UNIT-6
Input/Output Systems

:Mr. S. C. Sagare

CONTENTS

- Overview
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O requests to hardware Operations
- Streams
- Performance

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.
  - E.g. when we browse a web page or edit a file, our immediate interest is to read or enter some information, not to compute an answer.

- The role of the operating system in computer I/O is to manage and control **I/O operations** and **I/O devices**.

Introduction

- The control of devices connected to the computer is a major concern of operating-system designers.
- Variety of methods are needed to control the I/O devices because these devices vary so widely in their function and speed (e.g. mouse, a hard disk, etc.)
- 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.

- I/O device technology exhibits two conflicting trends.
  - On the one hand, we see **increasing standardization of software and hardware interfaces**. This trend helps us to incorporate improved device generations into existing computers and operating systems.

  - On the other hand, we see an **increasingly broad variety of I/O devices**. Some new devices are so unlike previous devices that it is a challenge to incorporate them into our computers and operating systems. This challenge is met by a combination of hardware and software techniques.
- 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

- I/O devices can be roughly categorized as storage, communications, user-interface, and other
- Devices communicate with the computer via signals sent over wires or through the air.
- Devices connect with the computer via ***ports***, e.g. a serial or parallel 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.
  - In terms of the electronics, the messages are conveyed by patterns of electrical voltages applied to the wires with defined timings.

- Four bus types commonly found in a modern PC
  - The ***PCI bus*** connects high-speed high-bandwidth devices to the memory subsystem (and the CPU.)
  - The ***expansion bus*** connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering.)
  - The ***SCSI bus*** connects a number of SCSI devices to a common SCSI controller.
  - A ***daisy-chain bus,*** is when a string of devices is connected to each other like beads on a chain, and only one of the devices is directly connected to the host.
    - 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**

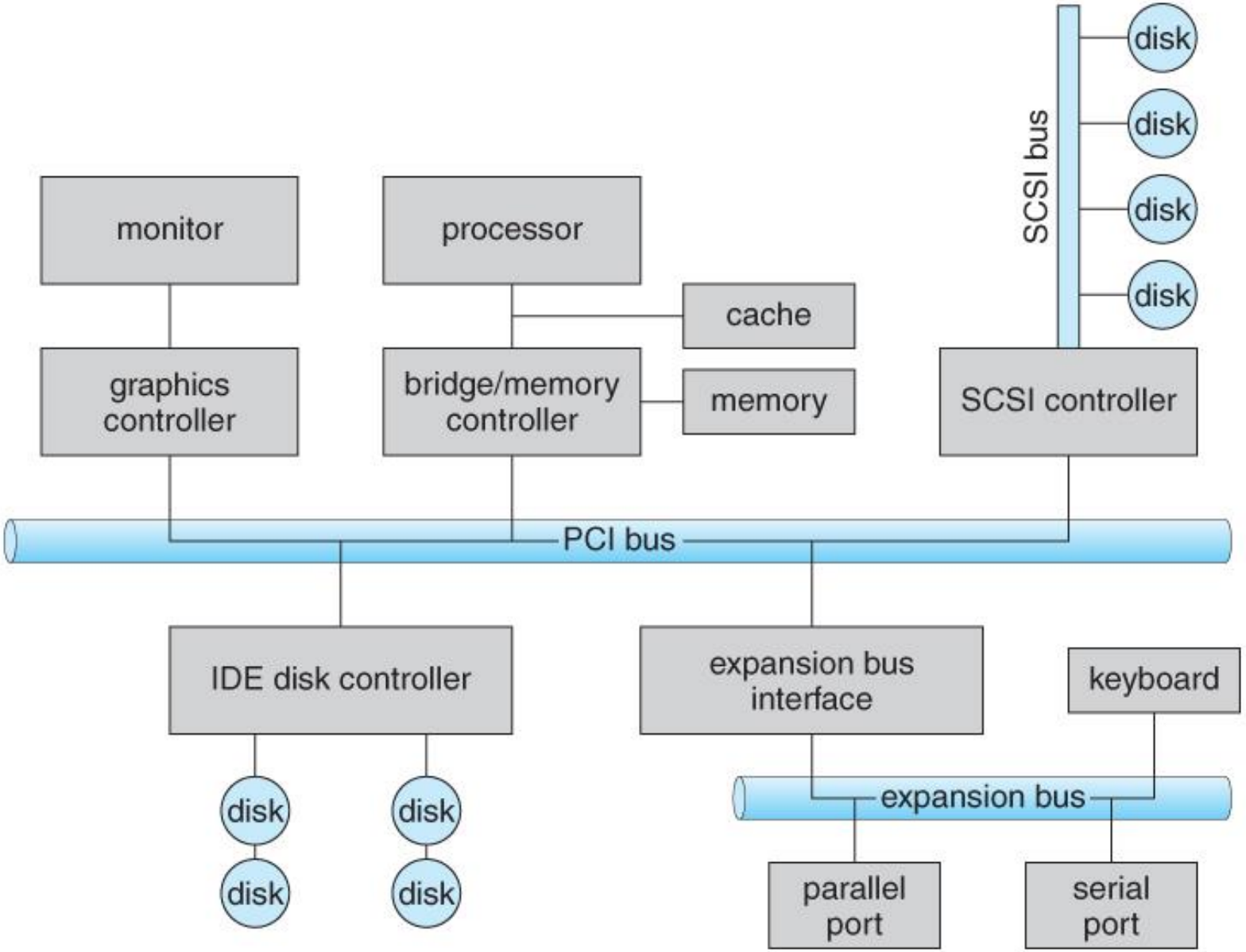


Figure A typical PC bus structure

- 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.
- Figure shows some of the most common I/O port address ranges.

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)

- How can the processor give commands and data to a 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
- 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 from the device.
  - 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.

- Another technique for communicating with devices is ***memory-mapped I/O.***
- In this case a certain portion of the processor's address space is mapped to the device, and communications occur by reading and writing directly to/from those memory areas.

- Memory-mapped I/O is suitable for devices which must move large quantities of data quickly, such as graphics cards.

- Memory-mapped I/O can be used either instead of or more often in combination with traditional registers. For example, graphics cards still use registers for control information such as setting the video mode.

- **Polling I/O**
- Polling is the simplest way for an I/O device to communicate with the processor.

- The process of periodically checking status of the device to see if it is time for the next I/O operation, is called **polling**.

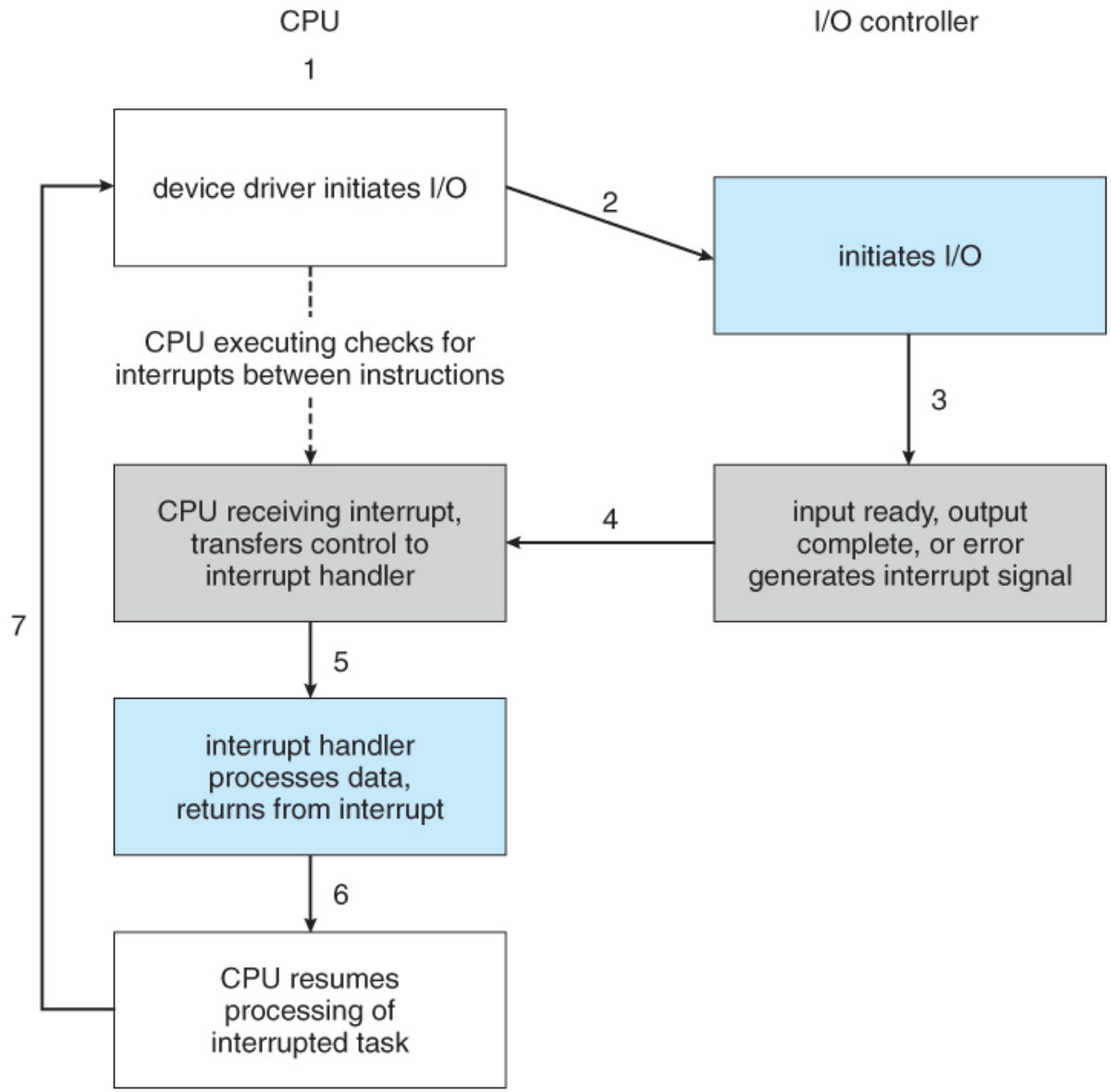
- The I/O device simply puts the information in a Status register, and the processor must come and get the information.

- **I/O Interrupts**
- An alternative scheme for dealing with I/O is the interrupt-driven method.
- An interrupt is a signal to the microprocessor from a device that requires attention.

- A device controller puts an interrupt signal on the bus when it needs CPU's attention when CPU receives an interrupt, It saves its current state and invokes the appropriate interrupt handler using the interrupt vector.

- When the interrupting device has been dealt with, the CPU continues with its original task as if it had never been interrupted.

Interrupt-driven I/O cycle.



- The previous description is adequate for simple interrupt-driven I/O, but there are three needs in modern computing which complicate the picture...

- The need to defer interrupt handling during **critical processing**,

- The need to determine ***which*** interrupt handler to invoke, without having to poll all devices to see which one needs attention,

- The need for multi-level interrupts, so the system can differentiate between high- and low-priority interrupts for proper response.

Maskable and Non-maskable interrupts

- Maskable interrupts:-
- Those which can be disabled by the programmer
- That means, when disabled even if interrupt comes, the cpu simply ignores it and doesn't provide a service to it.

- Nonmaskable interrupts:-
- Is that which can't be disabled and when comes CPU has to give service to it.

Figure shows the Intel Pentium interrupt vector. Interrupts 0 to 31 are non-maskable and reserved for serious hardware and other errors. Maskable interrupts, including normal device I/O interrupts begin at interrupt 32.

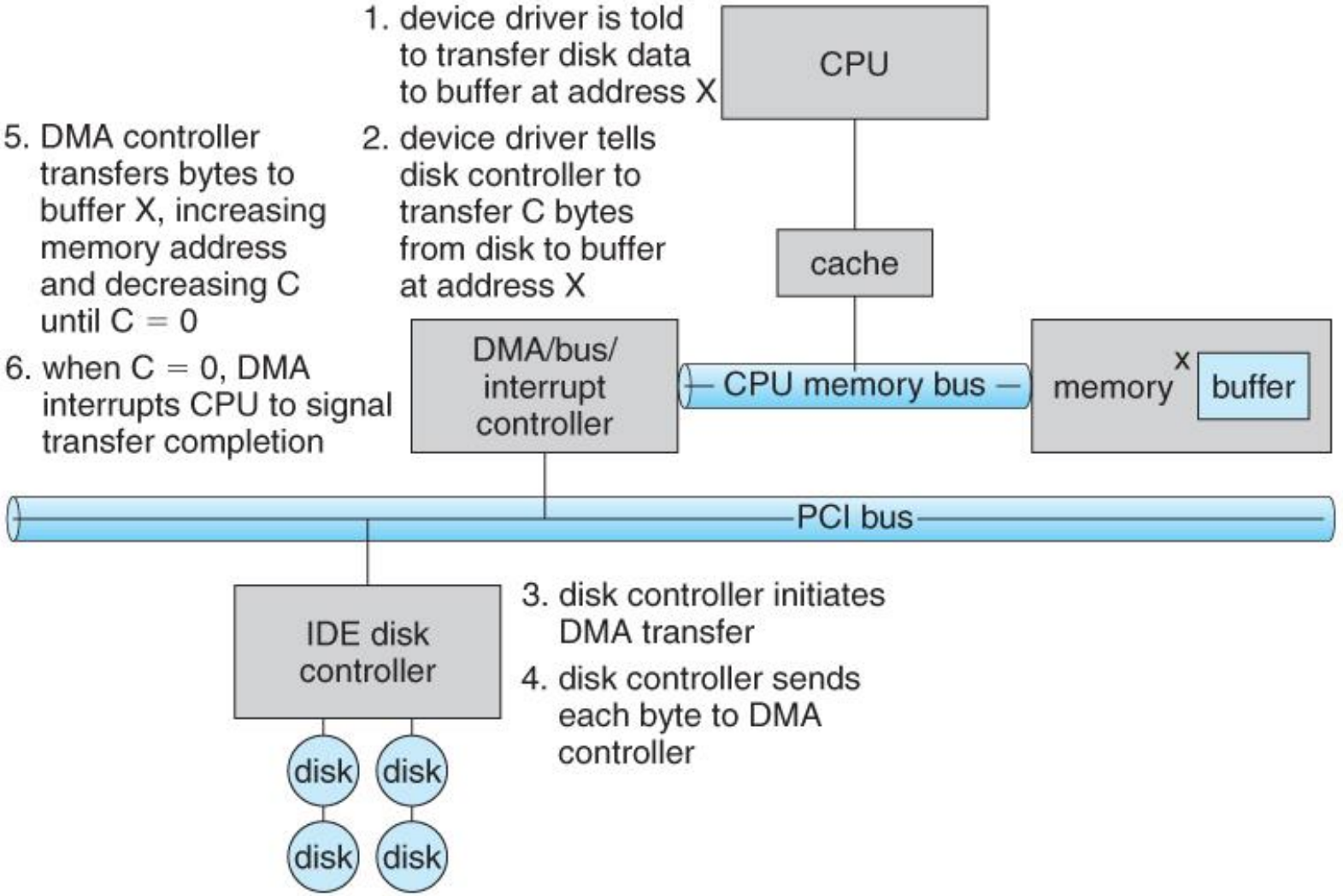
vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault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

At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.

During operation, devices signal errors or the completion of commands via interrupts

Direct Memory Access

- For devices that transfer large quantities of data (such as disk controllers), it is wasteful to tie up the CPU transferring data in and out of registers one byte at a time.
- Instead this work can be off-loaded to a special processor, known as the ***Direct Memory Access, DMA, Controller.***
- The host issues a command to the DMA controller, indicating the location where the data is located, the location where the data is to be transferred to, and the number of bytes of data to transfer. The DMA controller handles the data transfer, and then interrupts the CPU when the transfer is complete.
- A simple DMA controller is a standard component in modern PCs, and many ***bus-mastering*** I/O cards contain their own DMA hardware.
- Handshaking between DMA controllers and their devices is accomplished through two wires called the DMA-request and DMA-acknowledge wires.

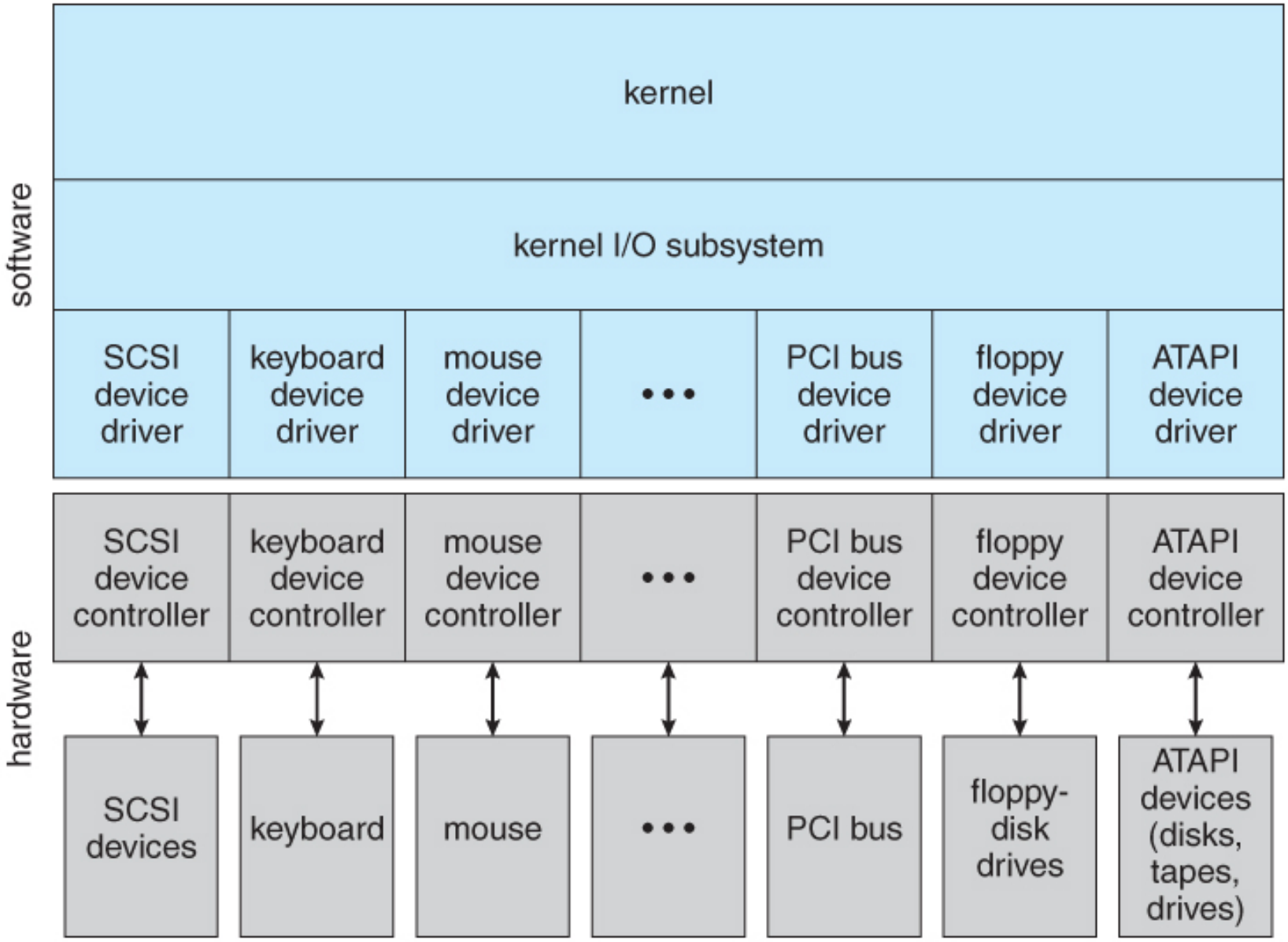


Steps in a DMA transfer.

Application I/O Interface

- User application access to a wide variety of different devices is accomplished through abstraction, software layering, and through encapsulating all of the device-specific code into ***device drivers***, while application layers are presented with a common interface for all (or at least large general categories of) devices.
- Devices differ on many different dimensions.

- 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.



A kernel I/O structure.

- **Character-stream or block**. A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.

- **Sequential or random access**. A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.

- **Synchronous or asynchronous**. A synchronous device performs data transfers with predictable response times, in coordination with other aspects of the system. An asynchronous device exhibits irregular or unpredictable response times not coordinated with other computer events.
- **Sharable or dedicated**. A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.

- **Speed of operation**. Device speeds range from a few bytes per second to a few gigabytes per second.
- **Read–write, read only, or write only**. Some devices perform both input and output, but others support only one data transfer direction.

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.

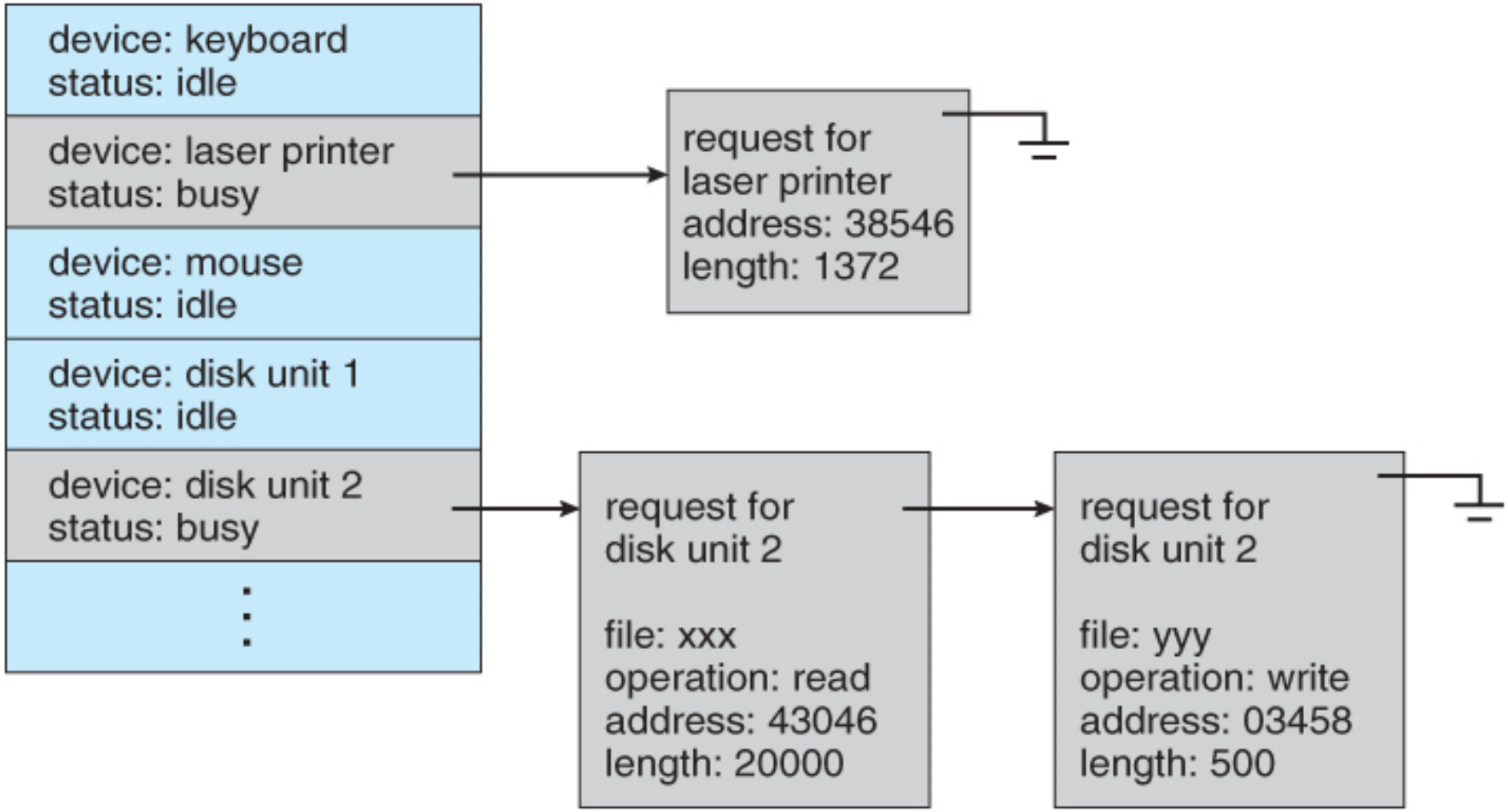
- **Block and Character Devices**
- ***Block devices*** are accessed a block at a time, and are indicated by a "b" as the first character in a long listing on UNIX systems. Operations supported include read(), write(), and seek(). Accessing blocks on a hard drive directly

- ***Character devices*** are accessed one byte at a time, and are indicated by a "c" in UNIX long listings. Supported operations include get( ) and put( ), with more advanced functionality such as reading an entire line supported by higher-level library routines.

- **Network Devices**
- Because network access is inherently different from local disk access, most systems provide a separate interface for network devices.
- One common and popular interface is the ***socket*** interface, which acts like a **cable or pipeline** connecting two networked entities.
- Data can be put into the socket at one end, and read out sequentially at the other end. Sockets are normally full-duplex, allowing for bi-directional data transfer.
- The select( ) system call allows servers (or other applications) to identify sockets which have data waiting, without having to poll all available sockets.

Kernel I/O Subsystem

- Kernels provide many services related to I/O. Several services—scheduling, buffering, caching, spooling, device reservation, and error handling—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
- **I/O Scheduling:-**
- To schedule a set of I/O requests means to determine a good order in which to execute them.
- Scheduling I/O requests can greatly improve overall efficiency. Priorities can also play a part in request scheduling.
- The classic example is the scheduling of disk accesses.
- Buffering and caching can also help, and can allow for more flexible scheduling options.
- On systems with many devices, separate request queues are often kept for each device


device: keyboard
status: idle
device: laser printer
status: busy
device: mouse
status: idle
device: disk unit 1
status: idle
device: disk unit 2
status: busy
request for
laser printer
address: 38546
length: 1372
request for
disk unit 2
file: xxx
operation: read
address: 43046
length: 20000
request for
disk unit 2
file: yyy
operation: write
address: 03458
length: 500

- **Buffering**
- Buffering of I/O is performed for (at least) 3 major reasons
  - **1. Speed differences between two devices.**
  - A slow device may write data into a buffer, and when the buffer is full, the entire buffer is sent to the fast device all at once. So that the slow device still has somewhere to write while this is going on, a second buffer is used, and the two buffers alternate as each becomes full. This is known as ***double buffering.***
  - Double buffering is often used in (animated) graphics, so that one screen image can be generated in a buffer while the other (completed) buffer is displayed on the screen. This prevents the user from ever seeing any half-finished screen images.
  - **2. Data transfer size differences. Buffers are used in particular in networking systems to break messages up into smaller packets for transfer, and then for re-assembly at the receiving side**.

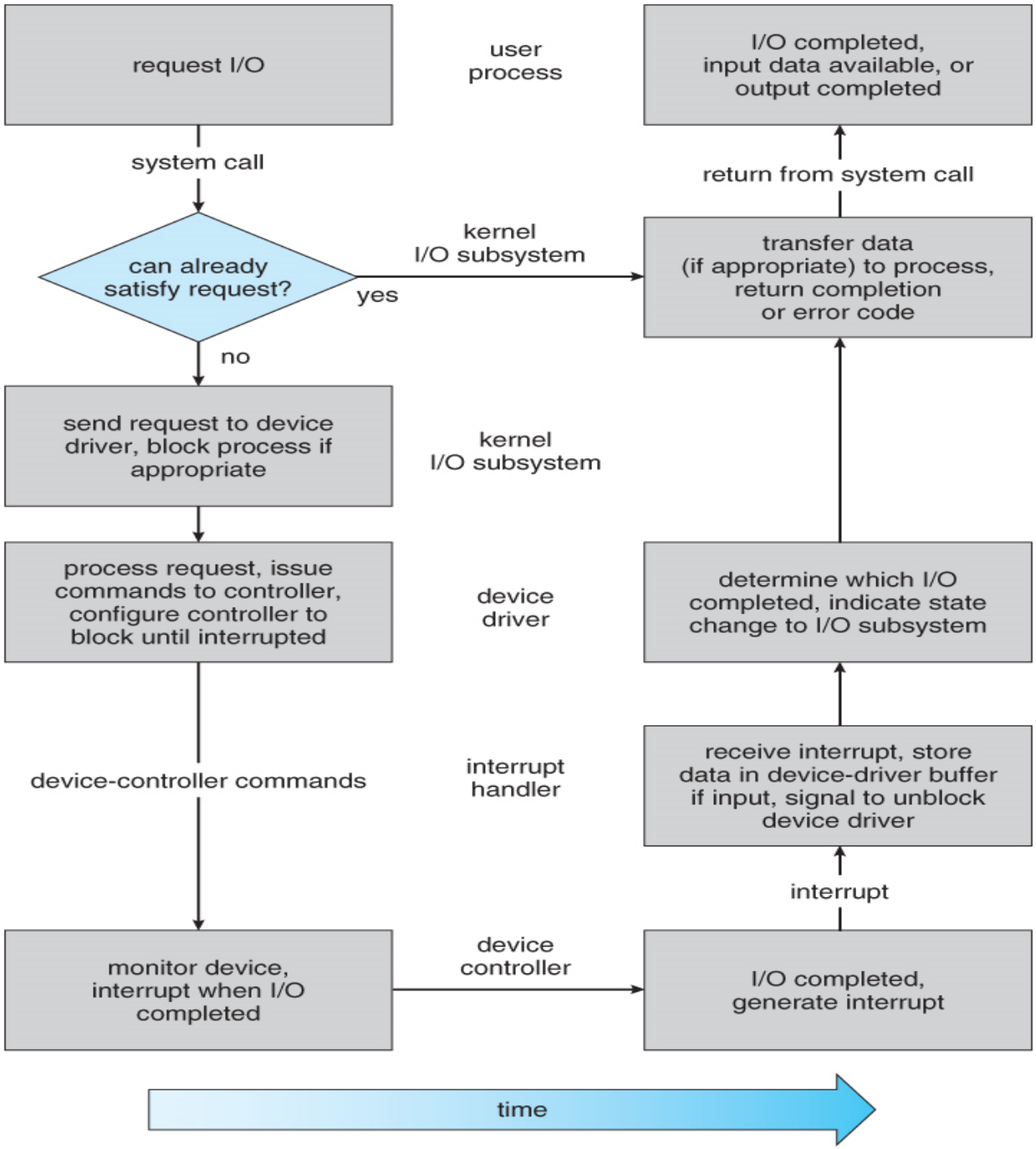
- **3. To support copy semantics.**
- For example, when an application makes a request for a disk **write**, the data is copied from the user's memory area into a kernel buffer. After the system call returns, what happens if the application changes the contents of the buffer?
- With **copy semantics**, 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.

- **Caching:-**
- Caching involves keeping a ***copy*** of data in a faster-access location than where the data is normally stored.
- Buffering and caching are very similar, except that a buffer may hold the only copy of a given data item, whereas a cache is just a duplicate copy of some other data stored elsewhere.

- **Spooling and Device Reservation:--**
- A ***spool (Simultaneous Peripheral Operations On-Line)*** buffers data for (peripheral) devices such as printers that cannot support interleaved data streams.
- If multiple processes want to print at the same time, they each send their print data to files stored in the spool directory. When each file is closed, then the application sees that print job as complete, and the print scheduler sends each file to the appropriate printer one at a time.

Transforming I/O Requests to Hardware Operations

- Consider reading a file from disk for a process:
  - Determine device holding file
  - Translate name to device representation
  - Physically read data from disk into the buffer
  - Make data available to requesting process
  - Return control to process


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
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

STREAMS

- **STREAMS**, is a UNIX System V mechanism, enables an application to assemble pipelines of driver code dynamically. A stream is a full-duplex connection between a device driver and a user-level process.
- It consists of a **stream head** that interfaces with the user process, a **driver end** that controls the device, and zero or more **stream modules** between the stream head and the driver end.
- Each of these components contains a pair of queues —a read queue and a write queue.
- Message passing is used to transfer data between queues.
- The STREAMS structure is shown in Figure.

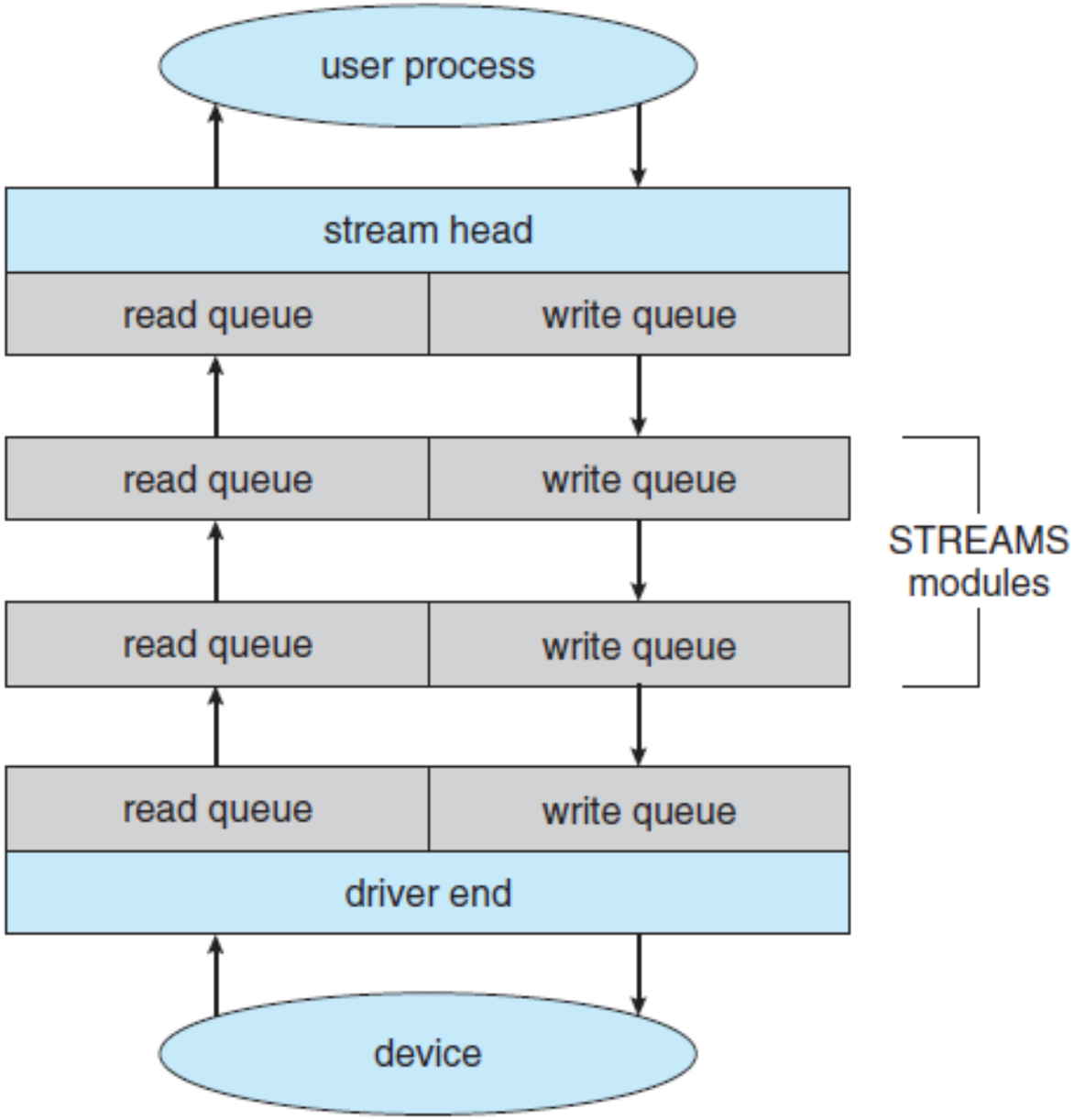


Figure 13.14 The STREAMS structure.

I/O Performance

We can employ several principles to improve the efficiency of I/O:

- Reduce the number of context switches.

- Reduce the number of times that data must be copied in memory while passing between device and application.

- Reduce the frequency of interrupts by using large transfers, smart controllers, and polling (if busy waiting can be minimized).

- Increase concurrency by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.

- Move processing primitives into hardware, to allow their operation in device controllers to be concurrent with CPU and bus operation.

- Balance CPU, memory subsystem, bus, and I/O performance, because an overload in any one area will cause idleness in others.

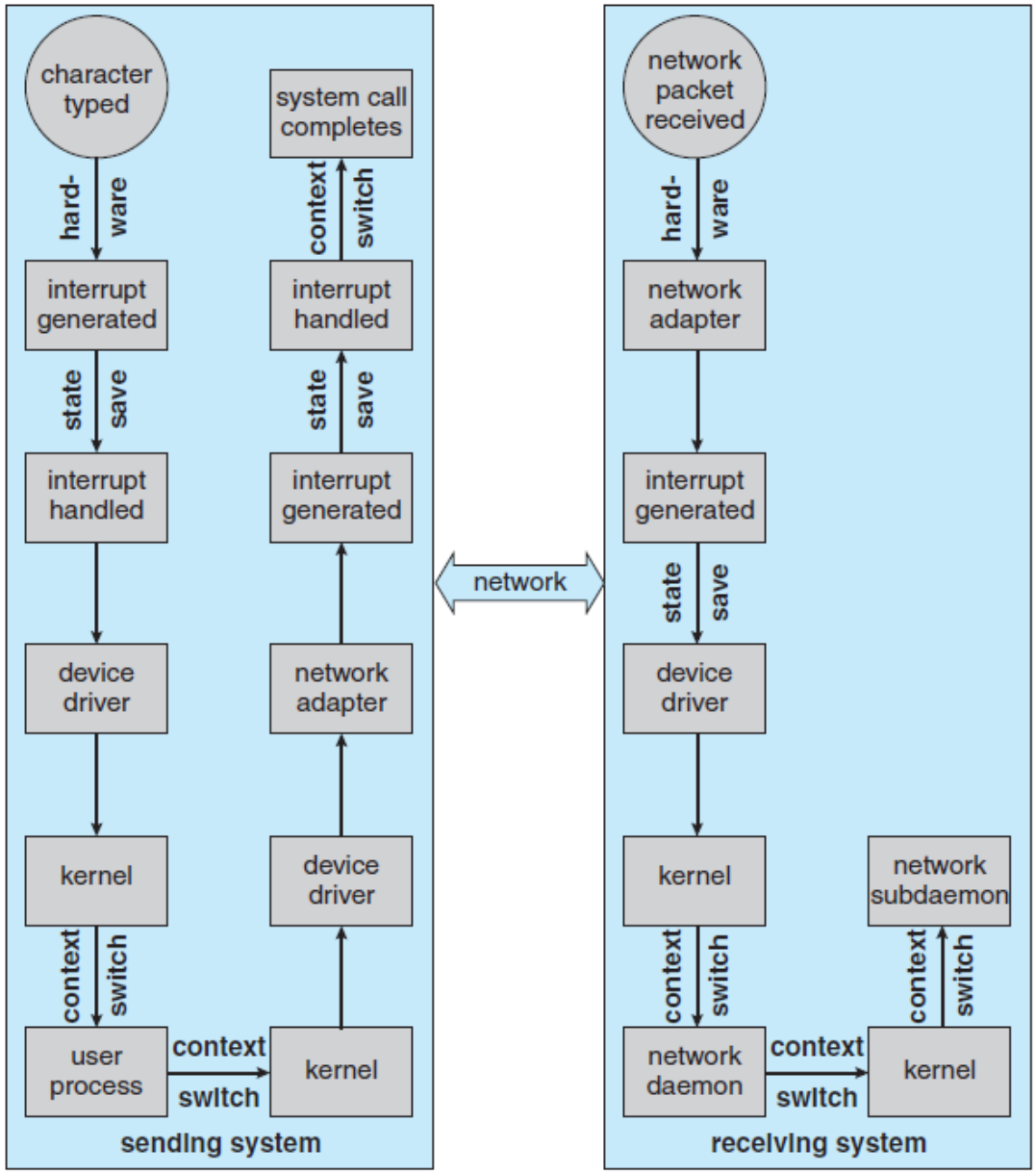


Figure 13.15 Intercomputer communications.

Where should the I/O functionality be implemented—in the device hardware, in the device driver, or in application software? Sometimes we observe the progression depicted in Figure 13.16.

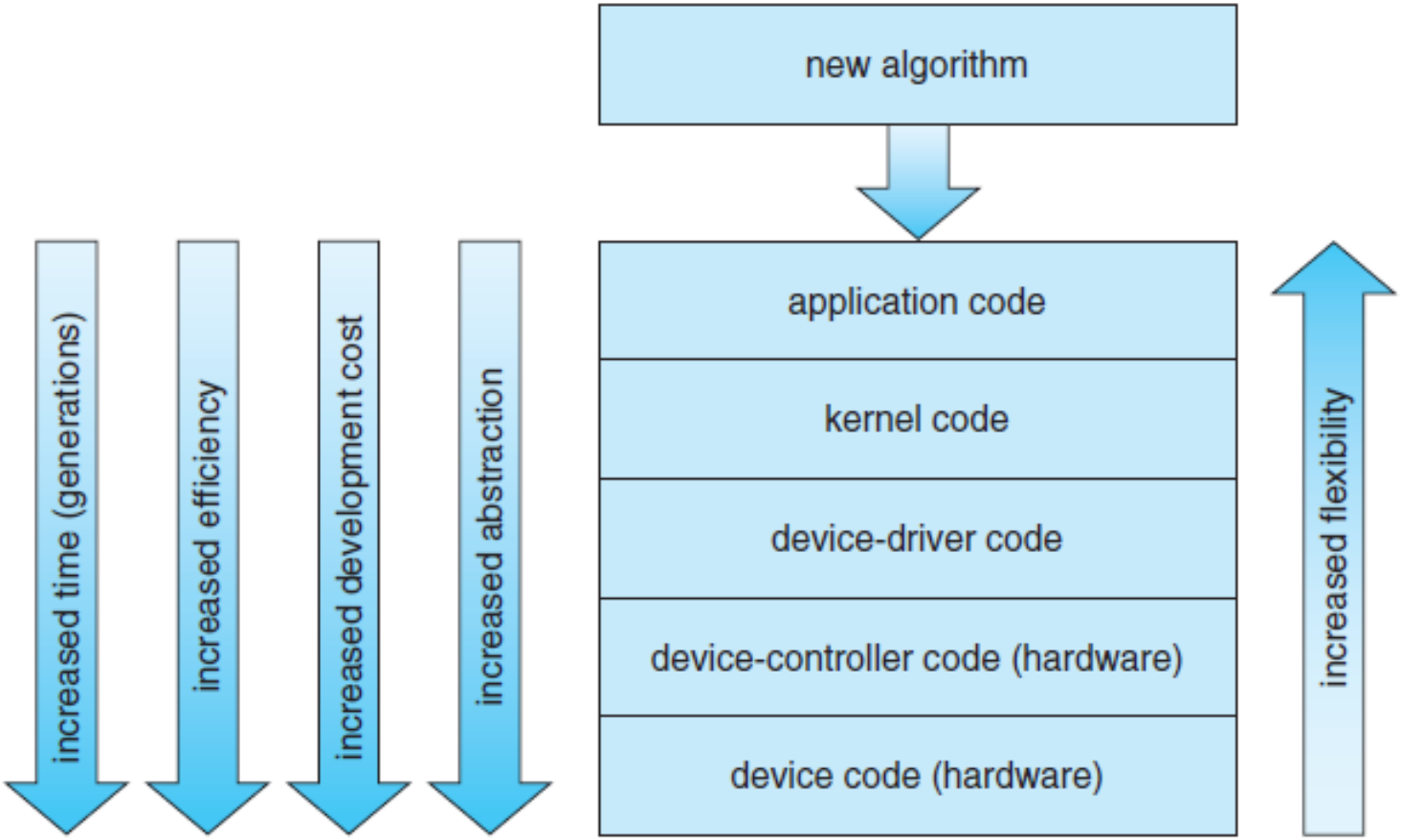


Figure 13.16 Device functionality progression.