1.1 What operating systems do

Introduction

Overview

An *operating system* acts as an intermediary between the user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a *convenient* and *efficient* manner.

An operating system is software that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent programs from interfering with the proper operation of the system.

Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task, and it is important that the goals of the system be well defined before the design begins.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions.

An ***operating system*** is software that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. An amazing aspect of operating systems is how they vary in accomplishing these tasks in a wide variety of computing environments. Operating systems are everywhere, from cars and home appliances that include "Internet of Things" devices, to smart phones, personal computers, enterprise computers, and cloud computing environments.

In order to explore the role of an operating system in a modern computing environment, it is important first to understand the organization and architecture of computer hardware. This includes the CPU, memory, and I/O devices, as well as storage. A fundamental responsibility of an operating system is to allocate these resources to programs.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions. In this chapter, we provide a general overview of the major components of a contemporary computer system as well as the functions provided by the operating system. Additionally, we cover several topics to help set the stage for the remainder of the text: data structures used in operating systems, computing environments, and open-source and free operating systems.

Chapter objectives

- Describe the general organization of a computer system and the role of interrupts.
- Describe the components in a modern multiprocessor computer system.
- Illustrate the transition from user mode to kernel mode.
- Discuss how operating systems are used in various computing environments.
- Provide examples of free and open-source operating systems.

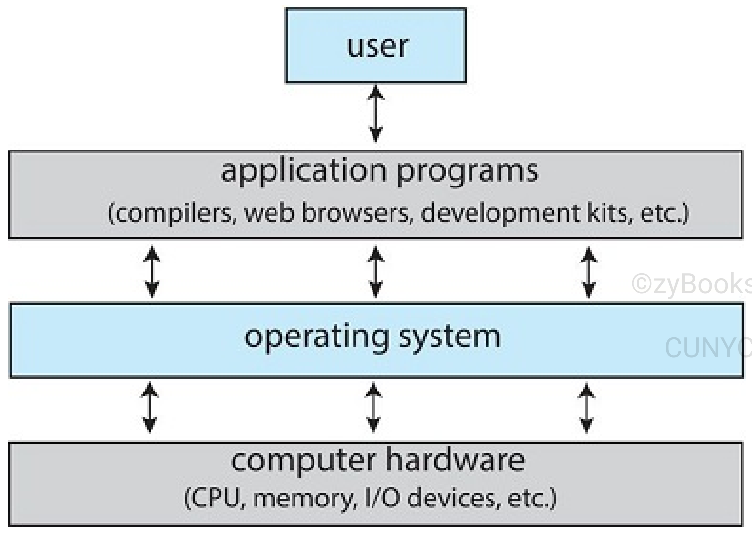
Section glossary

operating system: A program that manages a computer's hardware, provides a basis for application programs, and acts as an intermediary between the computer user and the computer hardware.

What operating systems do

We begin our discussion by looking at the operating system's role in the overall computer system. A computer system can be divided roughly into four components: the ***hardware,*** the ***operating system,*** the ***application programs,*** and a ***user*** (Figure 1.1.1).

Figure 1.1.1: Abstract view of the components of a computer system.


user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

The ***hardware***—the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system. The ***application programs***—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

We can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation of the computer system. An operating system is similar to a government. Like a government, it performs no useful function by itself. It simply provides an ***environment*** within which other programs can do useful work.

To understand more fully the operating system's role, we next explore operating systems from two viewpoints: that of the user and that of the system.

User view

The user's view of the computer varies according to the interface being used. Many computer users sit with a laptop or in front of a PC consisting of a monitor, keyboard, and mouse. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for ***ease of use***, with some attention paid to performance and security and none paid to ***resource utilization***—how various hardware and software resources are shared.

Increasingly, many users interact with mobile devices such as smartphones and tablets—devices that are replacing desktop and laptop computer systems for some users. These devices are typically connected to networks through cellular or other wireless technologies. The user interface for mobile computers generally features a ***touch screen***, where the user interacts with the system by pressing and swiping fingers across the screen rather than using a physical keyboard and mouse. Many mobile devices also allow users to interact through a ***voice recognition*** interface, such as Apple's ***Siri***.

Some computers have little or no user view. For example, ***embedded computers*** in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems and applications are designed primarily to run without user intervention.

System view

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a ***resource allocator***. A computer system has many resources that may be required to solve a problem: CPU time, memory space, storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly.

A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a control program. A ***control program*** manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

Defining operating systems

By now, you can probably see that the term ***operating system*** covers many roles and functions. That is the case, at least in part, because of the myriad designs and uses of computers. Computers are present within toasters, cars, ships, spacecraft, homes, and businesses. They are the basis for game machines, cable TV tuners, and industrial control systems.

To explain this diversity, we can turn to the history of computers. Although computers have a relatively short history, they have evolved rapidly. Computing started as an experiment to determine what could be done and quickly moved to fixed-purpose systems for military uses, such as code breaking and trajectory plotting, and governmental uses, such as census calculation. Those early computers evolved into general-purpose, multifunction mainframes, and that's when operating systems were born. In the 1960s, ***Moore's Law*** predicted that the number of transistors on an integrated circuit would double every 18 months, and that prediction has held true. Computers gained in functionality and shrank in size, leading to a vast number of uses and a vast number and variety of operating systems. (See Appendix A for more details on the history of operating systems.)

How, then, can we define what an operating system is? In general, we have no completely adequate definition of an operating system. Operating systems exist because they offer a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute programs and to make solving user problems easier. Computer hardware is constructed toward this goal. Since bare hardware alone is not particularly easy to use, application programs are developed. These programs require certain common operations, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

In addition, we have no universally accepted definition of what is part of the operating system. A simple viewpoint is that it includes everything a vendor ships when you order "the operating system." The features included, however, vary greatly across systems. Some systems take up less than a megabyte of space and lack even a full-screen editor, whereas others require gigabytes of space and are based entirely on graphical windowing systems. A more common definition, and the one that we usually follow, is that the operating system is the one program running at all times on the computer—usually called the ***kernel***. Along with the kernel, there are two other types of programs: ***system programs***, which are associated with the operating system but are not necessarily part of the kernel, and application programs, which include all programs not associated with the operation of the system.

There are also awkward bits of code that defy categorization. Device drivers for example manage hardware devices, run within the kernel, but are frequently provided by the hardware manufacturer. They are arguably part of the operating system or possibly thought of as being a layer between the hardware and the operating system. Another confusion comes from loadable kernel modules. These chunks of compiled code sometimes come with the operating system (for example, a non-default file system) and are automatically loaded by the system as needed. Sometimes the system administrator adds and can manage these modules (for example, adding a file system that doesn't ship with the operating system). So here we have a component running within the operating system that didn't ship with it.

Further complicating the definition of an operating system is the difference between common understanding and reality. Most people think of their computer as running "an operating system". But if we define an operating system as constantly running, and managing hardware, then we need to consider the **many** operating systems running on a common computer. The CPU has firmware - software embedded in the hardware - the motherboard has BIOS code that boots and manages it, and devices have firmware managing their functionality. In reality, there might be more code running in these other operating systems than in the main general-purpose operating system. We discuss some of these aspects further in this book but mostly constrain coverage to the general-purpose operating system.

So, where does that leave us in our quest to define a general-purpose computer's "operating system"? A reasonable summary is that the operating system is inclusive of the kernel that is loaded at boot time, any device drivers and kernel functions loaded at run time, and any system programs related to the operation of the system (as opposed to applications). The operating system is the piece of software that sits between the hardware and applications, providing a set of services to the computer and users, and is not part of the firmware or hardware logic.

The matter of what constitutes an operating system became increasingly important as personal computers became more widespread and operating systems grew increasingly sophisticated. In 1998, the United States Department of Justice filed suit against Microsoft, in essence claiming that Microsoft included too much functionality in its operating systems and thus prevented application vendors from competing. (For example, a web browser was an integral part of Microsoft's operating systems.) As a result, Microsoft was found guilty of using its operating-system monopoly to limit competition.

Today, however, if we look at operating systems for mobile devices, we see that once again the number of features constituting the operating system is increasing. Mobile operating systems often include not only a core kernel but also ***middleware***—a set of software frameworks that provide additional services to application developers. For example, each of the two most prominent mobile operating systems—Apple's IOS and Google's Android—features a core kernel along with middleware that supports databases, multimedia, and graphics (to name only a few).

Why study operating systems?

Although there are many practitioners of computer science, only a small percentage of them will be involved in the creation or modification of an operating system. Why, then, study operating systems and how they work? Simply because, as almost all code runs on top of an operating system, knowledge of how operating systems work is crucial to proper, efficient, effective, and secure programming. Understanding the fundamentals of operating systems, how they drive computer hardware, and what they provide to applications is not only essential to those who program them but also highly useful to those who write programs on them and use them.

In summary, for our purposes, the operating system includes the always-running kernel, middleware frameworks that ease application development and provide features, and system programs that aid in managing the system while it is running. Most of this text is concerned with the kernel of general-purpose operating systems, but other components are discussed as needed to fully explain operating system design and operation.

PARTICIPATION ACTIVITY 1.1.1: Section review questions.

1) ___ is a software framework that provides additional services (to those provided by an operating system) to application developers.

- ○ System programs
- ○ Kernel
- ○ Middleware

Section glossary

hardware: The CPU, memory devices, input/output (I/O) devices, and any other physical components that are part of a computer.

application program: A program designed for end-user execution, such as a word processor, spreadsheet, compiler, or Web browser.

ease of use: The amount of difficulty and complexity involved in using some aspect of computing.

resource utilization: The amount of a given resource (hardware or software) that is being used.

touch screen: A touch-sensitive screen used as a computer input device.

voice recognition: A computer interface based on spoken commands, which the computer parses and turns into actions.

Siri: The Apple voice-recognition system.

embedded computer: A computer system within some other, larger system (such as a car) that performs specific, limited functions and has little or no user interface.

resource allocator: An operating system or application that determines how resources are to be used.

control program: A program that manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

Moore's Law: A law predicting that the number of transistors on an integrated circuit would double every eighteen months.

kernel: The operating system component running on the computer at all times after system boot.

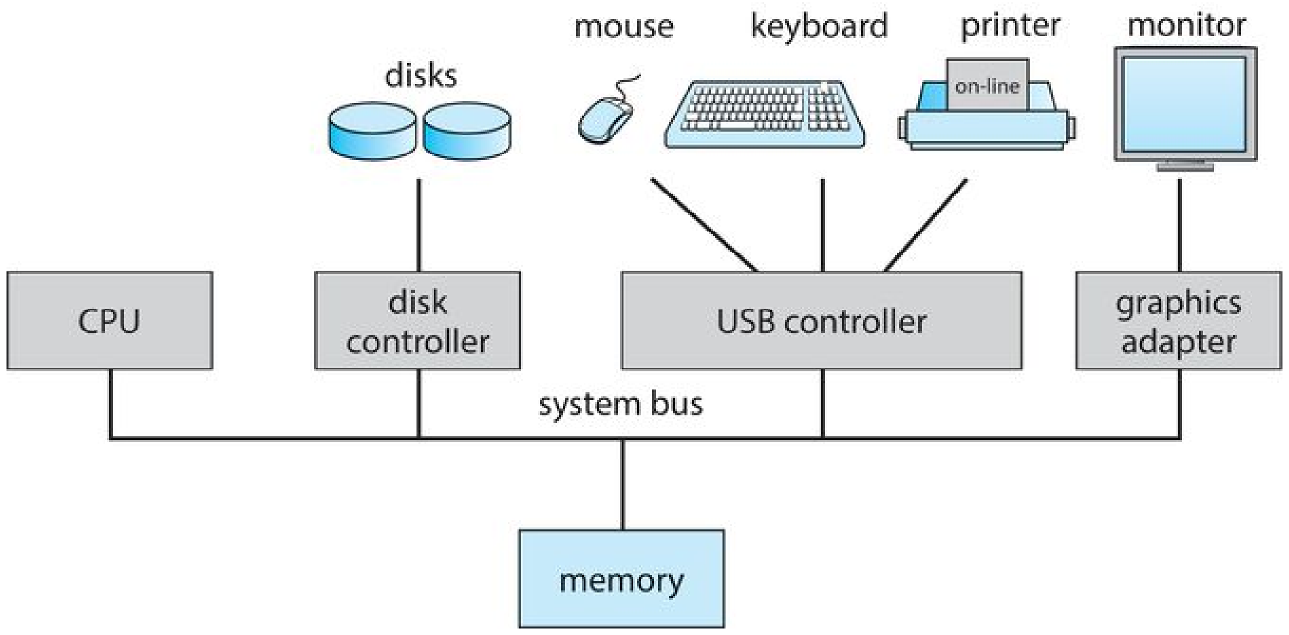
system program: A program associated with the operating system but not necessarily part of the kernel.

middleware: A set of software frameworks that provide additional services to application developers.

1.2 Computer-system organization

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common ***bus*** that provides access between components and shared memory (Figure 1.2.1). Each device controller is in charge of a specific type of device (for example, a disk drive, audio device, or graphics display). Depending on the controller, more than one device may be attached. For instance, one system USB port can connect to a USB hub, to which several devices can connect. A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.

Figure 1.2.1: A typical PC computer system.



Typically, operating systems have a ***device driver*** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device. The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

In the following subsections, we describe some basics of how such a system operates, focusing on three key aspects of the system. We start with interrupts, which alert the CPU to events that require attention. We then discuss storage structure and I/O structure.

Interrupts

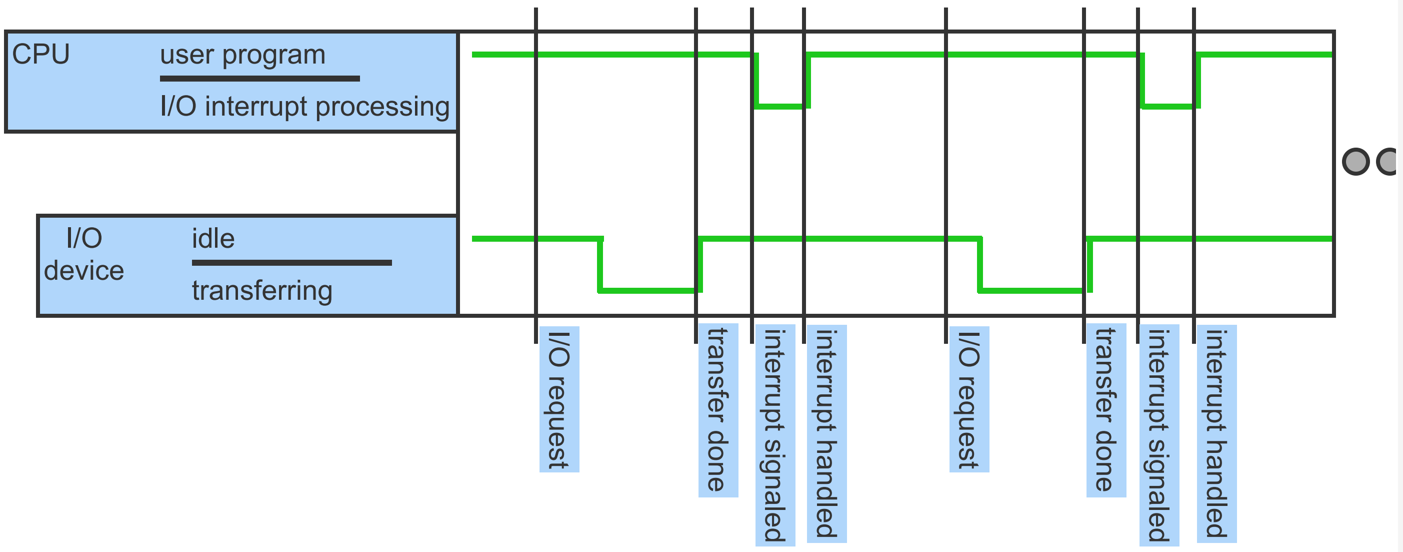
Consider a typical computer operation: a program performing I/O. To start an I/O operation, the device driver loads the appropriate registers in the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take (such as "read a character from the keyboard"). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver that it has finished its operation. The device driver then gives control to other parts of the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information such as "write completed successfully" or "device busy". But how does the controller inform the device driver that it has finished its operation? This is accomplished via an ***interrupt***.

Overview

Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. (There may be many buses within a computer system, but the system bus is the main communications path between the major components.) Interrupts are used for many other purposes as well and are a key part of how operating systems and hardware interact.

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation. A timeline of this operation is shown in the animation below.

PARTICIPATION ACTIVITY 1.2.1: Interrupt Timeline for a Single Process Doing Output.



Animation content:

Step 1: The user program runs. The I/O device receives an I/O request. Step 2: The I/O device transfers data. The I/O device then finishes the transfer and signals an interrupt. The CPU receives the interrupt and halts the execution of the user program. Eventually, the execution resumes. Step 3: The I/O device receives another I/O request. Step 4: The I/O device transfers data and signals another interrupt. The CPU receives the interrupt and handles the interrupt. Once the I/O interrupt is handled, the CPU resumes execution of the user program.

Animation captions:

1. The user program runs. The I/O device receives an I/O request.
2. The I/O device transfers data. The I/O device then finishes the transfer and signals an interrupt. The CPU receives the interrupt and halts the execution of the user program. Eventually, the execution resumes.
3. The I/O device receives another I/O request.
4. The I/O device transfers data and signals another interrupt. The CPU receives the interrupt and handles the interrupt. Once the I/O interrupt is handled, the CPU resumes execution of the user program.

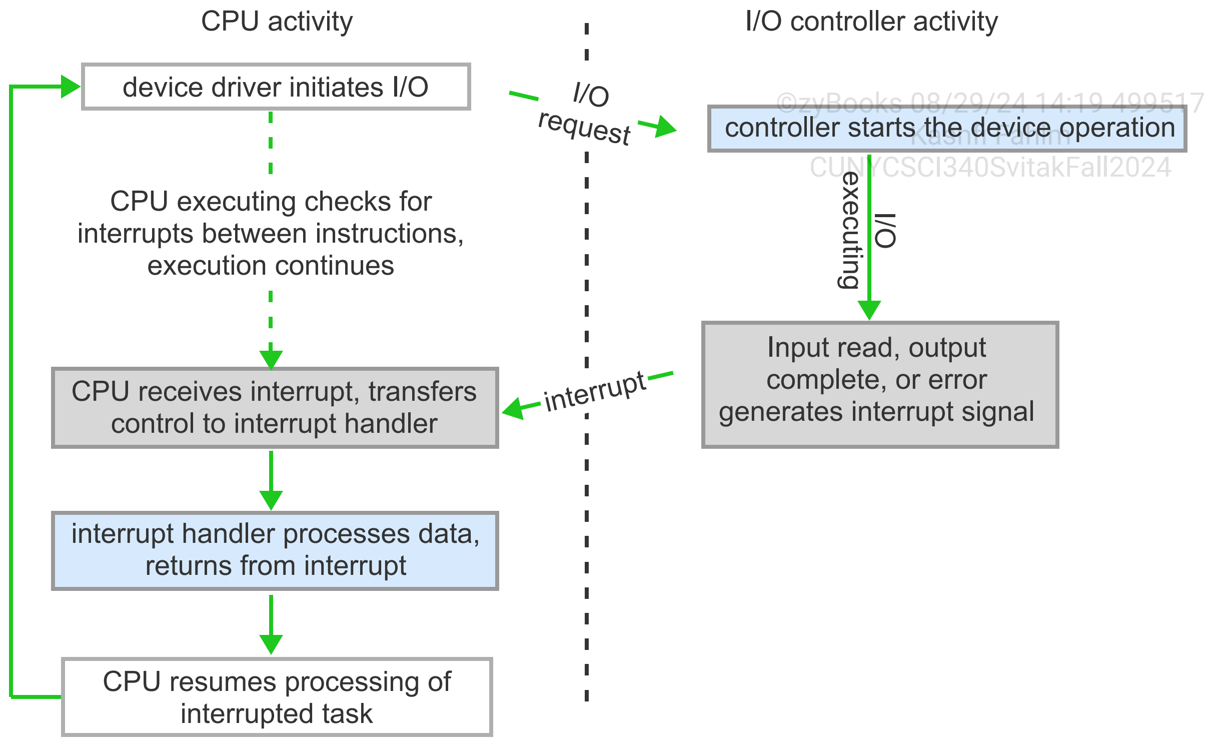
Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for managing this transfer would be to invoke a generic routine to examine the interrupt information. The routine, in turn, would call the interrupt-specific handler. However, interrupts must be handled quickly, as they occur very frequently. A table of pointers to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or ***interrupt vector***, of addresses is then indexed by a unique number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. Operating systems as different as Windows and UNIX dispatch interrupts in this manner.

The interrupt architecture must also save the state information of whatever was interrupted, so that it can restore this information after servicing the interrupt. If the interrupt routine needs to modify the processor state—for instance, by modifying register values—it must explicitly save the current state and then restore that state before returning. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

Implementation

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, it reads the interrupt number and jumps to the ***interrupt-handler routine*** by using that interrupt number as an index into the interrupt vector. It then starts execution at the address associated with that index. The interrupt handler saves any state it will be changing during its operation, 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 animation below summarizes the interrupt-driven I/O cycle.

PARTICIPATION ACTIVITY | 1.2.2: Interrupt-driven I/O Cycle.


CPU activity
I/O controller activity
device driver initiates I/O
I/O request
controller starts the device operation
I/O executing
CPU executing checks for interrupts between instructions, execution continues
Input read, output complete, or error generates interrupt signal
interrupt
CPU receives interrupt, transfers control to interrupt handler
interrupt handler processes data, returns from interrupt
CPU resumes processing of interrupted task

Animation content:

Step One: Device driver senses device I/O activity, initiates I/O. Step 2: Driver tells I/O controller to initiate I/O. Step 3: While I/O executes, the CPU checks for interrupts between instructions. Step 4: The device completes its operation, generates an interrupt which is received by the CPU. Step 5. CPU jumps to interrupt handler which processes the interrupt, returns from interrupt handler. Step 6: CPU resumes task that was interrupted. Step 7: Ready for the next interrupt to start the cycle again.

Animation captions:

1. Device driver senses device I/O activity, initiates I/O.
2. Driver tells I/O controller to initiate I/O.
3. While I/O executes, the CPU checks for interrupts between instructions.
4. The device completes its operation, generates an interrupt which is received by the CPU.
5. CPU jumps to interrupt handler which processes the interrupt, returns from interrupt handler.
6. CPU resumes task that was interrupted.
7. Ready for the next interrupt to start the cycle again.

The basic interrupt mechanism just described enables the CPU to respond to an asynchronous event, as when a device controller becomes ready for service. 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.
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.

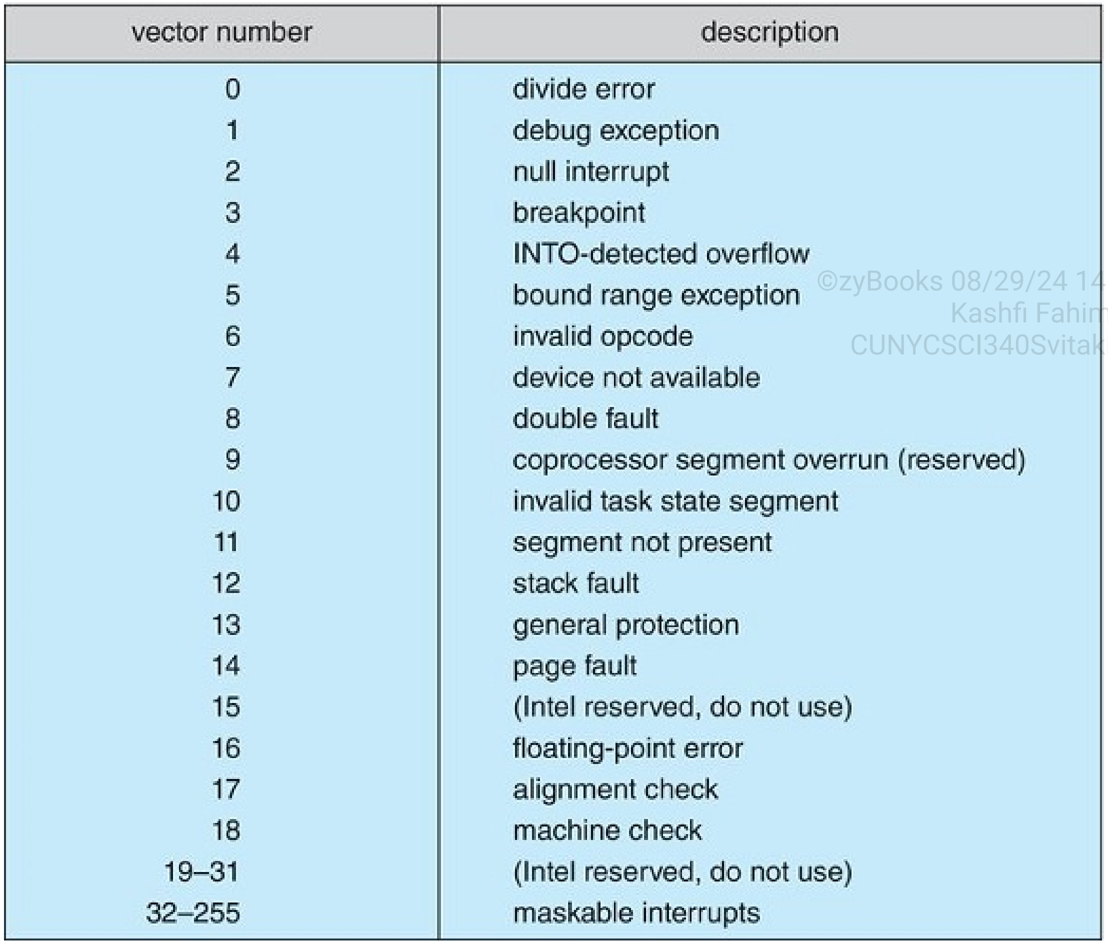
In modern computer hardware, these three features are provided by the CPU and the ***interrupt-controller hardware***.

Most CPUs have two interrupt request lines. One is the ***nonmaskable interrupt***, which is reserved for events such as unrecoverable memory errors. The second interrupt line is ***maskable***: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.

Recall that 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. 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***, in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.

Figure 1.2.2 illustrates the design of the interrupt vector for Intel processors. 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.

Figure 1.2.2: Intel 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

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.

In summary, interrupts are used throughout modern operating systems to handle asynchronous events (and for other purposes we will discuss throughout the text). Device controllers and hardware faults raise interrupts. To enable the most urgent work to be done first, modern computers use a system of interrupt priorities. Because interrupts are used so heavily for time-sensitive processing, efficient interrupt handling is required for good system performance.

Storage structure

The CPU can load instructions only from memory, so any programs must first be loaded into memory to run. General-purpose computers run most of their programs from rewritable memory, called main memory (also called ***random-access memory***, or *RAM*). Main memory commonly is implemented in a semiconductor technology called ***dynamic random-access memory*** (*DRAM*).

Computers use other forms of memory as well. For example, the first program to run on computer power-on is a ***bootstrap program***, which then loads the operating system. Since RAM is ***volatile***—loses its content when power is turned off or otherwise lost—we cannot trust it to hold the bootstrap program. Instead, for this and some other purposes, the computer uses electrically erasable programmable read-only memory (EEPROM) and other forms of ***firmware***—storage that is infrequently written to and is nonvolatile. EEPROM can be changed but cannot be changed frequently. In addition, it is low speed, and so it contains mostly static programs and data that aren't frequently used. For example, the iPhone uses EEPROM to store serial numbers and hardware information about the device.

Storage definitions and notation

The basic unit of computer storage is the ***bit***. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A ***byte*** is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is ***word***, which is a given computer architecture's native unit of data. A word is made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words. A computer executes many operations in its native word size rather than a byte at a time.

Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes. A ***kilobyte***, or *KB*, is 1,024 bytes; a megabyte, or *MB*, is $1,024^2$ bytes; a ***gigabyte***, or *GB*, is $1,024^3$ bytes; a ***terabyte***, or *TB*, is $1,024^4$ bytes; and a ***petabyte***, or *PB*, is $1,024^5$ bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of `load` or `store` instructions to specific memory addresses. The `load` instruction moves a byte or word from main memory to an internal register within the CPU, whereas the `store` instruction moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution from the location stored in the program counter.

A typical instruction-execution cycle, as executed on a system with a ***von Neumann architecture***, first fetches an instruction from memory and stores that instruction in the ***instruction register***. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory. Notice that the memory unit sees only a stream of memory addresses. It does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, or some other means) or what they are for (instructions or data). Accordingly, we can ignore ***how*** a memory address is generated by a program. We are interested only in the sequence of memory addresses generated by the running program.

Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible on most systems for two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.
2. Main memory, as mentioned, is volatile—it loses its contents when power is turned off or otherwise lost.

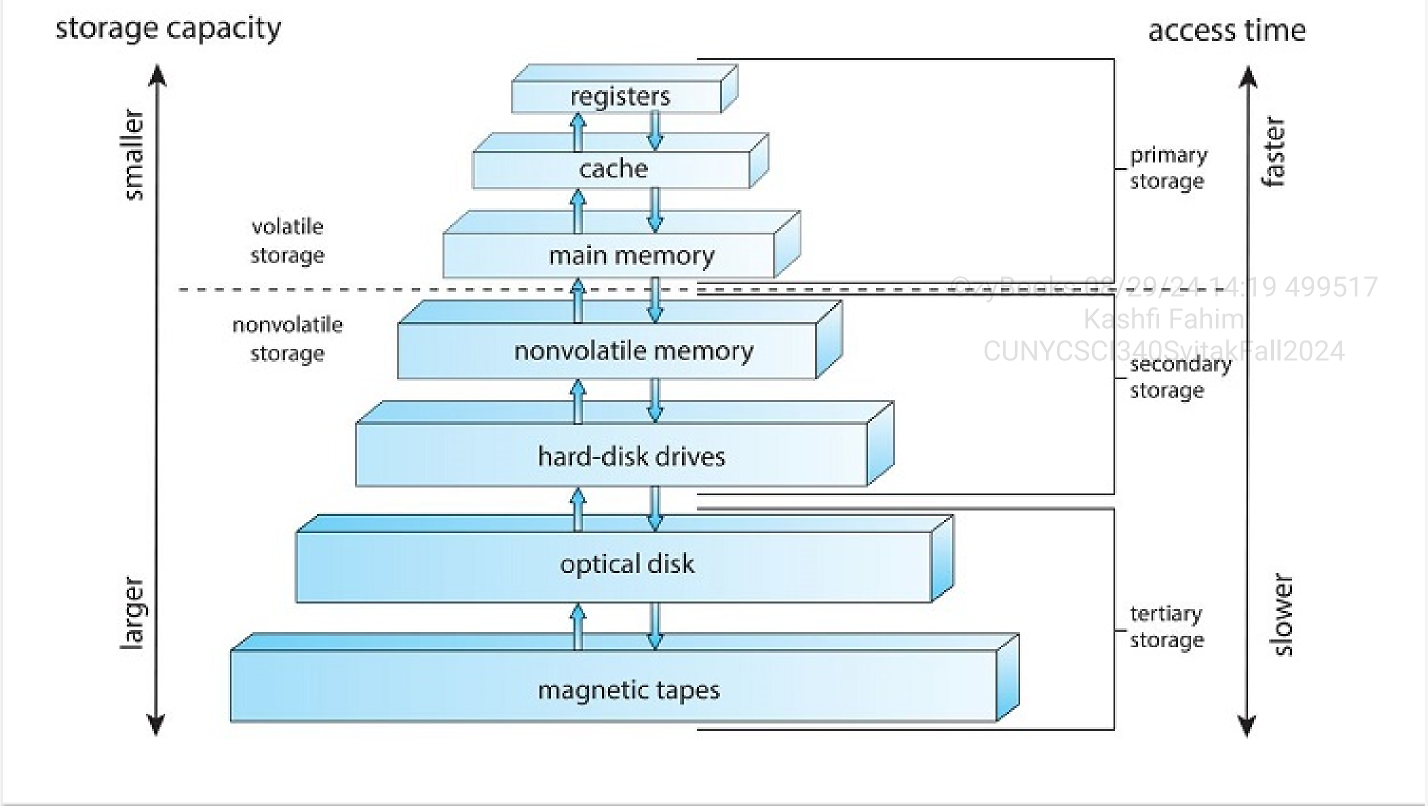
Thus, most computer systems provide ***secondary storage*** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.

The most common secondary-storage devices are ***hard-disk drives*** (*HDDs*) and ***nonvolatile memory*** (*NVM*) ***devices***, which provide storage for both programs and data. Most programs (system and application) are stored in secondary storage until they are loaded into memory. Many programs then use secondary storage as both the source and the destination of their processing. Secondary storage is also much slower than main memory. Hence, the proper management of secondary storage is of central importance to a computer system, as we discuss in the chapter Mass-Storage Structure.

In a larger sense, however, the storage structure that we have described—consisting of registers, main memory, and secondary storage—is only one of many possible storage system designs. Other possible components include cache memory, CD-ROM or blu-ray, magnetic tapes, and so on. Those that are slow enough and large enough that they are used only for special purposes—to store backup copies of material stored on other devices, for example—are called ***tertiary storage***. Each storage system provides the basic functions of storing a datum and holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, size, and volatility.

The wide variety of storage systems can be organized in a hierarchy (Figure 1.2.3) according to storage capacity and access time. As a general rule, there is a trade-off between size and speed, with smaller and faster memory closer to the CPU. As shown in the figure, in addition to differing in speed and capacity, the various storage systems are either volatile or nonvolatile. Volatile storage, as mentioned earlier, loses its contents when the power to the device is removed, so data must be written to nonvolatile storage for safekeeping.

Figure 1.2.3: Storage-device hierarchy.



The top four levels of memory in the figure are constructed using ***semiconductor memory***, which consists of semiconductor-based electronic circuits. NVM devices, at the fourth level, have several variants but in general are faster than hard disks. The most common form of NVM device is flash memory, which is popular in mobile devices such as smartphones and tablets. Increasingly, flash memory is being used for long-term storage on laptops, desktops, and servers as well.

Since storage plays an important role in operating-system structure, we will refer to it frequently in the text. In general, we will use the following terminology:

- Volatile storage will be referred to simply as ***memory***. If we need to emphasize a particular type of storage device (for example, a register), we will do so explicitly.
- Nonvolatile storage retains its contents when power is lost. It will be referred to as ***NVS***. The vast majority of the time we spend on NVS will be on secondary storage. This type of storage can be classified into two distinct types:
  - ***Mechanical.*** A few examples of such storage systems are HDDs, optical disks, holographic storage, and magnetic tape. If we need to emphasize a particular type of mechanical storage device (for example, magnetic tape), we will do so explicitly.
  - ***Electrical.*** A few examples of such storage systems are flash memory, FRAM, NRAM, and SSD. Electrical storage will be referred to as ***NVM***. If we need to emphasize a particular type of electrical storage device (for example, SSD), we will do so explicitly.

  Mechanical storage is generally larger and less expensive per byte than electrical storage. Conversely, electrical storage is typically costly, smaller, and faster than mechanical storage.

The design of a complete storage system must balance all the factors just discussed: it must use only as much expensive memory as necessary while providing as much inexpensive, nonvolatile storage as possible. Caches can be installed to improve performance where a large disparity in access time or transfer rate exists between two components.

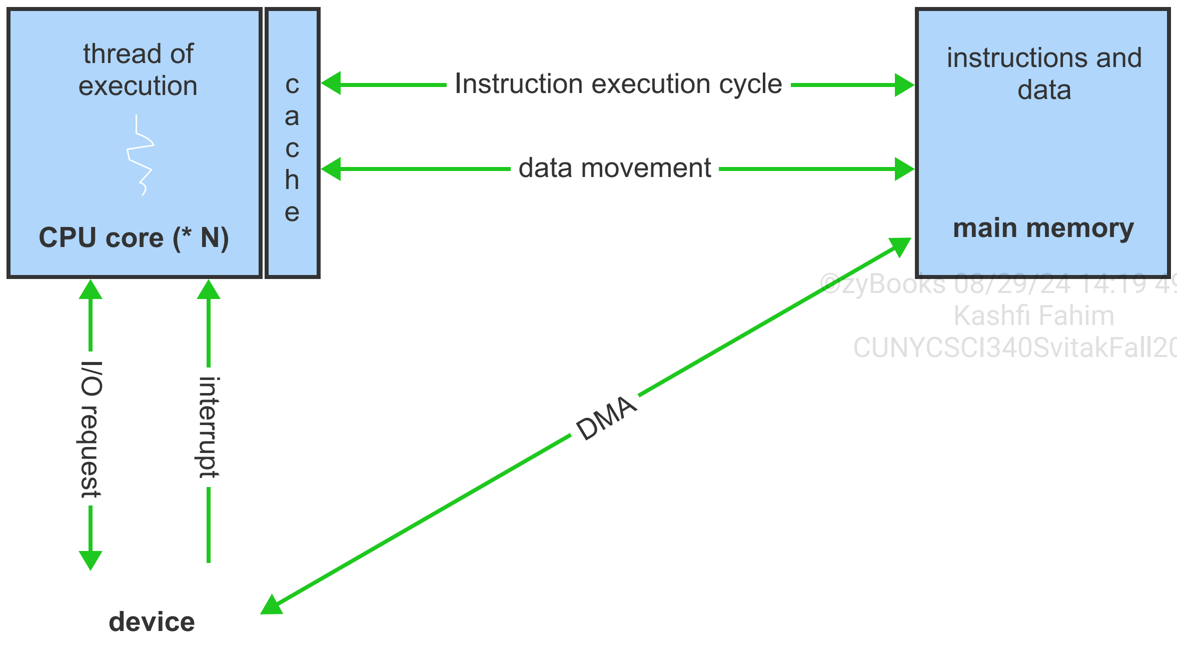
I/O structure

A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.

Recall from the beginning of this section that a general-purpose computer system consists of multiple devices, all of which exchange data via a common bus. The form of interrupt-driven I/O described in Section Interrupts is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as NVS I/O. To solve this problem, ***direct memory access*** (*DMA*) is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from the device and main memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. While the device controller is performing these operations, the CPU is available to accomplish other work.

Some high-end systems use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. In this case, DMA is even more effective. The animation below shows the interplay of all components of a computer system.

PARTICIPATION ACTIVITY 1.2.3: How a modern computer works.



Animation content:

Step 1: The CPU core instruction cycle involves reading instructions and read and writing data to main memory. Step 2: An instruction might trigger an I/O request to a device. Step 3: That request leads to data transfer with the device, and an interrupt to signal the CPU when the transfer is complete. Step 4: For larger mass media data movement, an instruction requests the I/O. Step 5: The I/O request triggers a DMA transfer of data from the device to main memory. Step 6: The interrupt informs the CPU of the I/O completion.

Animation captions:

1. The CPU core instruction cycle involves reading instructions and read and writing data to main memory.
2. An instruction might trigger an I/O request to a device.
3. That request leads to data transfer with the device, and an interrupt to signal the CPU when the transfer is complete.
4. For larger mass media data movement, an instruction requests the I/O.
5. The I/O request triggers a DMA transfer of data from the device to main memory.
6. The interrupt informs the CPU of the I/O completion.

PARTICIPATION ACTIVITY | 1.2.4: Section review questions.

1) A device controller informs a device driver it has completed an I/O operation using ________.

- ○ direct memory access (DMA)
- ○ an interrupt handler
- ○ an interrupt

2) What best describes a maskable interrupt?

- ○ An interrupt that can temporarily be disabled.
- ○ An interrupt that must be responded to immediately, regardless what the operating system is performing at the moment.
- ○ A technique for prioritizing various interrupts.

3) Which of the following forms of storage has the largest capacity?

- ○ registers
- ○ hard-disk drives
- ○ nonvolatile memory

Section glossary

bus: A communication system; e.g., within a computer, a bus connects various components, such as the CPU and I/O devices, allowing them to transfer data and commands.

device driver: An operating system component that provides uniform access to various devices and manages I/O to those devices.

interrupt: A hardware mechanism that enables a device to notify the CPU that it needs attention.

interrupt vector: An operating-system data structure indexed by interrupt address and pointing to the interrupt handlers. A kernel memory data structure that holds the addresses of the interrupt service routines for the various devices.

interrupt-request line: The hardware connection to the CPU on which interrupts are signaled.

interrupt-handler routine: An operating system routine that is called when an interrupt signal is received.

interrupt-controller hardware: Computer hardware components for interrupt management.

nonmaskable interrupt: An interrupt that cannot be delayed or blocked (such as an unrecoverable memory error)

maskable: Describes an interrupt that can be delayed or blocked (such as when the kernel is in a critical section).

interrupt chaining: A mechanism by which each element in an interrupt vector points to the head of a list of interrupt handlers, which are called individually until one is found to service the interrupt request.

interrupt priority level: Prioritization of interrupts to indicate handling order.

random-access memory (RAM): Rewritable memory, also called main memory. Most programs run from RAM, which is managed by the kernel.

dynamic random-access memory (DRAM): The common version of RAM, which features high read and write speeds.

bootstrap program: The program that allows the computer to start running by initializing hardware and loading the kernel.

volatile: Describes storage whose content can be lost in a power outage or similar event.

firmware: Software stored in ROM or EEPROM for booting the system and managing low level hardware.

bit: The basic unit of computer storage. A bit can contain one of two values, 0 or 1.

byte: Eight bits.

word: A unit made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words.

kilobyte (KB): 1,024 bytes.

megabyte (MB): $1,024^2$ bytes.

gigabyte (GB): $1,024^3$ bytes.

terabyte (TB): $1,024^4$ bytes.

petabyte (PB): $1,024^5$ bytes.

von Neumann architecture: The structure of most computers, in which both process instructions and data are stored in the same main memory.

secondary storage: A storage system capable of holding large amounts of data permanently; most commonly, HDDs and NVM devices.

hard disk drive (HDD): A secondary storage device based on mechanical components, including spinning magnetic media platters and moving read-write heads.

nonvolatile memory (NVM): Persistent storage based on circuits and electric charges.

tertiary storage: A type of storage that is slower and cheaper than main memory or secondary storage; frequently magnetic tape or optical disk.

semiconductor memory: The various types of memory constructed from semiconductors.

memory: Volatile storage within a computer system.

nonvolatile storage (NVS): Storage in which data will not be lost in a power outage or similar event.

nonvolatile memory (NVM): Persistent storage based on circuits and electric charges.

direct memory access (DMA): A resource-conserving and performance-improving operation for device controllers allowing devices to transfer large amounts of data directly to and from main memory.

1.3 Computer-system architecture

In Section 1.2, we introduced the general structure of a typical computer system. A computer system can be organized in a number of different ways, which we can categorize roughly according to the number of general-purpose processors used.

Single-processor systems

Many years ago, most computer systems used a single processor containing one CPU with a single processing core. The ***core*** is the component that executes instructions and contains registers for storing data locally. The one main CPU with its core is capable of executing a general-purpose instruction set, including instructions from processes. These systems have other special-purpose processors as well. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers.

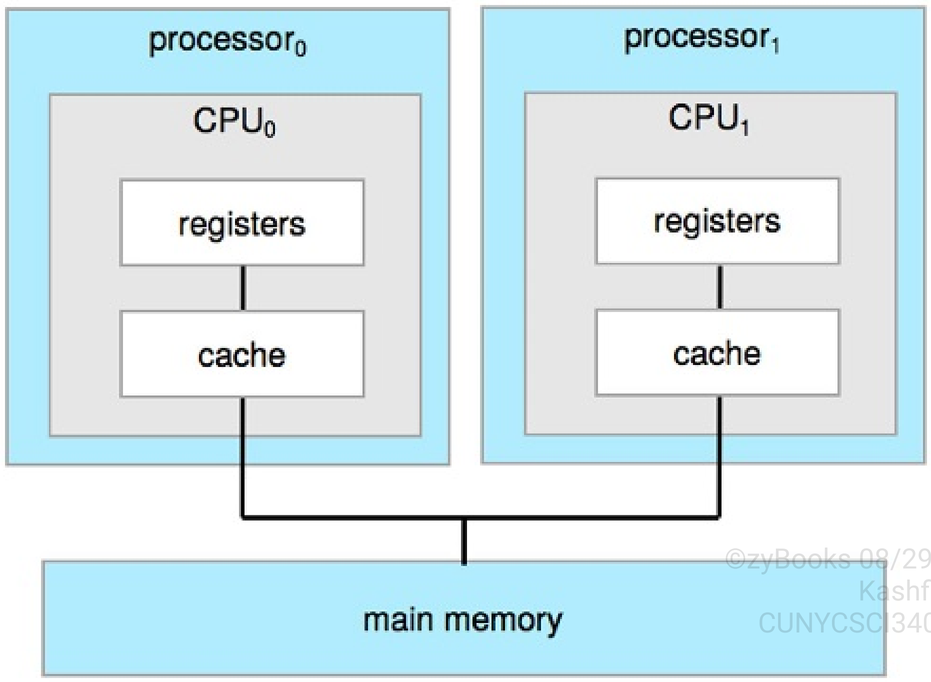
All of these special-purpose processors run a limited instruction set and do not run processes. Sometimes, they are managed by the operating system, in that the operating system sends them information about their next task and monitors their status. For example, a disk-controller microprocessor receives a sequence of requests from the main CPU core and implements its own disk queue and scheduling algorithm. This arrangement relieves the main CPU of the overhead of disk scheduling. PCs contain a microprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU. In other systems or circumstances, special-purpose processors are low-level components built into the hardware. The operating system cannot communicate with these processors; they do their jobs autonomously. The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor. If there is only one general-purpose CPU with a single processing core, then the system is a single-processor system. According to this definition, however, very few contemporary computer systems are single-processor systems.

Multiprocessor systems

On modern computers, from mobile devices to servers, ***multiprocessor systems*** now dominate the landscape of computing. Traditionally, such systems have two (or more) processors, each with a single-core CPU. The processors share the computer bus and sometimes the clock, memory, and peripheral devices. The primary advantage of multiprocessor systems is increased throughput. That is, by increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with N processors is not N, however; it is less than N. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors.

The most common multiprocessor systems use ***symmetric multiprocessing*** (*SMP*), in which each peer CPU processor performs all tasks, including operating-system functions and user processes. Figure 1.3.1 illustrates a typical SMP architecture with two processors, each with its own CPU. Notice that each CPU processor has its own set of registers, as well as a private—or local—cache. However, all processors share physical memory over the system bus.

Figure 1.3.1: Symmetric multiprocessing architecture.

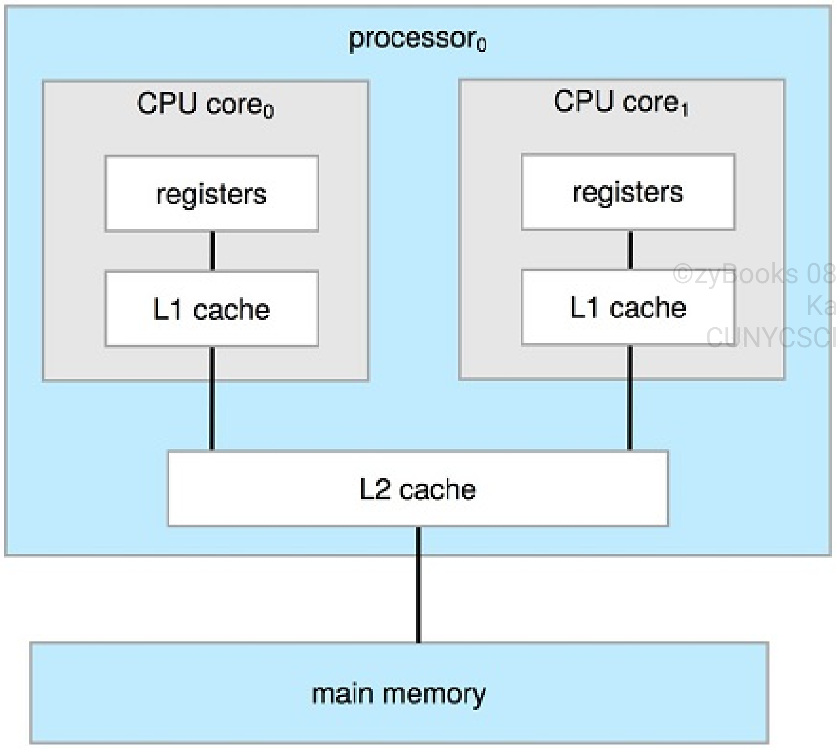


The benefit of this model is that many processes can run simultaneously—N processes can run if there are N CPUs—without causing performance to deteriorate significantly. However, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. These inefficiencies can be avoided if the processors share certain data structures. A multiprocessor system of this form will allow processes and resources—such as memory—to be shared dynamically among the various processors and can lower the workload variance among the processors. Such a system must be written carefully, as we shall see in the chapter CPU Scheduling and the chapter Synchronization Tools.

The definition of ***multiprocessor*** has evolved over time and now includes ***multicore*** systems, in which multiple computing cores reside on a single chip. Multicore systems can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication. In addition, one chip with multiple cores uses significantly less power than multiple single-core chips, an important issue for mobile devices as well as laptops.

In Figure 1.3.2, we show a dual-core design with two cores on the same processor chip. In this design, each core has its own register set, as well as its own local cache, often known as a level 1, or L1, cache. Notice, too, that a level 2 (L2) cache is local to the chip but is shared by the two processing cores. Most architectures adopt this approach, combining local and shared caches, where local, lower-level caches are generally smaller and faster than higher-level shared caches. Aside from architectural considerations, such as cache, memory, and bus contention, a multicore processor with N cores appears to the operating system as N standard CPUs. This characteristic puts pressure on operating-system designers—and application programmers—to make efficient use of these processing cores, an issue we pursue in the chapter Threads & Concurrency. Virtually all modern operating systems—including Windows, macOS, and Linux, as well as Android and iOS mobile systems—support multicore SMP systems.

Figure 1.3.2: A dual-core design with two cores on the same chip.



Definitions of computer system components

- ***CPU***—The hardware that executes instructions.
- ***Processor***—A physical chip that contains one or more CPUs.
- ***Core***—The basic computation unit of the CPU.
- ***Multicore***—Including multiple computing cores on the same CPU.
- ***Multiprocessor***—Including multiple processors.

Although virtually all systems are now multicore, we use the general term ***CPU*** when referring to a single computational unit of a computer system and ***core*** as well as ***multicore*** when specifically referring to one or more cores on a CPU.

PARTICIPATION ACTIVITY 1.3.1: Mid-section review question.

1) The ______ is the physical chip that contains one or more CPUs.

Check Show answer

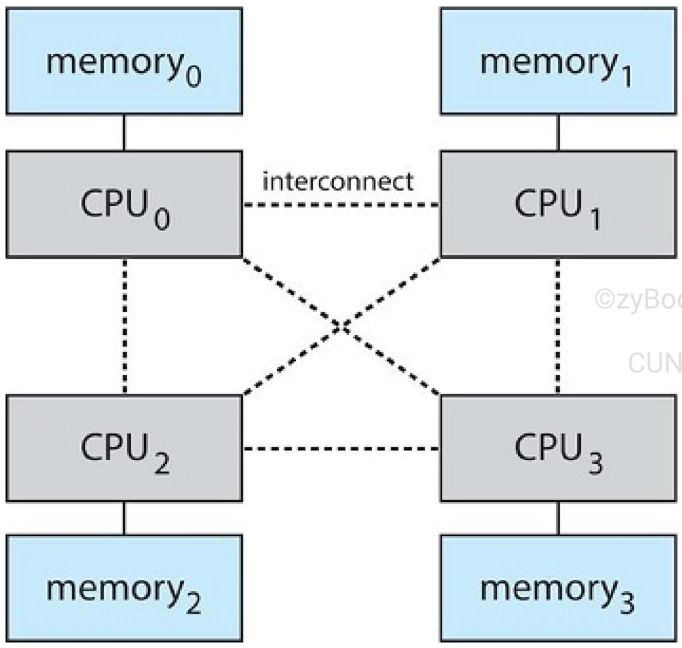
PARTICIPATION ACTIVITY 1.3.2: Mid-section review question.

1) Multicore also means multiple CPUs.

- True
- False

Adding additional CPUs to a multiprocessor system will increase computing power; however, as suggested earlier, the concept does not scale very well, and once we add too many CPUs, contention for the system bus becomes a bottleneck and performance begins to degrade. An alternative approach is instead to provide each CPU (or group of CPUs) with its own local memory that is accessed via a small, fast local bus. The CPUs are connected by a ***shared system interconnect***, so that all CPUs share one physical address space. This approach—known as ***non-uniform memory access***, or *NUMA*—is illustrated in Figure 1.3.3. The advantage is that, when a CPU accesses its local memory, not only is it fast, but there is also no contention over the system interconnect. Thus, NUMA systems can scale more effectively as more processors are added.

Figure 1.3.3: NUMA multiprocessing architecture.



A potential drawback with a NUMA system is increased latency when a CPU must access remote memory across the system interconnect, creating a possible performance penalty. In other words, for example, $\mathbf{CPU_0}$ cannot access the local memory of $\mathbf{CPU_3}$ as quickly as it can access its own local memory, slowing down performance. Operating systems can minimize this NUMA penalty through careful CPU scheduling and memory management, as discussed in Section Multicore Processors and Section Non-Uniform Memory Access. Because NUMA systems can scale to accommodate a large number of processors, they are becoming increasingly popular on servers as well as high-performance computing systems.

Finally, ***blade servers*** are systems in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system. Some blade-server boards are multiprocessor as well, which blurs the lines between types of computers. In essence, these servers consist of multiple independent multiprocessor systems.

Clustered systems

Another type of multiprocessor system is a ***clustered system***, which gathers together multiple CPUs. Clustered systems differ from the multiprocessor systems described in Section Multiprocessor systems in that they are composed of two or more individual systems—or nodes—joined together; each node is typically a multicore system. Such systems are considered ***loosely coupled***. We should note that the definition of ***clustered*** is not concrete; many commercial and open-source packages wrestle to define what a clustered system is and why one form is better than another. The generally accepted definition is that clustered computers share storage and are closely linked via a local-area network LAN (as described in the chapter Networks and Distributed Systems) or a faster interconnect, such as InfiniBand.

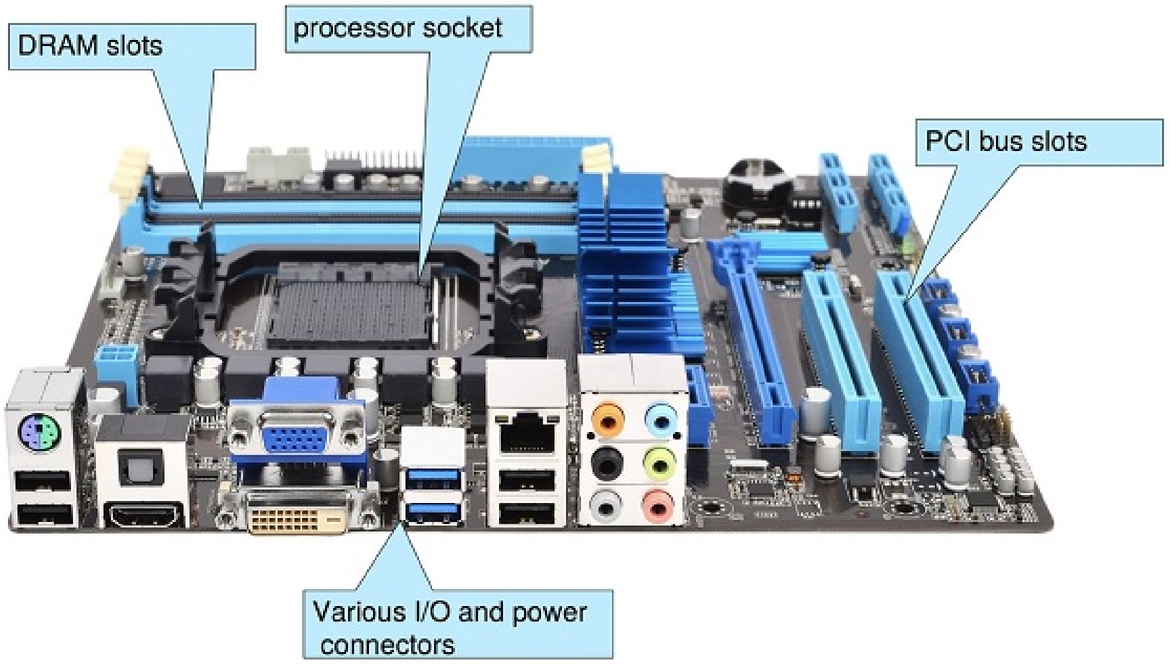
Clustering is usually used to provide ***high-availability service***—that is, service that will continue even if one or more systems in the cluster fail. Generally, we obtain high availability by adding a level of redundancy in the system. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the network). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. The users and clients of the applications see only a brief interruption of service.

High availability provides increased reliability, which is crucial in many applications. The ability to continue providing service proportional to the level of surviving hardware is called ***graceful degradation***. Some systems go beyond graceful degradation and are called ***fault tolerant***, because they can suffer a failure of any single component and still continue operation. Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected.

Clustering can be structured asymmetrically or symmetrically. In ***asymmetric clustering***, one machine is in ***hot-standby mode*** while the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server. In ***symmetric clustering***, two or more hosts are running applications and are monitoring each other. This structure is obviously more efficient, as it uses all of the available hardware. However, it does require that more than one application be available to run.

PC motherboard

Consider the desktop PC motherboard with a processor socket shown below:



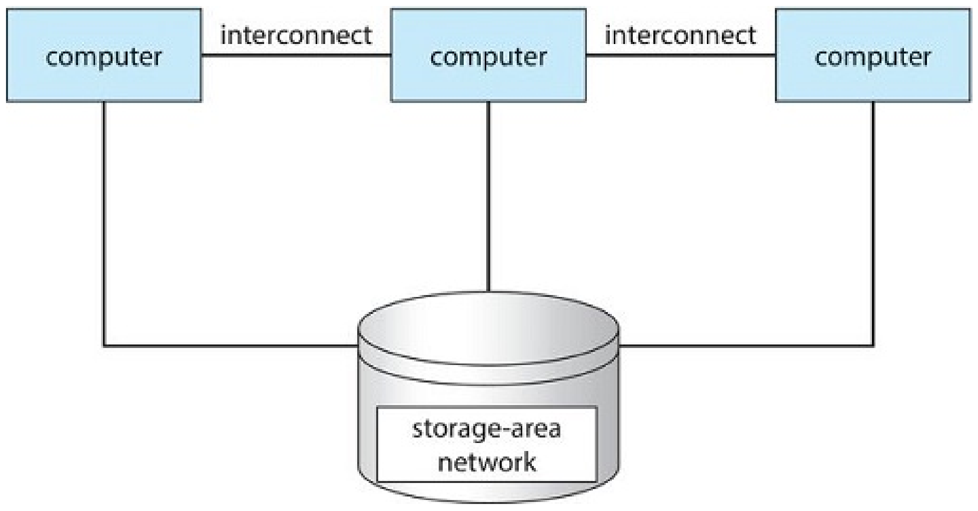
This board is a fully functioning computer, once its slots are populated. It consists of a processor socket containing a CPU, DRAM sockets, PCIe bus slots, and I/O connectors of various types. Even the lowest-cost general-purpose CPU contains multiple cores. Some motherboards contain multiple processor sockets. More advanced computers allow more than one system board, creating NUMA systems.

Since a cluster consists of several computer systems connected via a network, clusters can also be used to provide ***high-performance computing*** environments. Such systems can supply significantly greater computational power than single-processor or even SMP systems because they can run an application concurrently on all computers in the cluster. The application must have been written specifically to take advantage of the cluster, however. This involves a technique known as ***parallelization***, which divides a program into separate components that run in parallel on individual cores in a computer or computers in a cluster. Typically, these applications are designed so that once each computing node in the cluster has solved its portion of the problem, the results from all the nodes are combined into a final solution.

Other forms of clusters include parallel clusters and clustering over a wide-area network (WAN) (as described in the chapter Networks and Distributed Systems). Parallel clusters allow multiple hosts to access the same data on shared storage. Because most operating systems lack support for simultaneous data access by multiple hosts, parallel clusters usually require the use of special versions of software and special releases of applications. For example, Oracle Real Application Cluster is a version of Oracle's database that has been designed to run on a parallel cluster. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database. To provide this shared access, the system must also supply access control and locking to ensure that no conflicting operations occur. This function, commonly known as a ***distributed lock manager*** (*DLM*), is included in some cluster technology.

Cluster technology is changing rapidly. Some cluster products support thousands of systems in a cluster, as well as clustered nodes that are separated by miles. Many of these improvements are made possible by ***storage-area networks*** (*SANs*), as described in a Section Storage-Area Networks and Storage Arrays, which allow many systems to attach to a pool of storage. If the applications and their data are stored on the SAN, then the cluster software can assign the application to run on any host that is attached to the SAN. If the host fails, then any other host can take over. In a database cluster, dozens of hosts can share the same database, greatly increasing performance and reliability. Figure 1.3.4 depicts the general structure of a clustered system.

Figure 1.3.4: General structure of a clustered system.



PARTICIPATION ACTIVITY 1.3.3: Section review questions.

1) Which of the following statements regarding clustered systems is false?

- ○ They can provide high-availability service.
- ○ Because they are typically connected across a computer network, they cannot meet high-performance computing needs.
- ○ Data sharing is still possible on clustered systems even though the cluster consists of separate computer systems.

Section glossary

core: Within a CPU, the component that executes instructions.

multiprocessor systems: Systems that have two or more hardware processors (CPU cores) in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

symmetric multiprocessing (SMP): Multiprocessing in which each processor performs all tasks, including operating-system tasks and user processes.

multicore: Multiple processing cores within the same CPU chip or within a single system.

multiprocessor: multiple processors within the same cpu chip or within a single system.

shared system interconnect: A bus connecting CPUs to memory in such a way that all CPUs can access all system memory; the basis for NUMA systems.

non-uniform memory access (NUMA): An architecture aspect of many computer systems in which the time to access memory varies based on which core the thread is running on (e.g., a core interlink is slower than accessing DIMMs directly attached to core).

blade server: A computer with multiple processor boards, I/O boards, and networking boards placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system.

clustered system: A system that gathers together multiple CPUs. Clustered systems differ from multiprocessor systems in that they are composed of two or more individual systems—or nodes—joined together.

high-availability: Describes a service that will continue even if one or more systems in the cluster fail.

graceful degradation: The ability of a system to continue providing service proportional to the level of surviving hardware.

fault-tolerant system: A system that can suffer a failure of any single component and still continue operation.

asymmetric clustering: A configuration in which one machine in a cluster is in hot-standby mode while the other is running applications.

hot-standby mode: A condition in which a computer in a cluster does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server.

symmetric clustering: A situation in which two or more hosts are running applications and are monitoring each other.

high-performance computing: A computing facility designed for use with a large number of resources to be used by software designed for parallel operation.

parallelization: The process of dividing a program into separate components that run in parallel on individual cores in a computer or computers in a cluster.

distributed lock manager (DLM): A function used by a clustered system to supply access control and locking to ensure that no conflicting operations occur.

storage-area network (SAN): A local-area storage network allowing multiple computers to connect to one or more storage devices.