CSE308 Operating Systems

I/O Systems

S.Rajarajan

SASTRA

- The two main jobs of a computer are **I/O and processing.**
- The role of the operating system in computer I/O is **to manage and control I/O operations and I/O devices.**
- I/O management is a major component of operating system design and operation
  - Important aspect of computer operation
  - I/O devices **vary greatly**
  - **Various methods used** to control them
  - **Performance** management
  - **New types of devices** introduced frequently
- The role of the operating system in computer I/O is **to manage and control I/O operations and I/O devices.**

Overview

- 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**.

Device Drivers

- To encapsulate the **details and oddities** of **different devices**, the kernel of an operating system is structured to **use device-driver modules.**
- Present a **uniform device access interfac**e to the **I/O subsystem**, much as **system calls** provide a standard interface between the **application** and **the operating system**

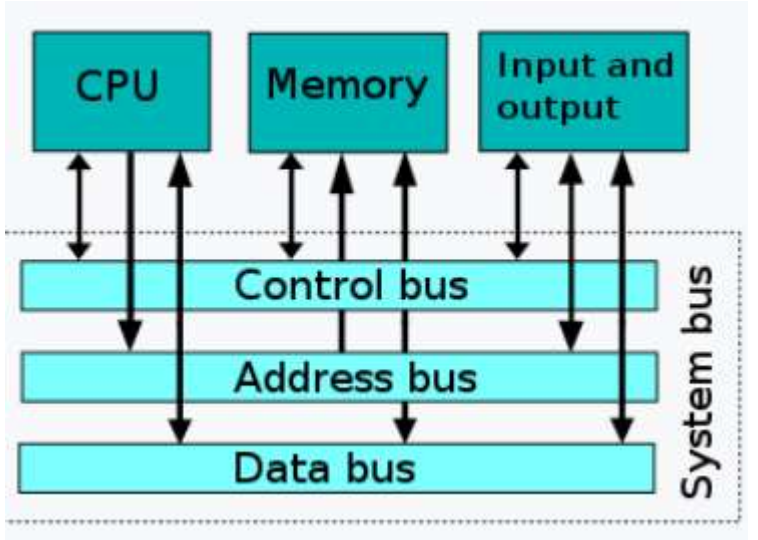
I/O Hardware

- Computers operate a great **many kinds of devices**.
- Most fit into the general categories of
  - **storage devices** (disks, tapes)
  - **transmission devices** (network connections, Bluetooth),
  - and **human-interface devices** (screen, keyboard, mouse, audio in and out).
- Despite the variety of I/O devices, though, we **need only a few concepts to understand** how the **devices are attached** and **how the software** can control the hardware.

- A **device communicates** with a computer system by **sending signals** over a **cable** or even **wireless**
- Common ways for transmission of signals from I/O devices interface with computer
  - **Port** – The device communicates with the machine via a connection point, or port—for example, a serial port
  - **Bus** - 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.
  - **daisy chain-** 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
  - **Controller** (**host adapter**) – is a collection of electronics that can operate a port, a bus, or a device.

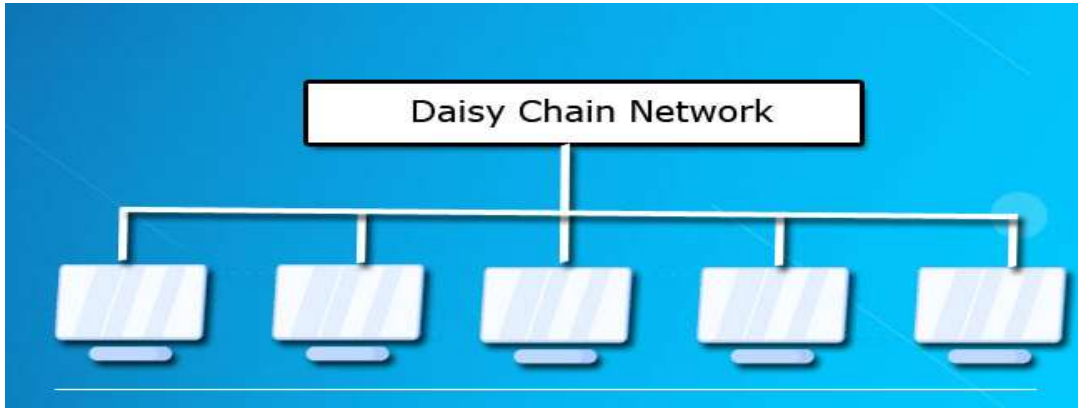
Bus

- 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.
- In terms of the electronics, the messages are **conveyed by patterns of electrical voltages** applied to the wires **with defined timings**.



Daisy chain

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



PCI Bus architecture

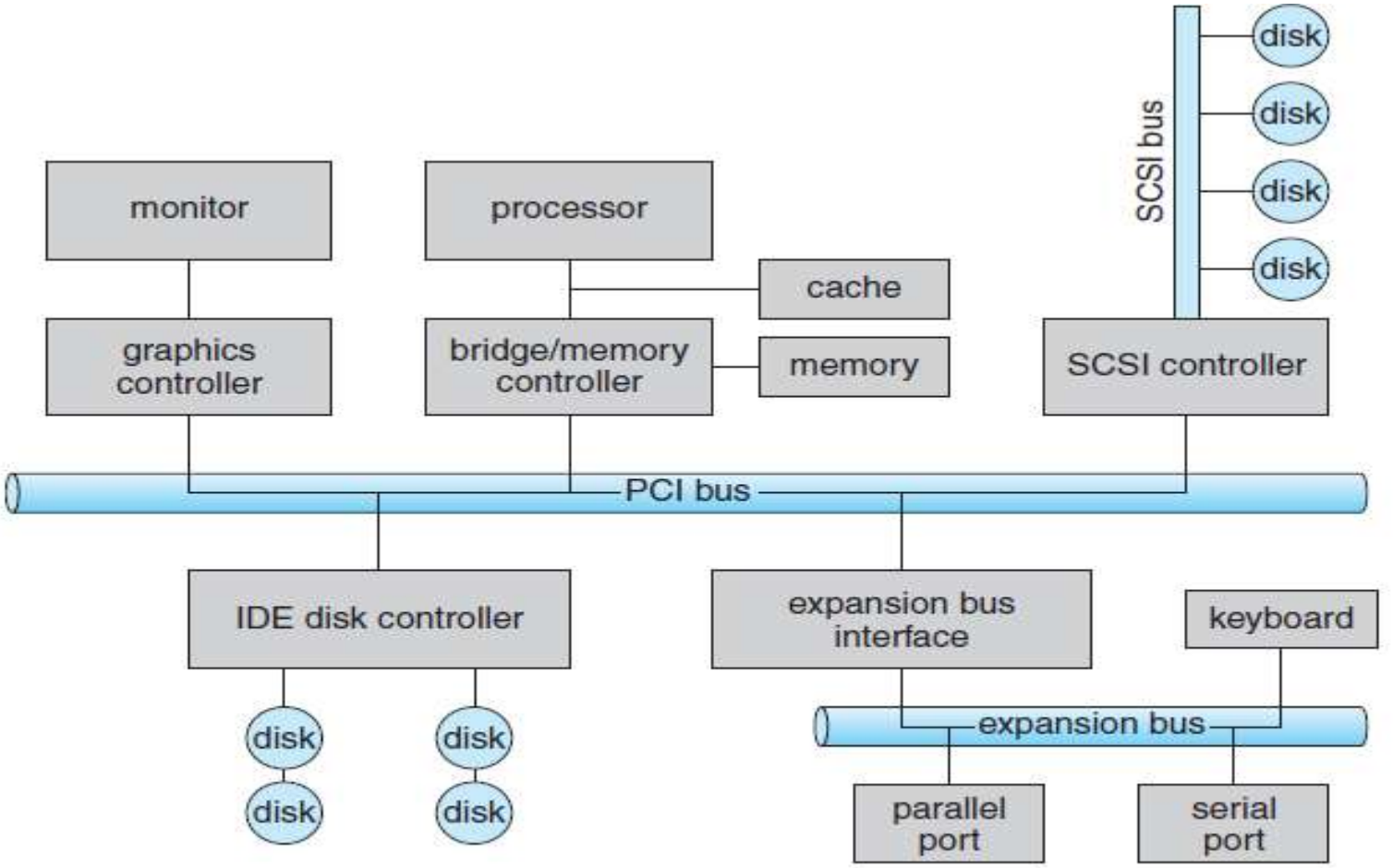


Figure 13.1 A typical PC bus structure.

PCI Bus

- 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).**
- **HyperTransport** bus has throughput of up to **25 GB per second.**

Controller

- A **controller** is a collection of electronics that can **operate a port or a bus** , or **a device**
  - A **serial-port controller** is a simple device controller. It is a **single chip** (or portion of a chip) **intergraded** in the computer that **controls the signals** on the wires of a serial port
  - By contrast, a **SCSI bus controller** is not simple. Because the SCSI protocol is complex, the SCSI bus controller is often implemented as a **separate circuit board** (or a host adapter) that **plugs into the computer**. Contains **processor, microcode, private memory, bus controller**, etc

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

- The short answer is that 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.

Isolated I/O or Direct I/O instructions

- 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 (**
- The I/O instruction **triggers bus lines to select the proper device** and **to move bits** into or out of a device register.

Memory-mapped I/O

- Alternatively, the device controller can support **memory-mapped I/O.**
- In this case**, the device-control registers** are **mapped into the address space** of the processor.
- The CPU executes **I/O requests** using the **standard data-transfer instructions** to read and write the **device-control registers at their mapped locations** in physical memory.
- Some systems use both techniques.

- 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. For instance, a certain bit in the control register of a serial port **chooses between full-duplex and half-duplex** communication, another bit enables **parity checking**, a third bit **sets the word length to 7 or 8 bits,** and other bits select one of the **speeds** supported by the serial port.

- 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 those data.

Polling

- The complete protocol for interaction **between the host** and a **controller** can be intricate, but the basic handshaking notion is simple.
- The controller indicates its state through the **busy bit** in the status register.
- **Sets busy bit** when it is **busy working** and **clears** the busy bit **when it is ready**
- The **host** signals its wishes via the **command-ready bit** in the command register
- The host **sets the command-ready bit** when a **command is available** to execute.

- The host coordinating with the controller by **handshaking** as follows
- For each byte of I/O
- **1.** The host repeatedly **reads the busy bit** until that bit becomes clear.
- **2.** The host **sets the write bit in the command register** and writes a byte into **the data-out register.**
- **3.** The host **sets the command-ready bit.**
- **4.** When the **controller notices** that the command-ready bit is set, it sets the busy bit.
- **5.** 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**.
- **6**. 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

- In step 1, 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.
- If the **controller and device are fast**, this method is **a reasonable** one.
- In many computer architectures, **three CPU-instruction cycles are sufficient to poll a device**: read a device register, logical--and to extract a status bit, and branch if not zero.
- But if the **wait is too long**, then the host should probably **switch to another task.**
- How, then, does the host know when the controller has become idle?

Interrupts

- 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.

- 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 interrupt*** handler, and the handler ***clears the interrupt by servicing the device.***
- The basic interrupt mechanism just described enables the CPU **to respond to an asynchronous event**, as when a device controller becomes ready for service..

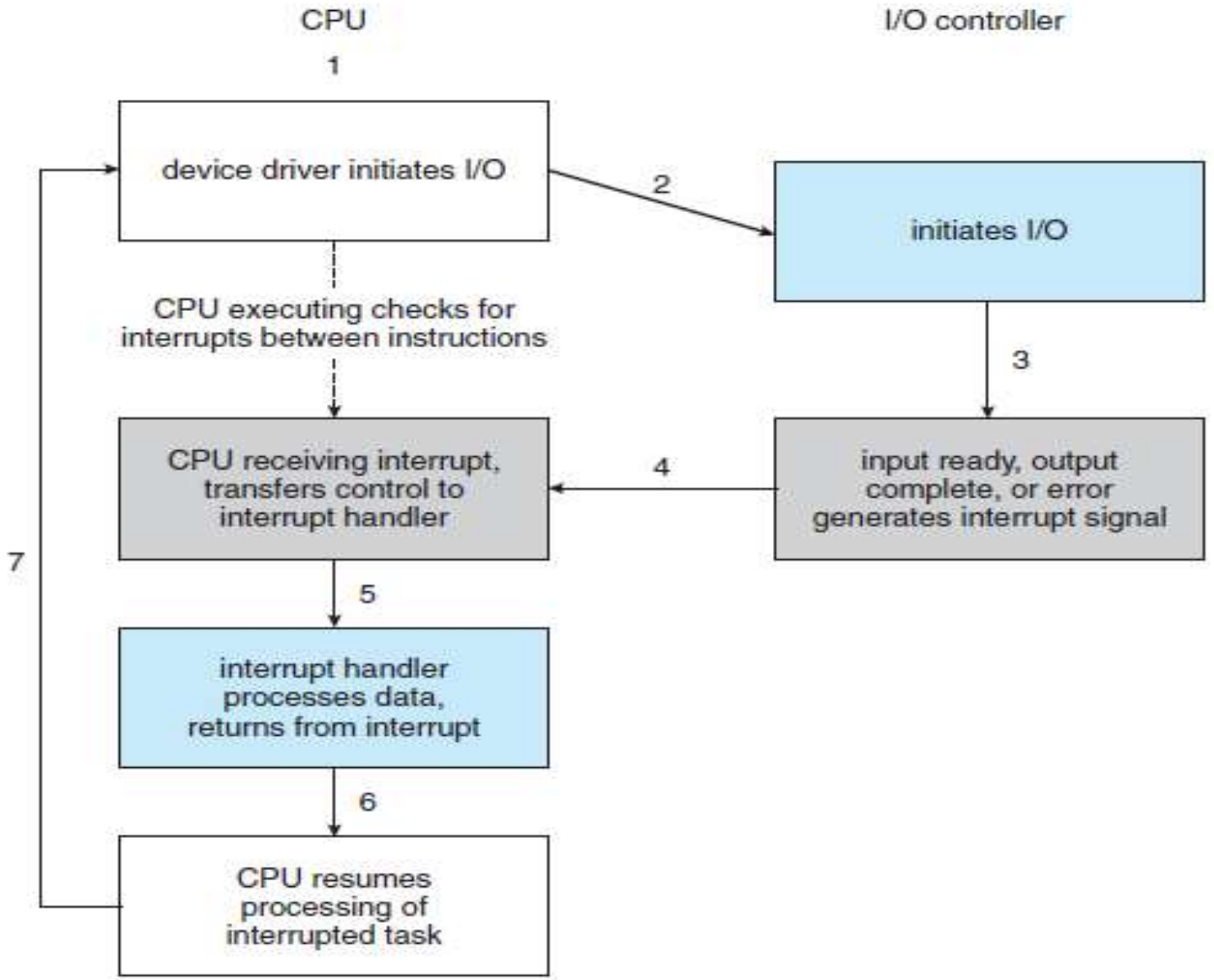


Figure 13.3 Interrupt-driven I/O cycle.

- In a modern operating system, however, we need more sophisticated interrupt-handling features.
- **1.** We need the ability to **defer interrupt handling during critical processing.**
- **2**. 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.
- **3.** We need **multilevel interrupts,** so that the operating system can distinguish between **high- and low-priority interrupts** and can respond with the appropriate **degree of urgency.**
- In modern computer hardware, these three features are provided by the CPU and by the **interrupt-controller hardware.**

Interrupt request lines

- Most CPUs have two interrupt request lines.
- One is the **non-maskable 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.

Which device made the interrupt ?

- 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.
- The purpose of a **vectored interrupt mechanism** is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service.

Interrupt chaining

- In practice, however, **computers have more devices** (and, **hence, interrupt handlers**) than they have address elements in the interrupt vector.
- A common way to solve this problem is to use **interrupt chaining (linked list) 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.

Multilevel 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

Exceptions

- 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, terminate process, crash system due to **hardware error**
- A **page fault** is an exception that raises an interrupt.
- Another example is the implementation of **system calls**.
- Usually, a program uses library calls to issue system calls.
- Library routines **check arguments** given by the application, **build a data structure** to convey the arguments to the kernel, and execute a special instruction called software **interrupt, or trap**

- 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

Direct Memory Access

- For a device that does large transfers, such as a disk drive, it seems **wasteful to use 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** 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 smart phones to mainframes.
- **Handshaking** between the DMA controller and the device controller is performed via a pair of wires called **DMA-request and DMA-acknowledge.**

- When the DMA controller seizes the memory bus, the CPU is momentarily prevented from accessing main memory – **cycle stealing.**

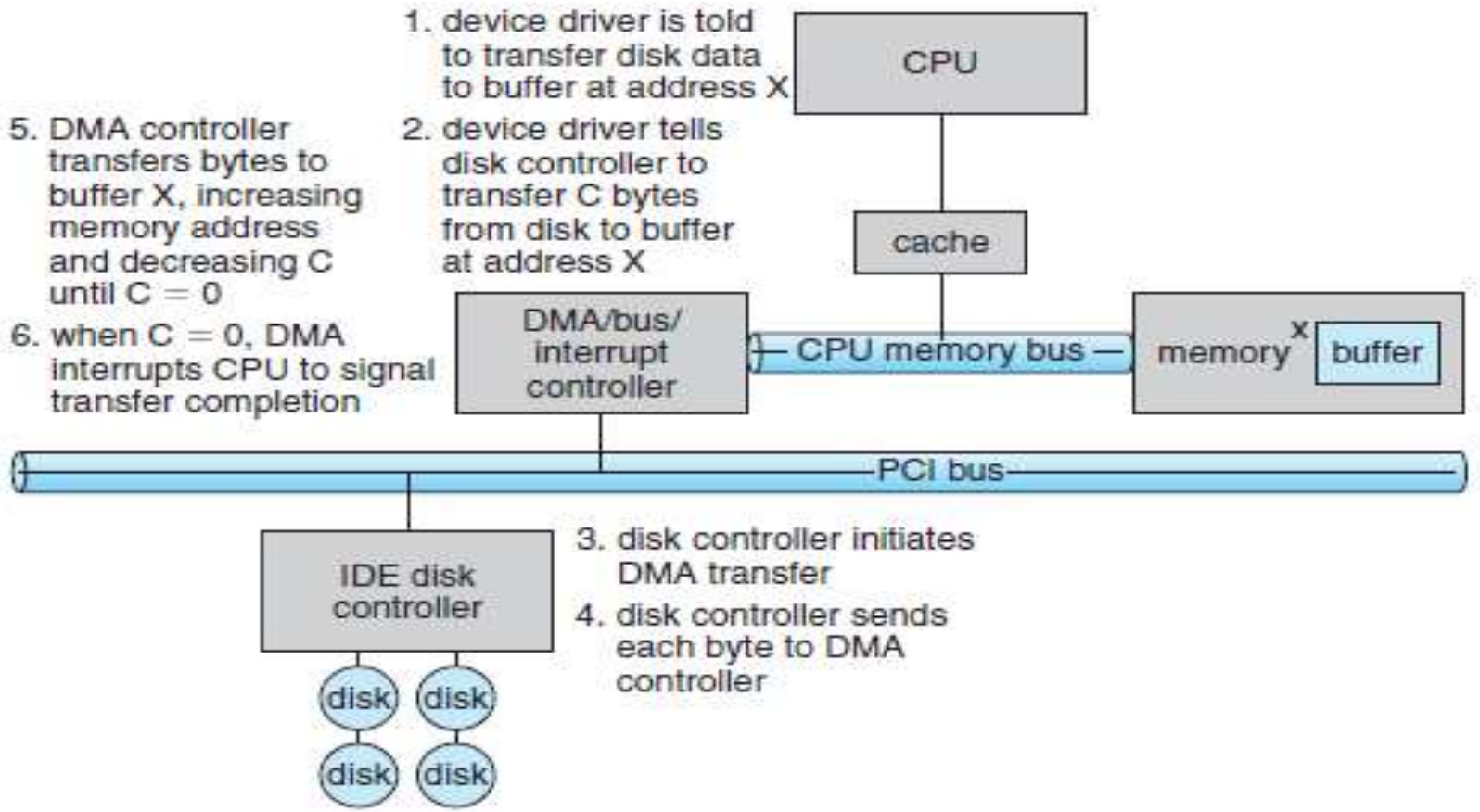


Figure 13.5 Steps in a DMA transfer.

Application I/O Interface

- Like other complex software-engineering problems, the approach here **involves abstraction**, encapsulation, and software layering.
- Specifically, we can **abstract away the detailed differences in I/O devices** by identifying a few general kinds.
- Each general kind is accessed through a standardized set of functions—an **interface**

- The purpose of the **device-driver layer is to hide the differences among device controllers** from the I/O subsystem of the kernel, much as the I/O system calls encapsulate the behavior of devices in a few generic classes that hide hardware differences from applications.
- Making the I/O subsystem independent of the hardware **simplifies the job of the operating-system** developer.
- It also benefits the hardware manufacturers.
- They either design new devices to be compatible with an existing host controller interface (such as SATA), or they write device drivers to interface the new hardware to popular operating systems.

Devices vary on many dimensions

- Devices vary in many dimensions
  - **Character-stream** or **block**
  - **Sequential** or **random-access**
  - **Synchronous** or **asynchronous** (or both)
  - **Sharable** or **dedicated**
  - **Speed of operation**
  - **read-write, read only,** or **write only**

Characteristics of I/O Devices

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Block and Character Devices

- The **block-device interface captures all the aspects necessary for accessing disk** drives and other **block-oriented devices**.
- The device is expected to understand commands such as **read() and write();** if it is a **random-access device**, it is also expected to have a **seek() command** to specify which block to transfer next.
- Applications normally access such a device through a **file-system interface.**
- Operating may prefer to access a block device as a **simple linear array of blocks.**
- This mode of access is sometimes called **raw I/O.**

- A compromise that is becoming common is for the operating system to allow a mode of operation on a file that **disables buffering and locking.**
- In the UNIX world, this is called **direct I/O.**
- A **keyboard** is an example of a device that is accessed through a **character stream interface.**
- The basic system calls in this interface enable an application to **get() or put()** one character.
- On top of this interface, libraries can be built that offer line-at-a-time access, with buffering and editing services

Network Devices

- Because the performance and addressing characteristics of network I/O differ significantly from those of disk I/O, most operating systems provide a network I/Ointerface that is different from the read()–write()–seek() interface used for disks.
- One interface available in many operating systems, including UNIX and Windows NT, is the network **socket interface.**
- By analogy, the **system calls in the socket interface** enable an application **to create a socket**, to **connect a local socket to a remote address** to listen for any **remote application to plug into the local socket**, and to **send and receive packets** over the connection.

Clocks and Timers

- Most computers have hardware clocks and timers that provide three basic functions:
  - Give the **current time.**
  - Give the elapsed time.
  - **Set a timer** to trigger operation *X at time T.*
- These functions are used heavily by the operating system, as well as by time sensitive applications.
- Unfortunately, the system calls that implement these functions are not standardized across operating systems

Blocking and Non-blocking I/O

- Another aspect of the system-call interface relates to the choice between blocking I/O and non-blocking I/O.
- When an application issues a **blocking** system call, the **execution of the application is suspended**.
- The application is moved from the operating system's **run queue to a wait queue**.
- After the system call completes, the application is **moved back to the run queue**.
- When it resumes execution, it will **receive the values returned by the system call**.

- Some user-level processes need **nonblocking I/O.**
- One example is a **user interface that receives keyboard and mouse input** while **processing and displaying data on the screen**.
- Another example is a **video application** that reads frames from a file on disk while simultaneously decompressing and displaying the output on the display.
- One way an application writer can **overlap execution with I/O** is to write a **multithreaded application.** Some threads can perform blocking system calls, while others continue executing

- A good example of non-blocking behavior is the **select()** system call for **network sockets**
- This system call takes an **argument that specifies a maximum waiting time**.
- By **setting it to 0**, an application can poll for network activity **without blocking.**
- But using select() introduces extra overhead, because the select() call only checks whether I/O is possible.
- For a data transfer, select() must be followed by some kind of **read() or write()** command.

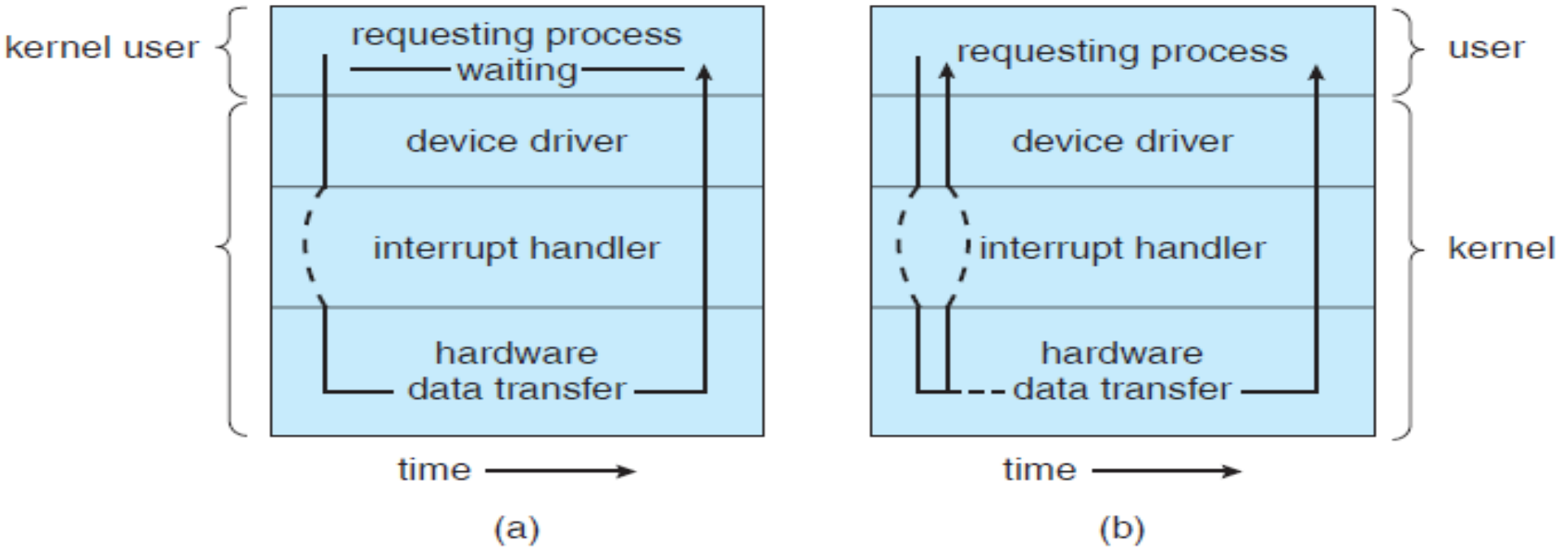


Figure 12.8 Two I/O methods: (a) synchronous and (b) asynchronous.

Kernel I/O Subsystem

- Kernels provide **many services related to I/O**.
- Several **services—**
  - **I/O scheduling**
  - **Buffering**
  - **Caching**
  - **Spooling**
  - **Device reservation**
  - **and error handling**—are provided by the kernel's I/O subsystem.
- The I/O subsystem is also **responsible for protecting itself** from errant processes and malicious users.

I/O Scheduling

- To schedule a set of I/O requests means to **determine a good order** in which to execute them.
- Scheduling can improve overall system performance, can **share device access fairly** among processes, and can **reduce the average waiting time for I/O** to complete.
- **Disk scheduling** is an example.
- Operating-system developers implement scheduling **by maintaining await queue** of requests for each device

- When a kernel supports **asynchronous I/O**, it must be able to **keep track of many I/O requests** at the same time.
- For this purpose, the operating system might attach the wait queue to a **device-status table.**
- **The kernel manages this** table, which contains an entry for each I/O device.

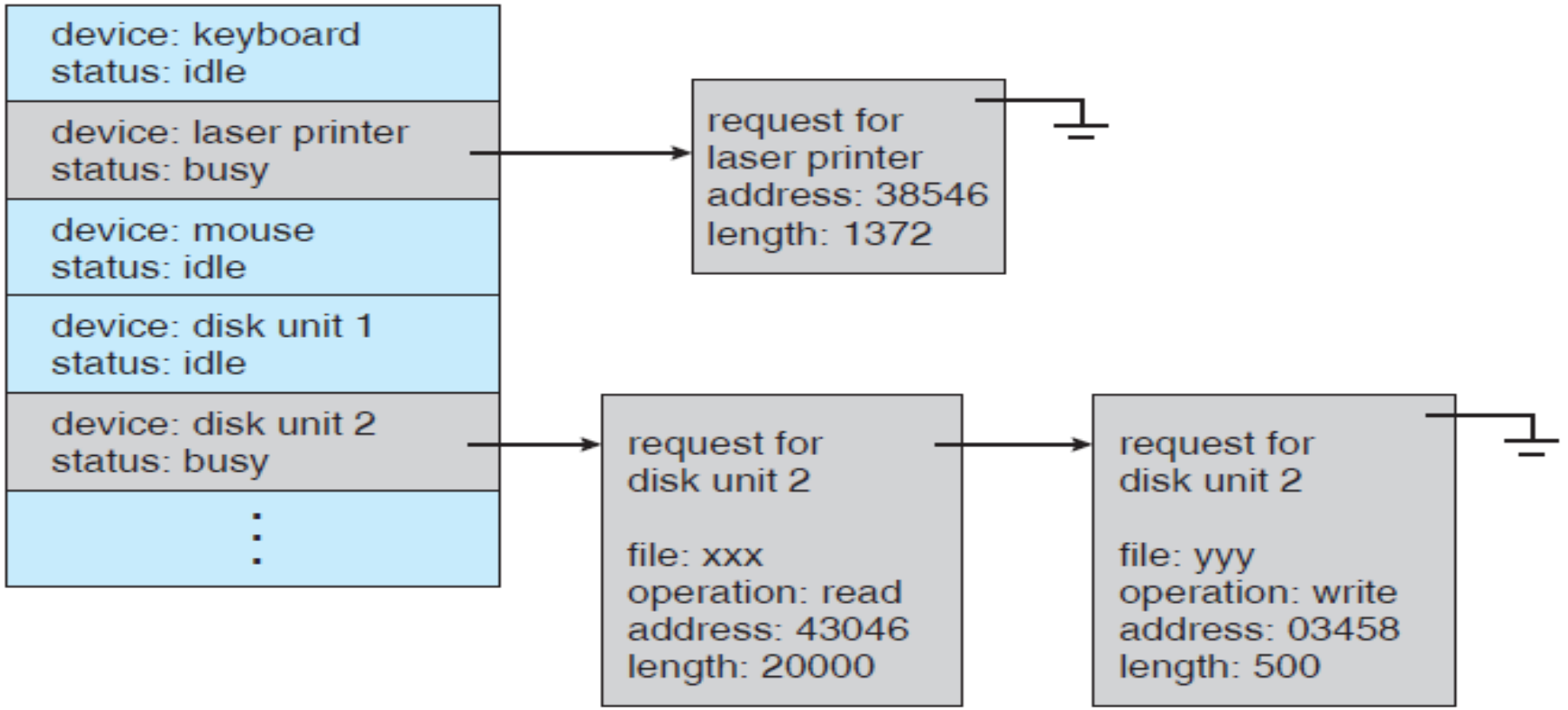


Figure 12.9 Device-status table.

Buffering

- A **buffer is a memory area that stores data being transferred between two** devices or between a device and an application.
- Buffering is done for **three reasons**
- Done for three reasons
  - To cope with a **speed mismatch** between the producer and consumer of a data stream
  - To provide adaptations for **devices** that have **different data-transfer sizes**
  - To support **copy semantics** for application I/O - The disk write is **performed from the kernel buffer** instead of from **the device buffer**, so that subsequent changes to the application buffer have no effect.

Copy Semantics

- Suppose that an **application has a buffer of data** that it wishes **to write to disk.**
- It calls the **write() system call**, providing a **pointer to the buffer** and an **integer** specifying the **number of bytes to write.**
- After the system call returns, what happens **if the application changes** the **contents of the buffer**?
- With **copy semantics,** the **version of the data** written to disk is **guaranteed to be the version at the time of the application system call**, independent of any subsequent changes in the application's buffer.

- A simple way in which the operating system can guarantee copy semantics is for the write() system call **to copy the application data into a kernel buffer** before returning control to the application.
- The **disk write is performed from the kernel buffer,** so that subsequent changes to the application buffer have no effect.

Caching

- A **cache** is a region of **fast memory** that holds copies of data.
- Access to the cached copy is more efficient than access to the original.
- For instance, the instructions of the currently running process are stored on disk, cached in physical memory, and copied again in the CPU's secondary and primary caches.
- The **difference between a buffer and a cache** is that a buffer holds the existing copy of a data item in **main memory,** whereas a cache, by definition, holds a copy **on faster storage** of an item.

- Also, **disk writes** are accumulated in the buffer cache for several seconds, so that large transfers are gathered to allow efficient write schedules.

Spooling and Device Reservation

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

Error Handling

- An operating system that uses protected memory can **guard against many kinds of hardware and application errors**, so that a **complete system failure is prevented**.
- Devices and I/O transfers **can fail in many ways**, either for **transient reasons**, as when a *network becomes overloaded*, or for "**permanent" reasons**, as when a **disk controller becomes defective**.
- Operating systems can often **compensate effectively for transient failures.**
- For instance, a disk read() failure results in a read() retry, and a network send() error results in a resend(), if the protocol so specifies.
- Unfortunately, if an important component experiences a **permanent failure**, the operating system is **unlikely to recover.**

I/O Protection

- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
  - All **I/O instructions defined to be privileged**
  - I/O must be **performed via system calls**

Kernel Data Structures

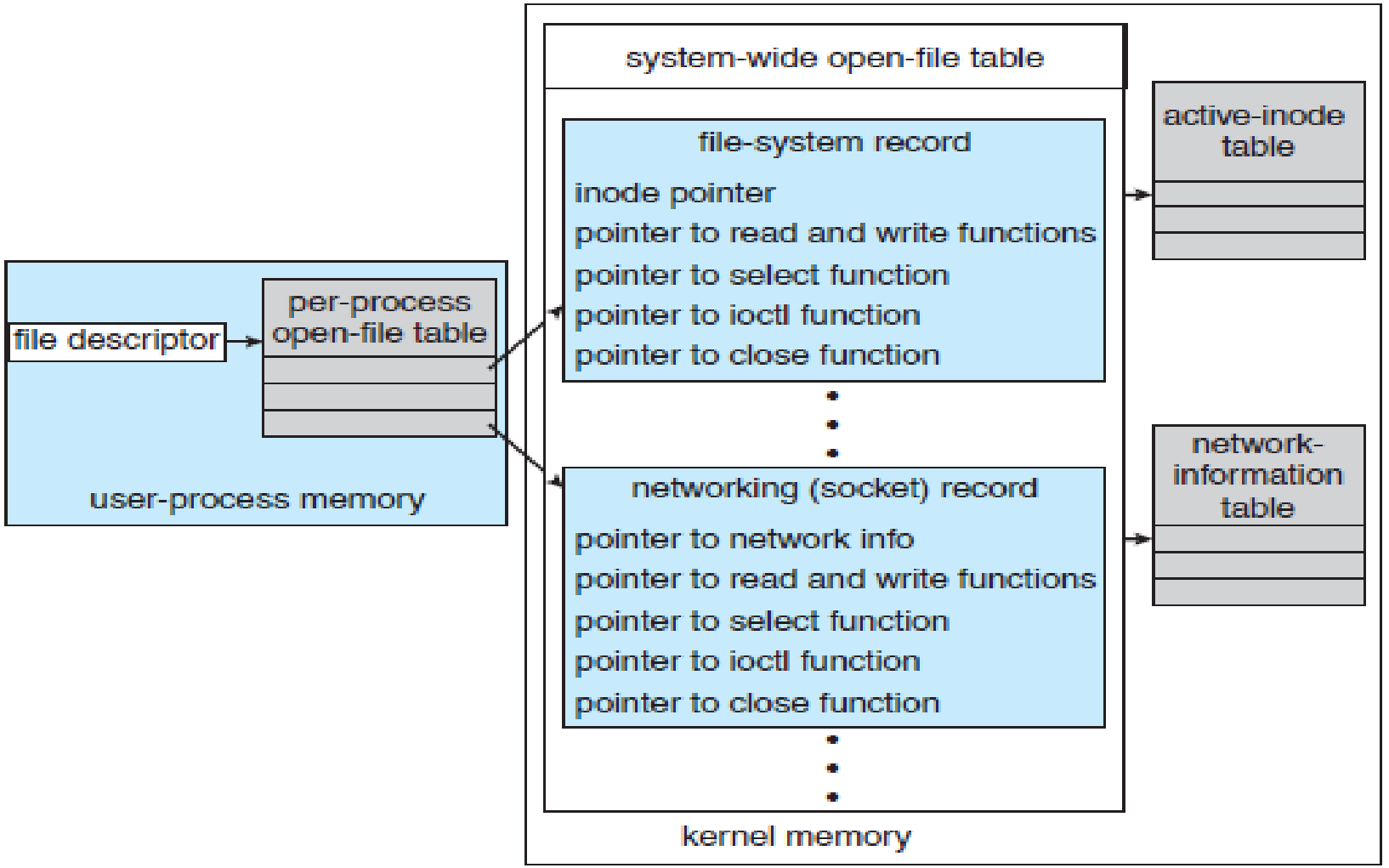


Figure 12.12 UNIX I/O kernel structure.