the **write bit** in the **command register** and writes a byte into the **I/O controller**'s **data-out register**.
  3. Then, the host **sets** the **command-ready bit.**
  4. When the **I/O controller** notices that the **command-ready bit** is **set**, it sets the **busy bit** to indicate a write operation.
  5. Then, the controller reads the **command register** and sees the **write command** from the host.
  6. It reads the **data-out register** to get the byte and does **the I/O write** to the device.
  7. Controller clears the **busy bit, error bit**, and **command-ready bit** when the I/O transfer is done

Polling: Programmed I/O or Busy waiting I/O

- The **I/O polling** operation is indicated in the **step1:**
  - the host is **busy waiting or polling:** it is in a loop, reading the **status register** repeatedly until the busy bit becomes clear.
- In many computer architectures, **three CPU instruction cycles** are sufficient to **poll** an external device:
  - ***read a device status register***,
  - ***extract its status bit***, and
  - ***branch if not zero***.

Polling: Programmed I/O or Busy waiting I/O

- **Programmed I/O Steps (OS viewpoint):**
- The running process sends an **I/O request** to the OS via a **system call**, hoping that the OS provides some I/O services (e.g., read a file in the Disk)
- The **OS receives the I/O request**; **it blocks the requested process**, which means that the CPU is released from the process for the **I/O operation** to be completed.
- **The I/O subsystem within the OS (kernel)** handles the **I/O request**.
- **I/O subsystem** passes the request to the **device driver**.
- The **device driver** sets the related **I/O commands** based on the request.
- The **device controller is ready to support the I/O device**.
- The OS can allocate the **CPU resources** for the **I/O operation**.
- In case, the **I/O controller** is **not ready**, the CPU may continuously **poll the I/O-device controller to check whether it is ready or not or if there is any error**.

Polling: Programmed I/O or Busy waiting I/O

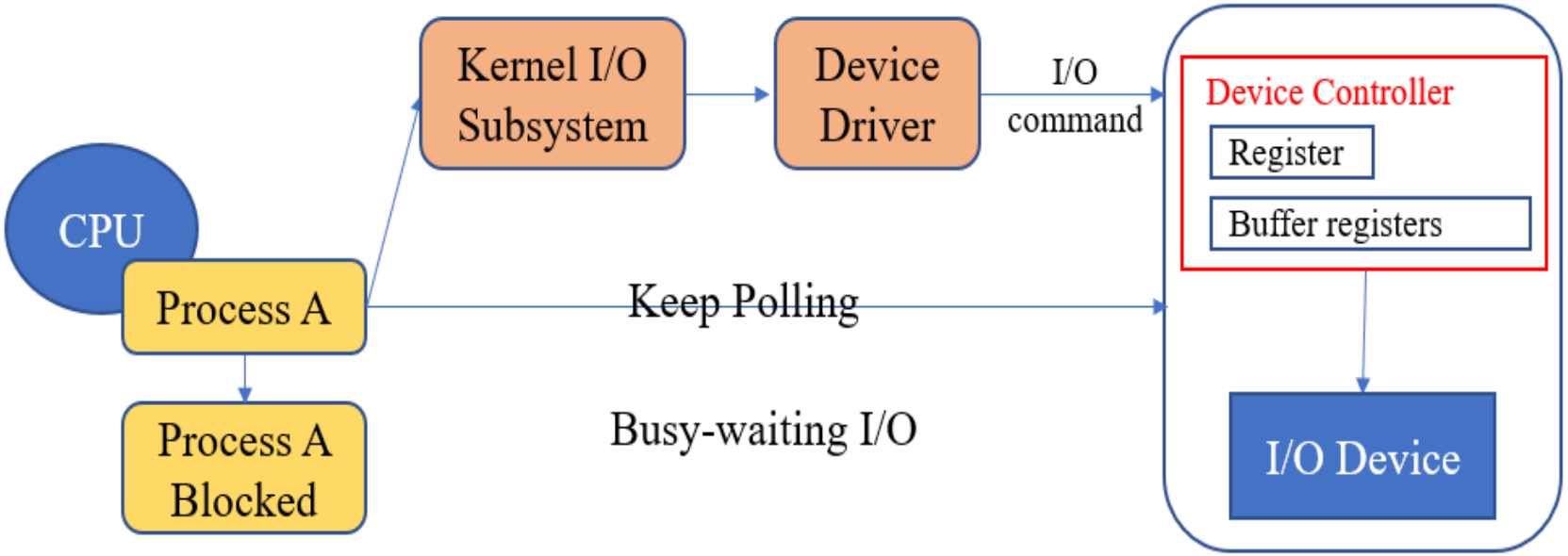
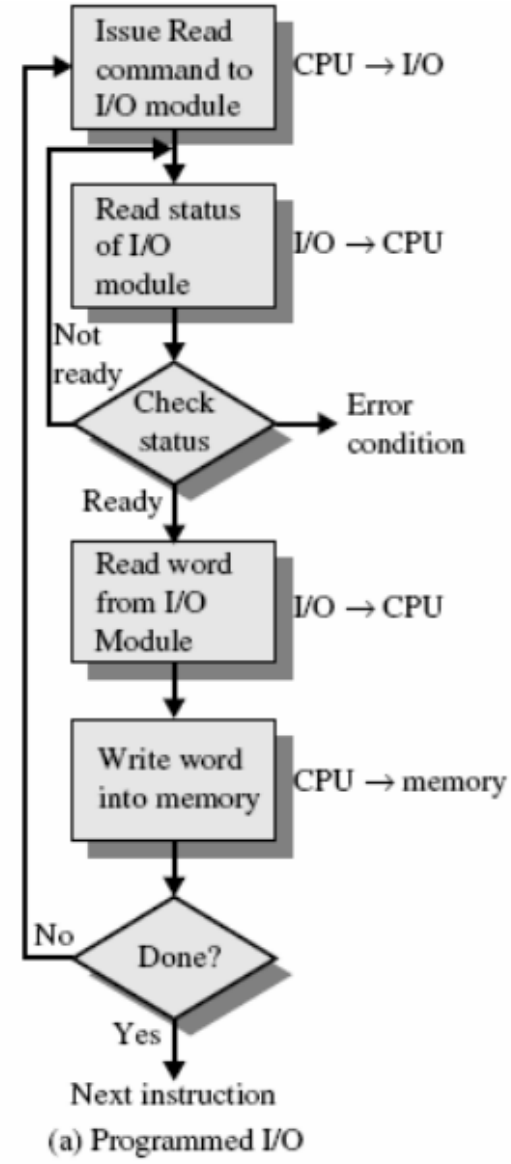


Figure resource:
https://medium.com/@dailan81923/operating-system-chapter-2-5996eb85e74b

Polling: Programmed I/O or Busy waiting I/O



(a) Programmed I/O

Polling: Programmed I/O or Busy waiting I/O

Advantages & Disadvantages of Programmed I/O

Advantages	simple to implement
	very little hardware support
Disadvantages	busy waiting
	ties up CPU for long period with no useful work

http://inputoutput5822.weebly.com/programmed-io.html

Memory Mapped I/O

- With **programmed I/O**, there is a close correspondence between the **I/O-related instructions** that the processor fetches from memory and the **I/O commands** (either I/O-read or I/O-write) that the processor issues to an I/O module to execute the instructions (I/O operation).
- When the processor issues an **I/O command**, the command contains the (**port**) **address** of the desired device.
  - Thus, the **I/O controller** must interpret the address lines to determine if the command is for itself and which external devices the address refers to.
- When the **processor** and **main memory** share a **common bus**, **two modes** of addressing are possible: **Memory mapped I/O** and **Isolated I/O**

http://inputoutput5822.weebly.com/programmed-io.html

Memory Mapped I/O

- With **memory-mapped I/O**, there is a single address space for memory locations and **I/O devices**. The **processor** treats the ***status*** and ***data registers*** of I/O modules as **memory locations** and uses the exact machine instructions to access both memory and I/O devices.
  - So, for example, ten address lines can support 1024 memory locations and I/O addresses in any combination.
  - With **memory-mapped I/O**, a single read line, and a single write line are needed on the bus.

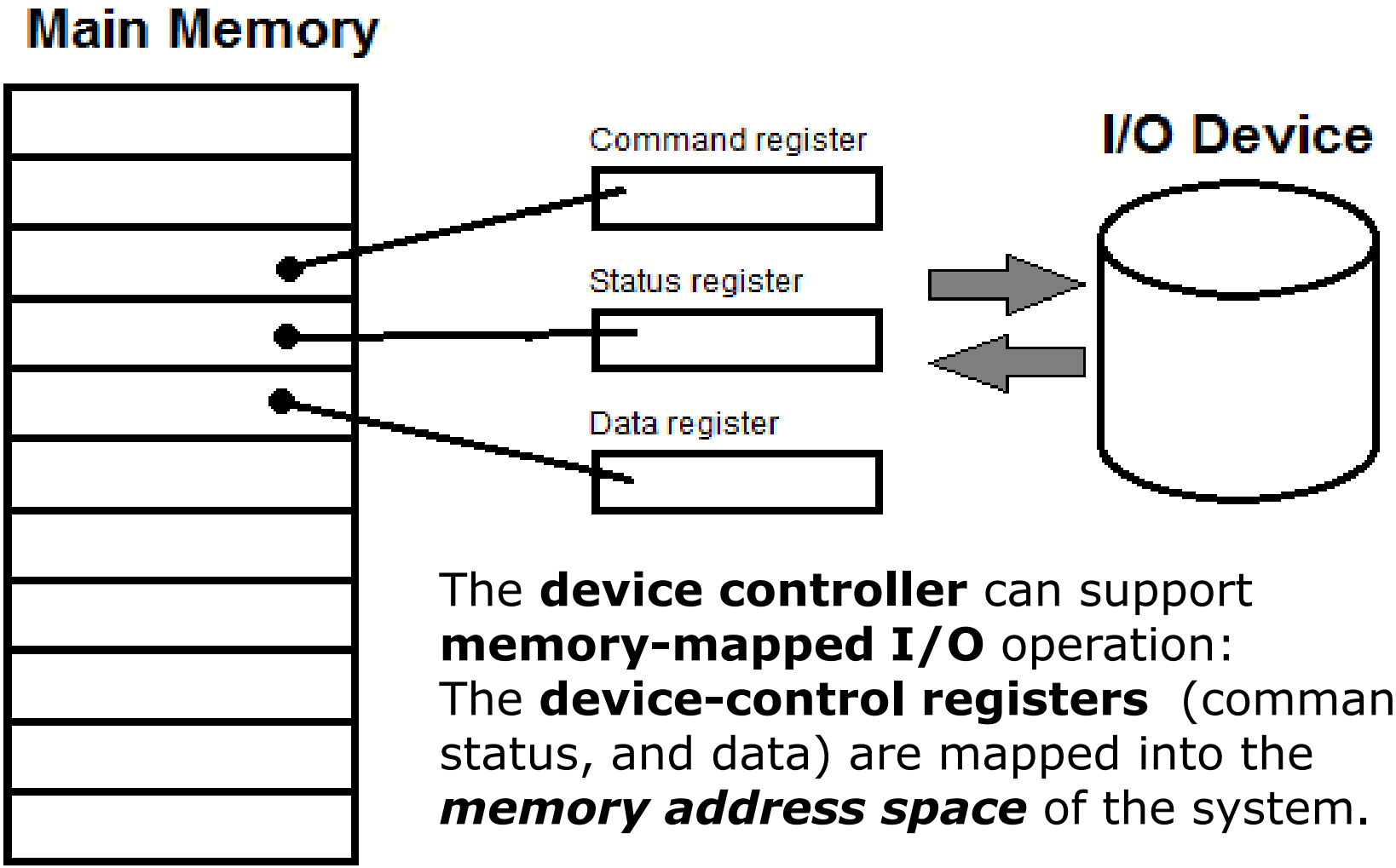
http://inputoutput5822.weebly.com/programmed-io.html

Memory Mapped I/O

- With **isolated I/O**, the I/O bus may have the memory to **read** and **write** plus **input** and **output** command lines.
- The **command line** specifies whether the address refers to a **memory location** or an **I/O device**.
- For example, with ten address lines, the system may support some locations for either I/O addresses or 1024 memory locations.
  - For most types of processors, there is a relatively large set of instructions for **referencing memory**. If **isolated I/O** is used, there are only a few I/O instructions, allowing more efficient programming. A disadvantage is that valuable memory address space is used up.

http://inputoutput5822.weebly.com/programmed-io.html

Memory Mapped I/O



The **device controller** can support **memory-mapped I/O** operation:
The **device-control registers** (command, status, and data) are mapped into the ***memory address space*** of the system.

Memory Mapped I/O

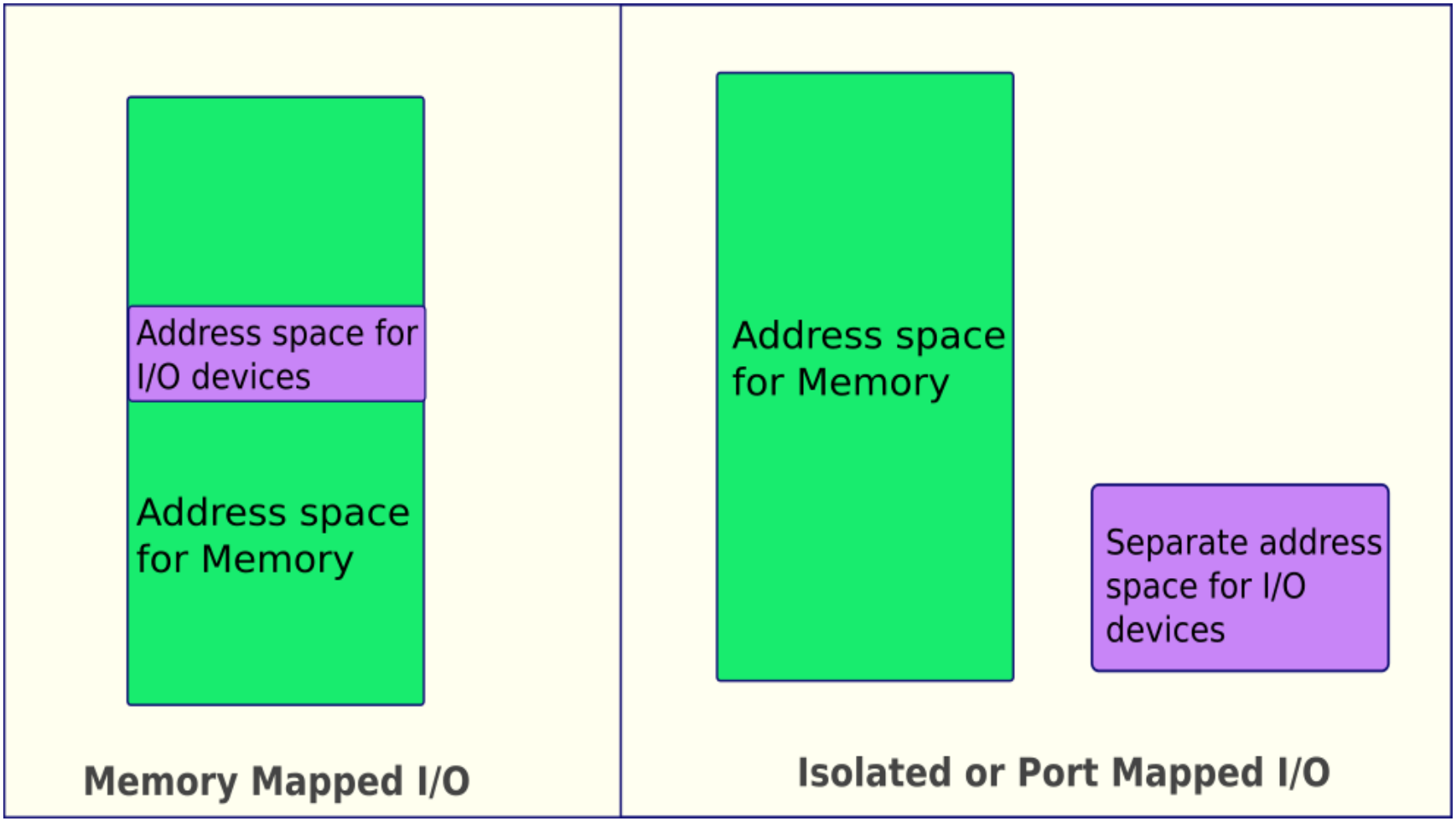


Figure resource: https://www.baeldung.com/cs/memory-mapped-vs-isolated-io

Memory Mapped I/O

Differences between Isolated I/O and Memory Mapped I/O:

Isolated I/O	Memory Mapped I/O
Isolated I/O uses separate memory space.	Memory mapped I/O uses memory from the main memory.
Limited instructions can be used. Those are IN, OUT, INS, OUTS.	Any instruction which references to memory can be used.
The address for Isolated I/O devices are called ports	Memory mapped I/O devices are treated as memory locations on the memory map.

http://inputoutput5822.weebly.com/programmed-io.html

Interrupt Driven I/O

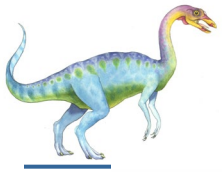
- Clearly, the basic **polling** operation is efficient.
- But **polling** becomes inefficient when an **I/O controller** repeatedly waits for the ***ready status*** of its I/O device**: wasting the CPU's useful processing time**.
- In such instances, it may be more efficient to arrange for the HW controller to notify the CPU when the device becomes **ready** for service rather than the **CPU to poll repeatedly** for the device by itself.
- The HW mechanism that enables a device to notify the CPU when it is **ready** is called an **interrupt.**

Interrupt Driven I/O

- **The basic interrupt mechanism works as follows:**
  - The CPU hardware has an **input pin** (or **line**) called the **interrupt-request line** (**IRQ-line**) that the CPU senses its signal after executing every instruction.
  - When the **CPU** detects that an **I/O controller** has asserted (1) a signal on the **IRQ line**, the CPU performs a state save operation and jumps to an **interrupt-handler routine** (**IHR**) or **ISR (Interrupt Service Routine)** instructions, which are stored at a fixed address in the **OS** memory.
    - **The OS provides the IHR of each device.**
  - The **IHR** determines the cause of the **interrupt** and performs the necessary processing;
    - CPU executes the instructions of **IHR** and then restores the system state to resume the interrupted process.

Interrupt Driven I/O

- The **device controller** raises an **interrupt** by asserting the **IRQ-line** of the CPU, and then the CPU catches the interrupt and dispatches it to the **IHR** of the **OS**, and the handler clears the interrupt by servicing the device.
- **Figure 13.3** summarizes the **interrupt-driven I/O** cycle.
- **Interrupt management** function of the OS is essential:
  - because even a single-user system manages hundreds of interrupts per second, and the server systems handle hundreds of thousands of interrupts per second.



Interrupt-Driven I/O Cycle

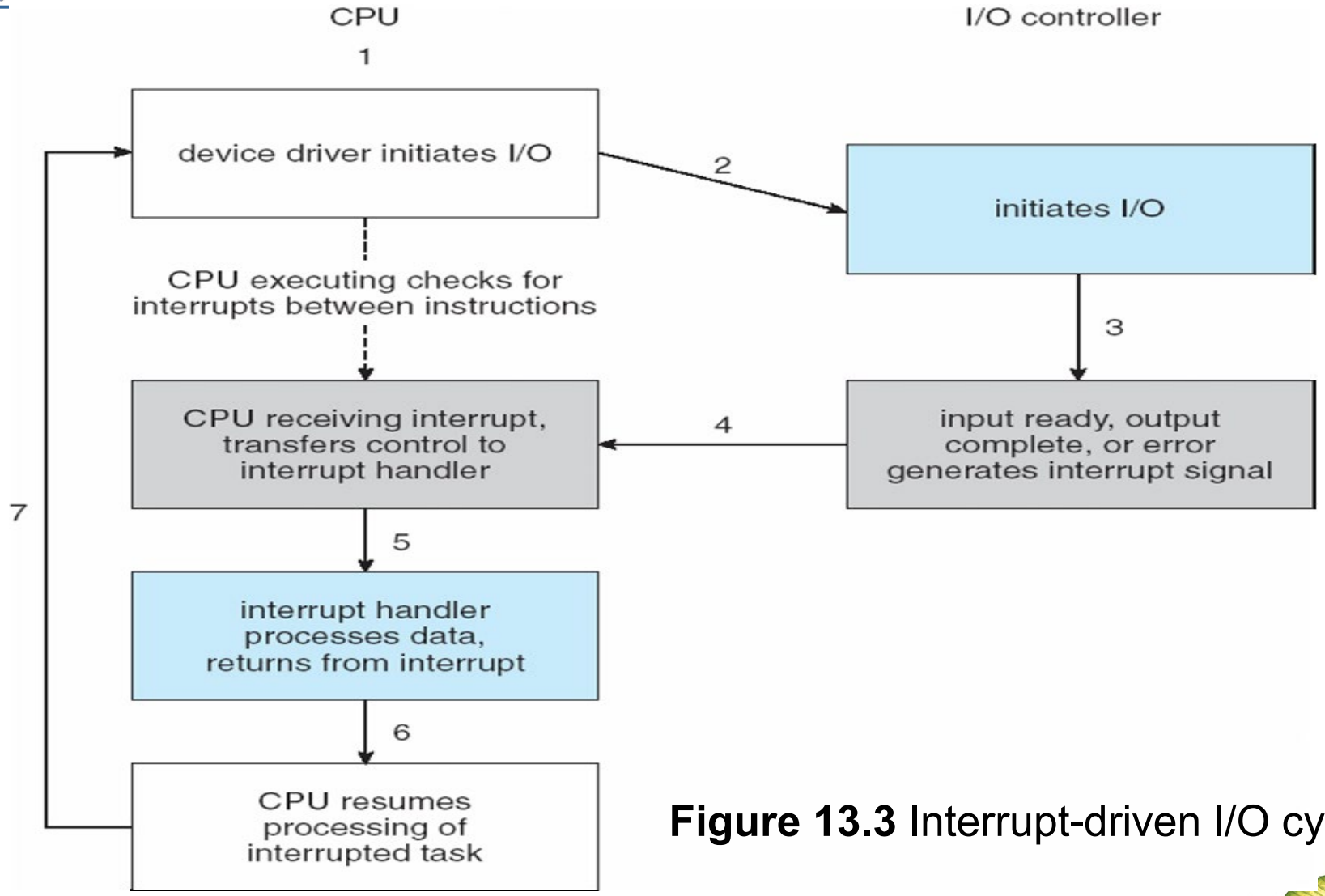
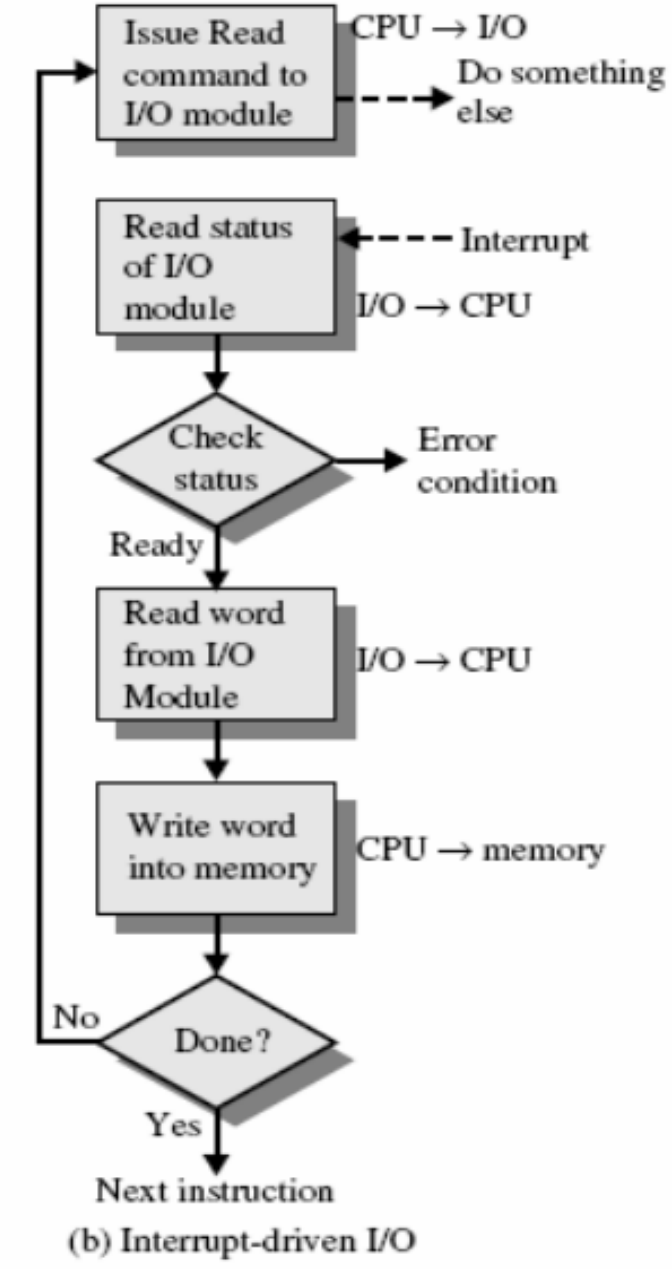


Figure 13.3 Interrupt-driven I/O cycle.



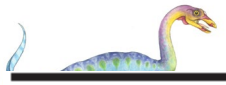
(b) Interrupt-driven I/O

Interrupt Driven I/O

- **Interrupt Driven I/O steps (OS viewpoint):**
- A **system call** is generated when the **CPU is interrupted by an I/O controller**.
- In the meantime, the OS **might block the current process** after receiving the **interrupted message**.
- The OS identifies the I/O device by comparing its interrupt request number with the kernel's **interrupt vector table**.
- This leads to the **ISR (Interrupt Service Routine) for the device by the CPU in the kernel.**
- **The ISR is to execute the driver instructions.**
- The **ISR** is done, and the control is returned to the **kernel I/O subsystem**.
- The OS recovers the blocked process, and the CPU resumes its execution.

Interrupts Types

- Most modern CPUs have **two interrupt request (IRQ)** lines:
  - **Non-maskable interrupt** is reserved for events such as unrecoverable memory errors.
  - **Maskable interrupt** can be **turned off** by the CPU before the execution of critical instruction sequences that must not be interrupted.
- Device controllers use the **maskable interrupt** to request service.



Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Figure A.27 Five categories are used to define what actions are needed for the different exception types shown in Figure A.26. Exceptions that must allow resumption are marked as resume, although the software may often choose to terminate the program. Synchronous, coerced exceptions occurring within instructions that can be resumed are the most difficult to implement. We might expect that memory protection access violations would always result in termination; however, modern operating systems use memory protection to detect events such as the first attempt to use a page or the first write to a page. Thus, CPUs should be able to resume after such exceptions.

Interrupts Vector

- The interrupt mechanism accepts an ***address***—*a number that selects a specific* **interrupt-handling routine**.
- In most architectures, **this address** is an offset in **a table** called the **interrupt vector.**
- **Interrupt vector** contains the memory addresses of specialized **interrupt handlers** for various **I/O devices**.
- The purpose of a **vectored interrupt** mechanism is to search all possible sources of interrupts to determine which one needs service.

Interrupts Types

- **Figure 13.4** 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.

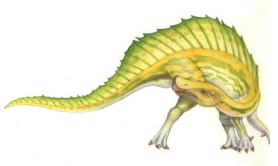
Interrupts

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

Figure 13.4 Intel Pentium processor event-vector table.

Interrupts Types

- 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 (hiding) all interrupts
  - it is possible to preempt a **low-priority interrupt** for a **high-priority interrupt**.
- The **interrupt mechanism** is also used to handle a wide variety of **exceptions;**
  - such as *dividing by* 0, *accessing a protected memory address*, or *attempting to execute a privileged instruction from user mode*.



Page Fault

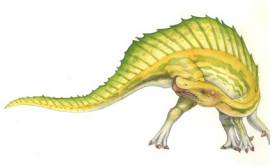
- An **OS** has an efficient HW and SW mechanism that saves the **processor state** and then calls a privileged routine in the **kernel** during an **I/O interrupt**.
- Many operating systems use the **interrupt mechanism** for **virtual memory paging**.
- A **page fault** is an **exception** that results in an **IHR**.
  - This **interrupt** suspends the current process and jumps to the **page-fault handler** in the **kernel**.
  - *This handler saves the state of the interrupted process, moves the process to the **wait queue**, performs **page-cache management**, schedules an I/O operation to fetch the page from the disk (VM), and schedules another process to resume the execution of the process.*

System Calls

- Another example of **kernel interrupt** is the implementation of **system calls**.
- **System call procedure:**
  - A user 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 **trap**.
  - This **system call 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.

Direct Memory Access

- For a large data transfer device such as a **hard disk**, it is wasteful to use the processor to watch **status bits** from time to time for transferring data in the form of ***one byte at a time*** — this type of I/O transaction is called **programmed I/O (PIO)** operation.
- Many computers avoid burdening the CPU with **PIO** (**PIO supports only *one-byte transfer at a time***) for large data transfer.
- Now a day's, large data transfer operations (such as USB transfer) happen in a computer system through a **special-purpose I/O controller** called a **direct-memory-access** controller **(DMA** controller**)**.



Direct Memory Access

- To initiate a **DMA transfer** (assume an **I/O write** operation), the host computer generates a **DMA command block** consisting of the following:
  - ***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.**
- Before a **DMA operation**, the CPU sends the **DMA command block** to the **DMA controller**, then goes on with other work.
- Then, the **DMA controller** proceeds to operate the memory bus directly by placing memory addresses of the data locations on the bus to perform data transfer **without the help of the CPU.**
  - ***A DMA controller is a standard component in all modern computers, including smartphones*.**

Direct Memory Access

- **Pairing** between the **DMA controller** and the **device controller** is performed via the **two control signal** of the **CPU** called **DMA-request (DRQ)** and **DMA-acknowledge** (**DACK**).
  - First, the **DMA controller** collects the **status** of its **I/O device** and sends it to the CPU as **DRQ** for initiating a data transfer.
  - If the CPU is willing to accept the **DRQ** from the **DMA controller**, it sends a **DACK** signal back to the **DMA** controller.
  - It causes the **DMA controller** to seize (called **cycle stealing**) the **system bus** (*address bus, data bus*, and *control bus*) from the CPU and places the desired **memory address** on the **address bus** for data transfer between the **memory** and the **I/O device** meantime the DMA controller cancels the **DRQ** signal.
- When the entire transfer is finished, the **DMA controller** interrupts the CPU. The whole process is depicted in **Figure 13.5.**

Six Step Process to Perform DMA Transfer

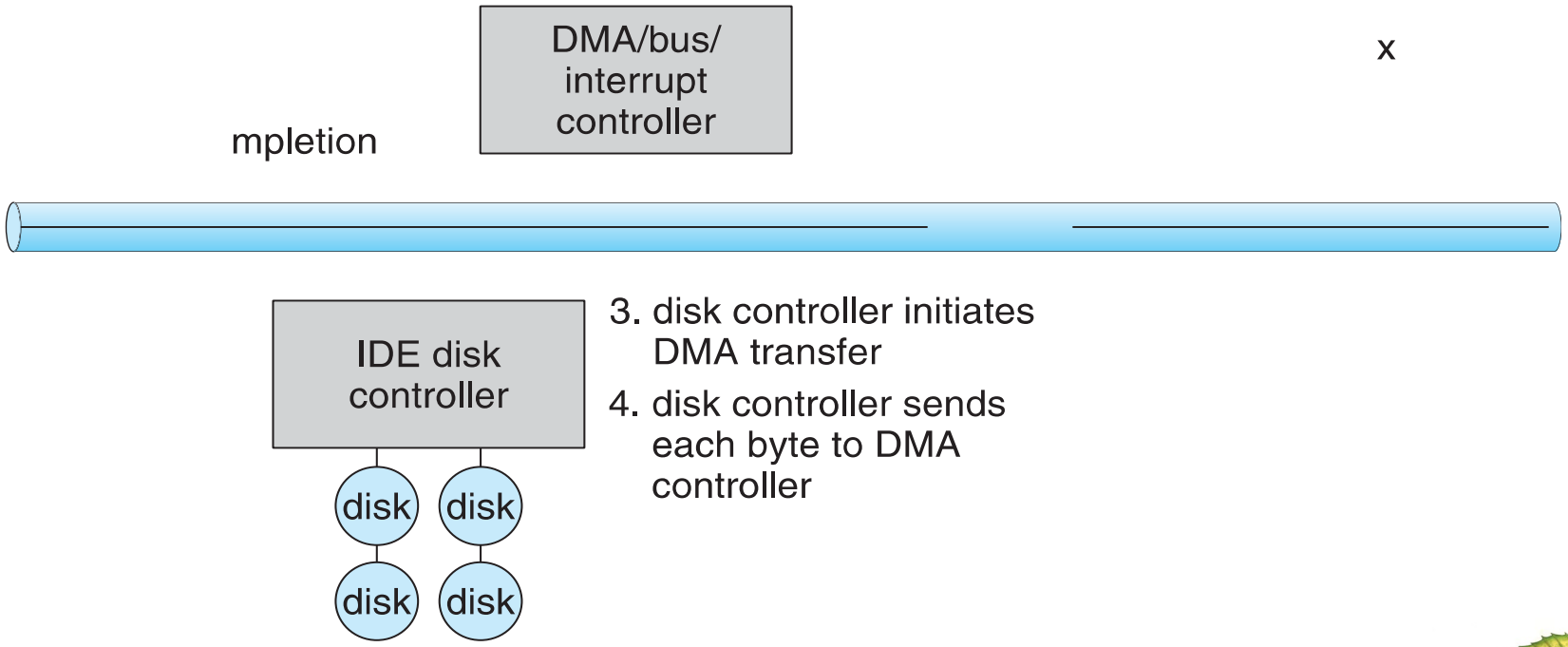
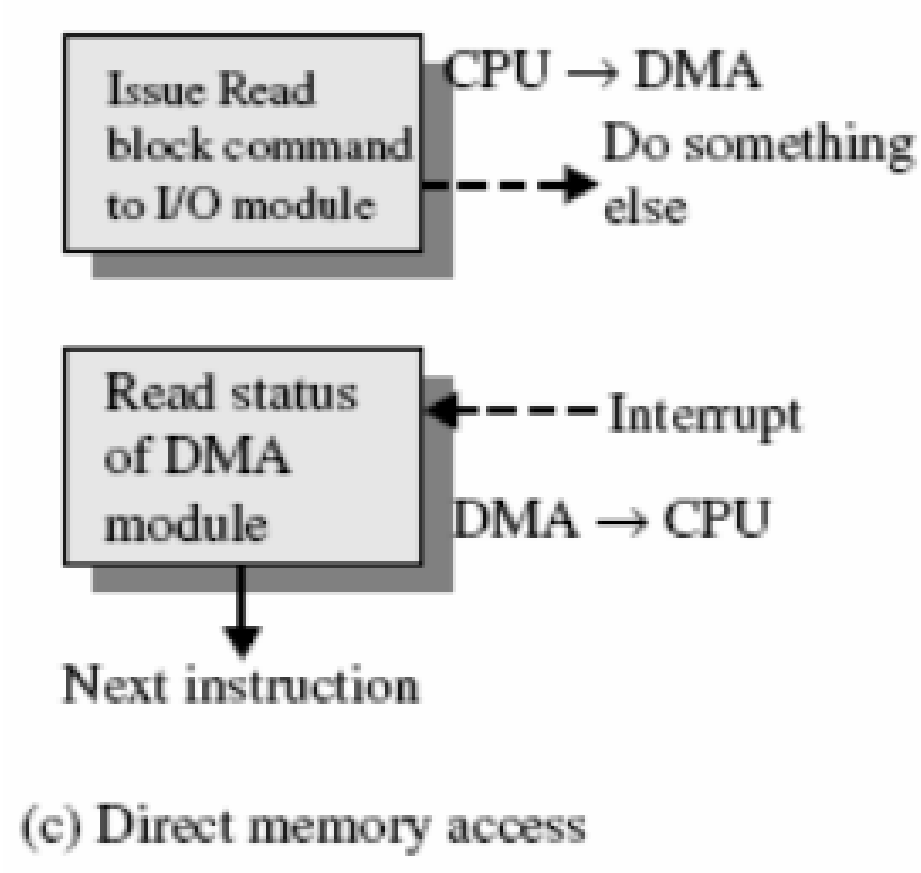


Figure 13.5 Steps in a DMA transfer.

Direct Memory Access



(c) Direct memory access

Application I/O Interface

- This section discusses the *structuring techniques* and *interfaces* for the **OS** that enable **I/O devices** to be treated in a standard and uniform way.
- **How can an application open a file on a disk without knowing what kind of disk it is,** and how can new disks and other devices be added to a computer without disrupting the OS?
- **Figure 13.6** illustrates how the **I/O-related portions of the kernel** are structured in **software layers** (**device-driver layer**):
  - The purpose of the **device-driver layer** is to hide the differences among device controllers from the **I/O subsystem of the kernel.**
  - Making the **I/O subsystem** independent of the HW simplifies the job of the OS developer.

Application I/O Interface

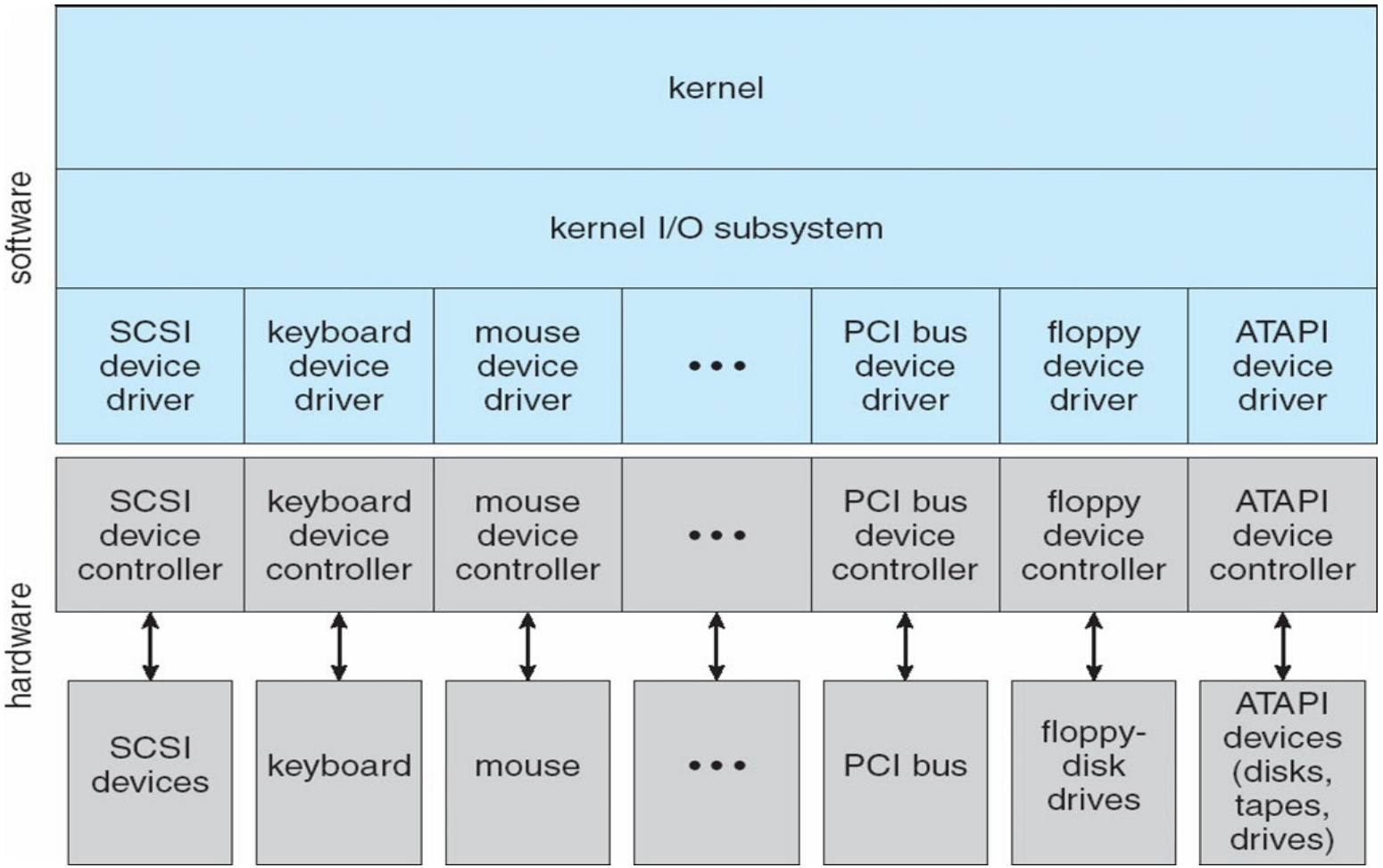


Figure 13.6 A kernel I/O structure.

Application I/O Interface

- For **device-hardware manufacturers**, each type of OS has its standards for the **device-driver interface**.
  - **A given device may ship with multiple device drivers—for instance, for Windows, Linux, AIX, and Mac OS X.**
- **Devices vary on many dimensions**, as illustrated in **Figure 13.7:**
  - **Character-stream** or **block:** Character-stream is the byte by byte transfer, and a block has many bytes
  - **Sequential** or **random access:** In sequential access, data transfer is in a fixed order
  - **Synchronous** or **asynchronous** (or both): if the data transfer is synchronous with the system clock cycle, then it is synchronous else, not
  - **Sharable** or **dedicated:** A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.
  - **Speed of operation:** from a few bytes per sec to a few gigabytes per sec
  - **read-write, read-only** or **write-only:** perform input-output, or support only one data transfer direction (either read or write).

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

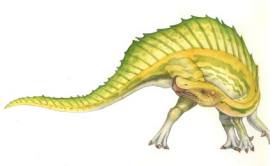
Figure 13.7 Characteristics of I/O devices.

Characteristics of I/O Devices

- Broadly **I/O devices** can be grouped by the **OS** into:
  - **Block device**- Hard Disk, Optical Disk, etc
  - **Character I/O (Stream)**- Keyboard, etc
  - **Memory-mapped file access**- DMA devices such as disk, etc
  - **Network sockets**- client-server accesses

Blocking and Nonblocking System Call

- **System-call** interface relates to **blocking I/O** and **non-blocking I/O**.
- **Blocking I/O system call:**
  - When an application issues a **blocking system call**, the execution of the application is **suspended.**
  - The application is moved from the OS's **run queue** to a **wait queue**.
  - After the **system call** completes, the application is moved back to the **run queue**, where it is eligible to **resume execution**.
  - When it **resumes execution**, it will receive the values returned by the **system call**.
- Nevertheless, most OSs use **blocking system calls** for the application ***interface*** because **blocking application** implementation is easier than **non-blocking** system calls.



Blocking and Nonblocking System Call

- **Non-blocking I/O system calls:**
  - Some OSs provide **nonblocking I/O system calls,** and some user-level processes need **nonblocking I/O**.
  - A **nonblocking call** does not halt the execution of the application for an extended time.
  - Instead, it returns quickly, with a return value that indicates how many bytes were transferred.
  - One example is a user interface that receives *keyboard* and *mouse* input while processing and displaying data on the screen.

Vectored I/O

- A **vectored I/O** allows **one system call** to perform **multiple I/O operations** involving multiple locations.
- For example, the **`UNIXreadv`** system call accepts a vector of multiple buffers and either reads from a source to that vector or writes from that vector to a destination.
- **Multiple separate buffers** can have their contents transferred via one system call, avoiding context-switching and system-call overhead.
- **Advantage:**
  - Without **vectored I/O**, the data might first need to be transferred to a larger buffer in the right order and then transmitted, which is inefficient.

Kernel I/O Subsystem

- **Kernels** provide many services related to **I/O operations.** These services are collectively called a **kernel I/O subsystem**:
- **The kernels I/O subsystem has the following services:**
  - *scheduling*,
  - *buffering*,
  - *caching*,
  - *spooling*,
  - *device reservation*, *and*
  - *Error handling*
- These are provided by the **kernel's I/O subsystem** and build on the hardware and **device driver** infrastructure.
- The **I/O subsystem** is also responsible for protecting itself from errant processes and malicious users.

Kernel I/O Subsystem

- **Scheduling**
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness
  - Some **Implement Quality Of Service** (i.e., IPQOS)
- **Buffering** – store data in memory while transferring between devices
  - To cope with a device speed mismatch
  - To cope with device transfer size mismatch
  - To maintain "copy semantics."
  - **Double buffering** – two copies of the data
    - Kernel and user
    - Varying sizes
    - Full / being processed and not full / being used
    - Copy-on-write can be used for efficiency in some cases

Kernel I/O Subsystem

- **Caching** - faster device holding a copy of data
  - Always just a copy
  - Key to the performance
  - Sometimes combined with buffering
- **Spooling** – hold output for a device
  - If the device can serve only one request at a time
  - i.e., Printing
- **Device reservation** - provides exclusive access to a device
  - System calls for allocation and de-allocation
  - Watch out for deadlock

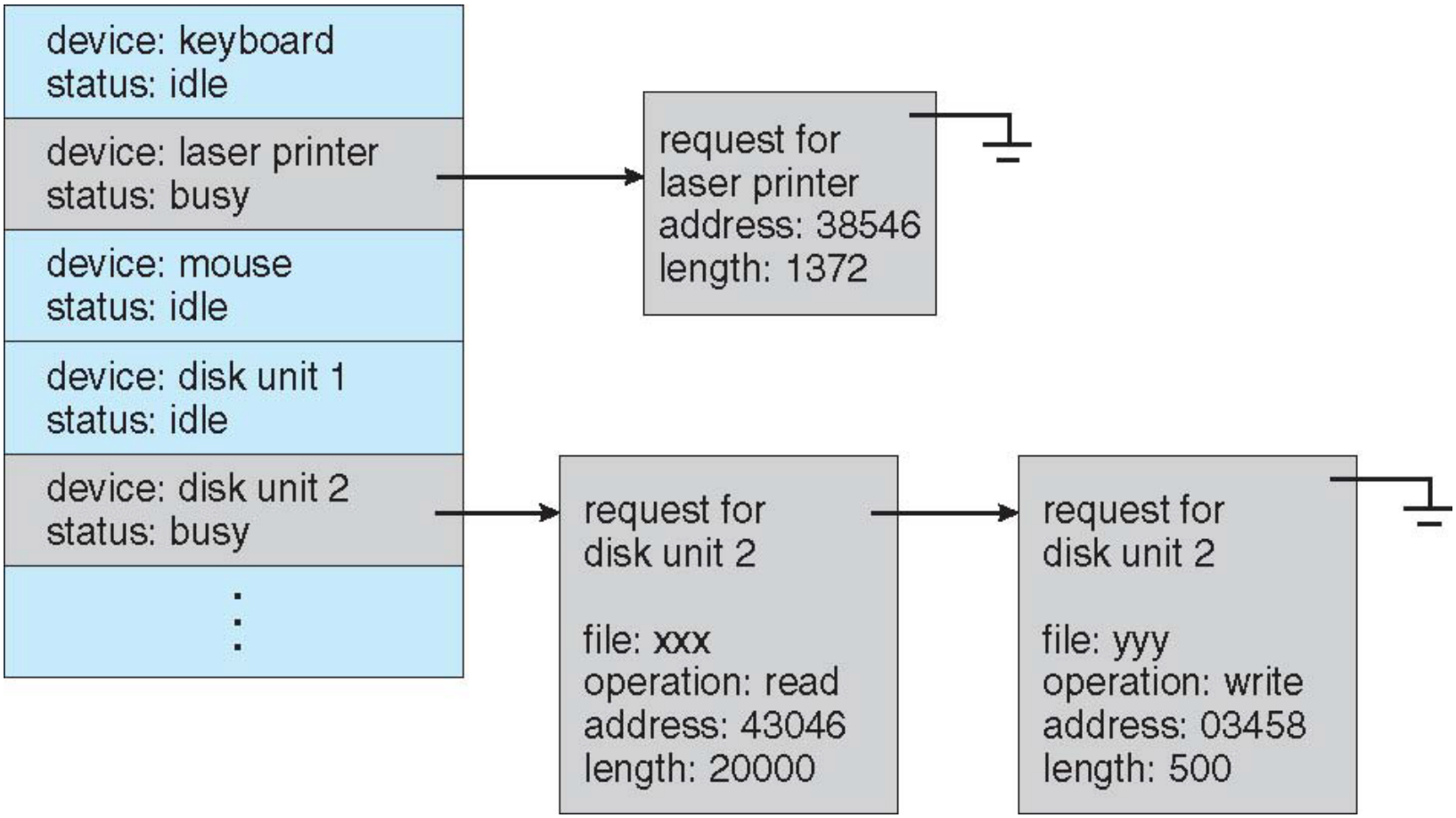
Kernel I/O Subsystem- Scheduling

- **I/O Scheduling**: scheduling a set of I/O requests means determining a good order in which to execute them.
- **I/O Scheduling** can improve overall system performance by sharing devices fairly among processes, reducing the **average waiting time** for I/O to complete.
  - Suppose that a ***disk arm*** is near a disk's beginning and **three applications** issue **blocking** read calls to that disk.
    - Application1 requests a block near the **end** of the disk,
    - application2 requests one block near the **beginning**, and
    - application3 requests one block in the **middle** of the disk.
  - The OS can reduce the distance the disk arm travels by serving the applications in the order 2, 3, and 1.
  - **Rearranging the service order** in this way is the essence of **I/O scheduling**.

Kernel I/O Subsystem Summary

- In summary, the **I/O subsystem** coordinates an extensive collection of services available to applications and other parts of the kernel. In addition, the I/O subsystem supervises these procedures:
  - Management of the namespace for files and devices
  - Access control to files and devices
  - Operation control (for example, a modem cannot seek())
  - File-system space allocation
  - Device allocation
  - Buffering, caching and spooling
  - I/O scheduling
  - Device-status monitoring, error handling, and failure recovery
  - Device-driver configuration and initialization

Device-status Table



Error Handling

- **OS** can recover from ***device unavailable, transient failures*** (write failure)
  - **Retry** a read or write, for example
  - Some systems are more advanced – Solaris FMA, AIX
    - Track error frequencies, stop using devices with increasing frequency of retry-able errors
- Most return an **error number** when an **I/O request** fails
- System error logs hold problem reports

I/O Protection

- To prevent users from performing illegal I/O, we define all **I/O instructions** to be privileged instructions.
- Thus, users cannot issue **I/O instructions** directly; they must do it through the **OS**.
- To do an **I/O operation**, a user program executes a **system call** to request that the OS perform I/O on its behalf (**Figure 13.11**).
- The OS, executing in **monitor mode**, checks that the request is **valid** and, if it is, does the **I/O requested**.
- The OS then returns the control to the user.

Use of a System Call to Perform I/O

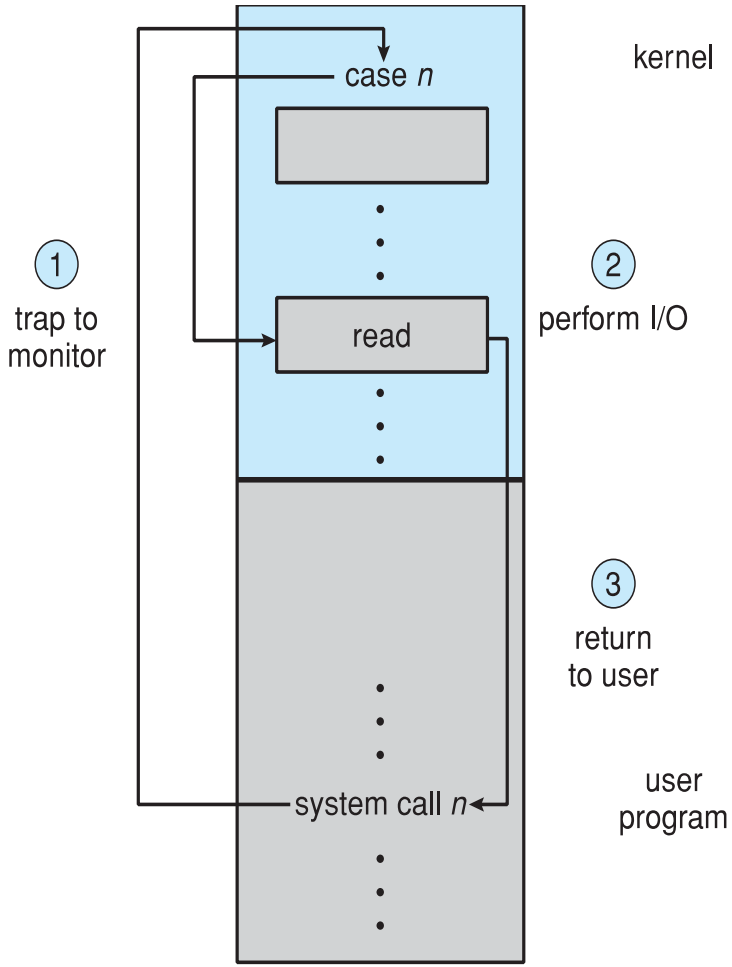


Figure 13.11 Use of a system call to perform I/O.

Improving I/O Performance

- We can employ several principles to improve the **efficiency of the I/O interface:**
  - Reduce the number of **context switches.**
  - Reduce the number of **data copying** in memory while passing between device and application.
  - Reduce the **frequency of interrupts** by using large transfers, smart controllers, and polling.
  - Use the **DMA I/O interface for** large data transfer.
  - Use **smarter hardware** for the I/O controller.
  - Balance CPU, memory subsystem, bus, and I/O performance, for higher throughput.
  - Move **user-mode** processes to **kernel threads**

Life Cycle of An I/O Request