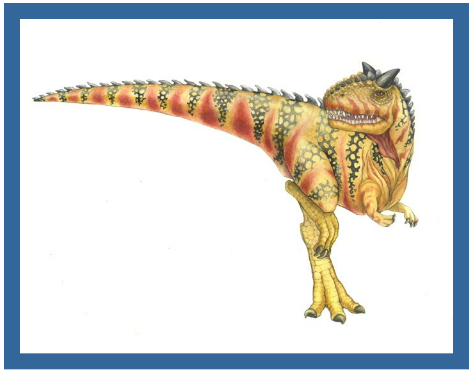
Chapter 1: Introduction



Objectives

- To describe the basic organization of computer systems
- To provide a grand tour of the major components of operating systems
- To give an overview of the many types of computing environments
- To explore several open-source operating systems

What is an Operating System?

- An **operating system**(**OS**) is a program that manages a computer's hardware:
  - provides a basis for **application programs** and
  - acts as an intermediary between the **user** and the computer **hardware**.
- **Operating system goals:**
  - Execute user programs and make solving user problems **easier**.
  - Make the computer system **convenient** to use.
  - Use the computer hardware in an **efficient** manner.

Computer System Structure

- A **computer system** can be divided into **four** components (**Figure 1.1**):
  - **Hardware**
    - provides basic computing resources such as CPU, memory, I/O devices
  - **OS**
    - Controls and coordinates use of hardware among various applications and users
  - **Application programs**
    - such as word processors, compilers, web browsers, database systems, video games
  - **Users**
    - People, other computers

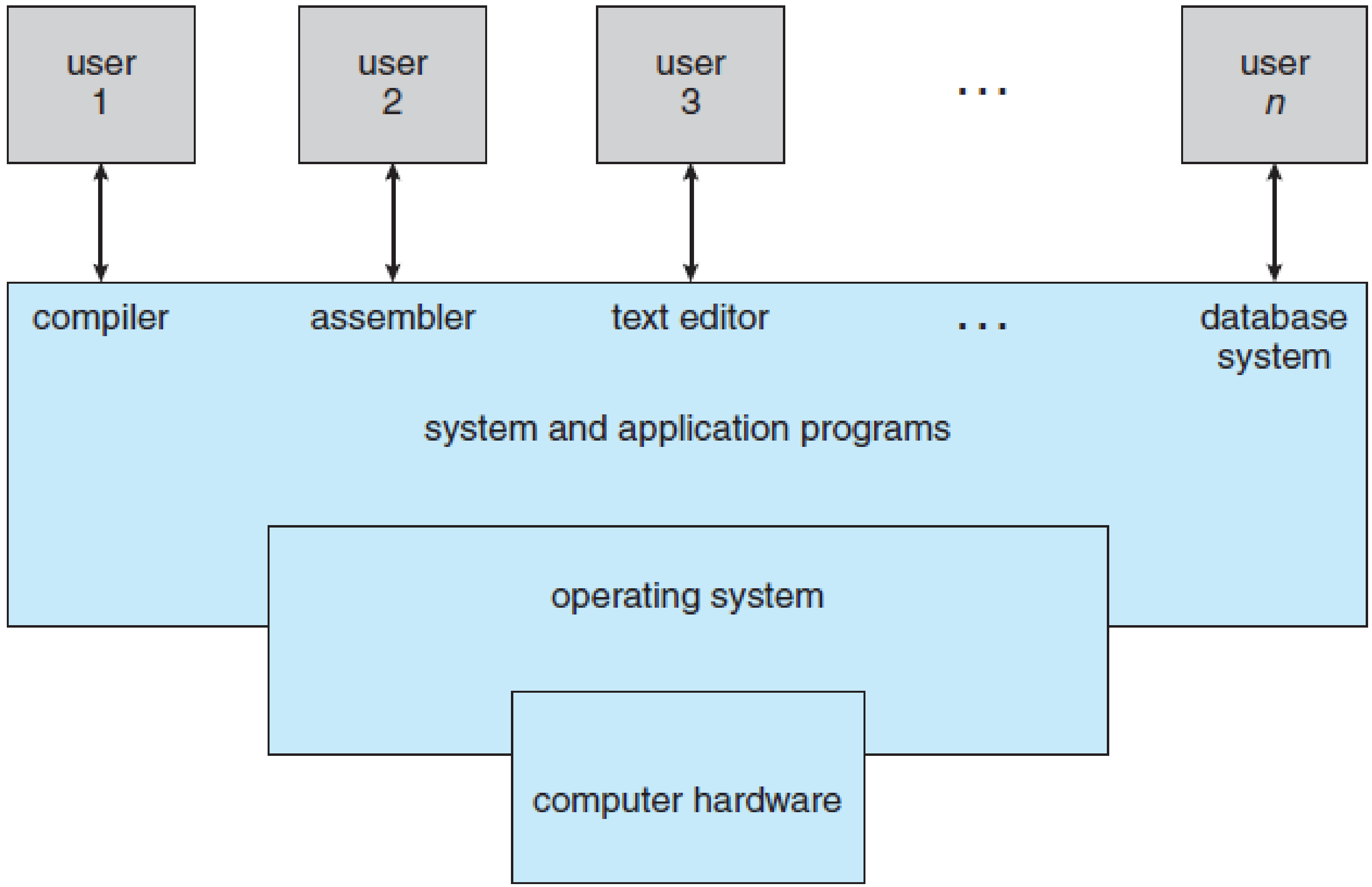


Figure 1.1 Abstract view of the components of a computer system.

What Operating Systems Do

- To understand an operating system's role, we need to explore the OS from following viewpoints:
  - **user** and
  - the **system**.

User View

- The **user's view** of the computer varies according to the system usage.
- When considering a **personal computer (PC) system**:
  - The OS is designed mostly for **ease of use**, with some attention paid to performance and none paid to **resource utilization**
    - how various hardware and software resources are shared.
  - Such systems are optimized for the **single-user** experience rather than the requirements of multiple users.

User View

- In other cases such as a **mainframe** or a **minicomputer** system:
  - various users are accessing the same computer through other terminals.
    - These users **share resources** and **exchange information**.
    - The OS in such cases is designed to **maximize resource utilization**
    - to assure that all available *CPU time*, *memory*, and I/O are used efficiently and that no individual user takes more than his or her fair share.

System View

- From the **computer's point of view**, an OS is a **resource allocator**.
- A computer system has many **resources** that may be required to solve problems such as:
  - **CPU time**, **memory space**, **file-storage space**, **I/O devices**, and so on.
- *The OS acts as the manager of these resources*.
  - Facing numerous and conflicting requests for resources;
    - the OS must decide how to allocate those resources to specific programs and
    - users so that it can make the computer system efficiently and fairly.

Defining Operating Systems

- **Operating systems** are essential because they solve the problem of creating a usable computing system.
  - The fundamental goal of **computer systems** is *to execute user programs and to make solving user problems easier*.
- The common functions of ***controlling*** and ***allocating*** resources (such as *memory*, *I/O*, etc) are brought together into one piece of software which is called the **operating system** (OS).

Defining Operating Systems

- The **OS** is a program that running at all times on a computer—usually called the **kernel,** it is the core of the OS and it facilitates interactions between HW and SW.
  - Along with the **kernel**, there are two other types of programs in a computer system:
    - **System programs**: Associated with the OS but are not necessarily part of the kernel, and
    - **Application programs**: Include all programs not associated with the OS.

Defining Operating Systems

- **Mobile operating systems** often include not only a **core kernel** but also **middleware**
  - **Middleware** is a set of software frameworks that provide additional services to *application developers*.
  - The mobile OS features a **core kernel** and **middleware** together supports databases, multimedia, and graphics.

Computer-System Operation

- A **general-purpose computer system** consists of one or more **CPUs** and a number of **device controllers** connected through a **common bus** that provides access to a **shared memory** (shown in **Figure 1.2**).
  - Each **device controller** is in charge of a specific type of device (for example, *disk drives*, *audio devices*, or *video displays*).
  - 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 the memory access during a program execution.

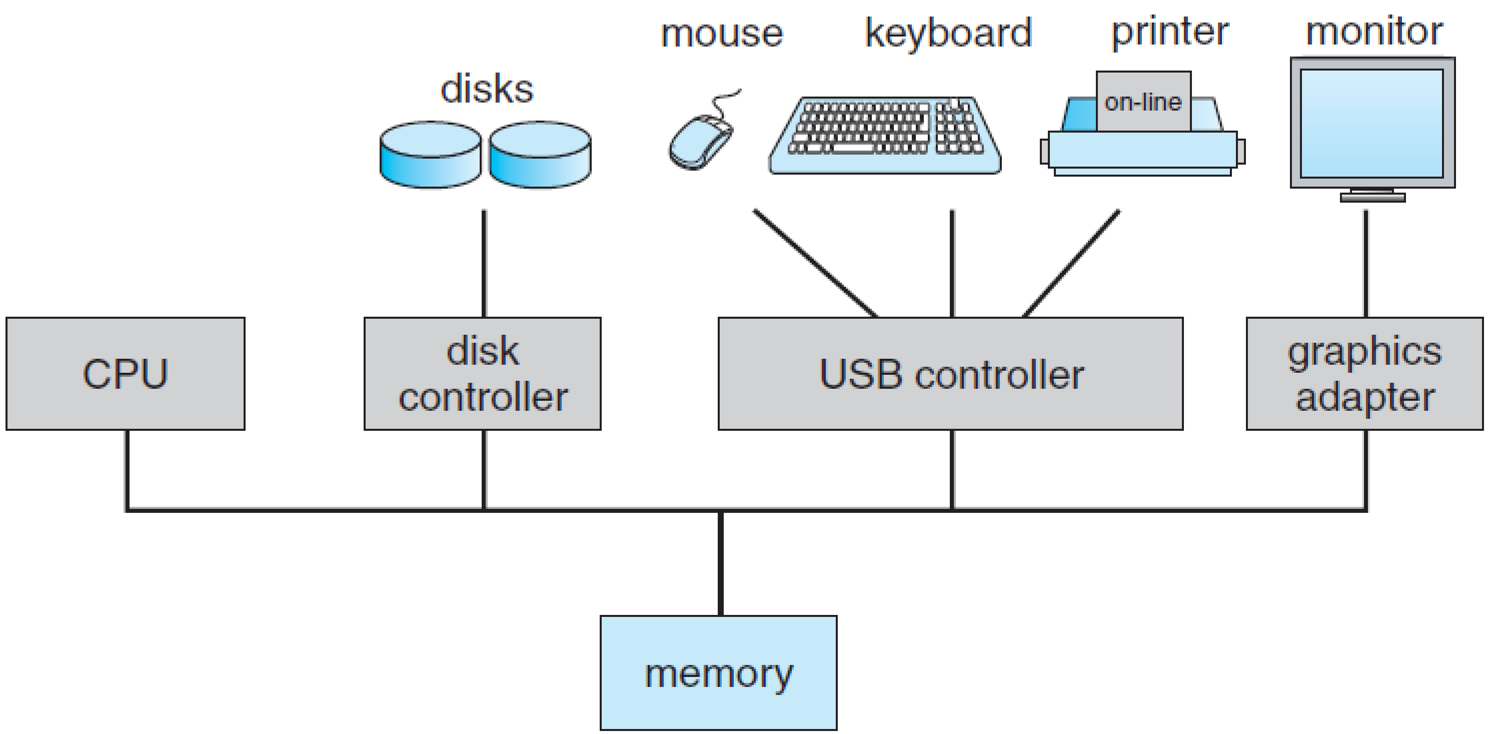


Figure 1.2 A modern computer system.

Computer-System Operation

- For a computer to start running, it needs to be powered up or **rebooted**—it needs to have an **initial program** to run.
- This **initial program** (or **bootstrap** program), tends to be simple**:**
  - The **initial program** is stored within the computer hardware in read-only memory **(ROM)** or electrically erasable programmable read-only memory **(EEPROM),** known by the term **firmware.**
  - It is loaded into the **main memory** for execution.
  - It initializes all aspects of the system, from CPU registers to device-controllers to memory contents.
  - The **bootstrap program** must know how to **load the OS** and how to start executing the system.

Computer-System Operation

- Once the OS **kernel** is loaded and executing, it can start providing services to the computer system and its user:
  - some services are provided **outside the** kernel by **system programs** loaded into memory at boot time.
  - The system programs coordinate data transfer across the various components and deal with *compiling*, *linking*, *starting*, *and stopping programs*, *reading from files*, *writing to files*, etc.
- The occurrence of *an* ***event*** in a computer system is usually signaled by an **interrupt** from either the *hardware* or the *software*.

Computer-System Operation

- **Hardware** may trigger an **interrupt** by sending a signal to the CPU, usually through the **system bus**.
- **Software** may trigger an **interrupt** by executing a special operation called a **system call** (also called a **monitor call).**
  - 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** of a new service routine (called **interrupt Service Routine, ISR**) for the interrupt.
  - The **ISR** executes; on completion, the CPU resumes the interrupted computation.

Computer-System Operation

- **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 (ISR).**
- The routine, in turn, would call the **interrupt-specific handler**.

Storage Structure

- During a program execution, the CPU can load instructions only from memory, so any programs to run must be stored there:
  - General-purpose computers run most of their programs from re-writable 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).**

Storage Structure

- Computers use other forms of memory as well.
  - We have already seen that, **read-only memory** (**ROM**) and **electrically erasable programmable read-only memory**, (**EEPROM**):
    - **ROM** cannot be changed, only static programs, such as the **bootstrap program** described earlier, are stored there.
    - **EEPROM** can be changed but cannot be changed frequently and so contains mostly static programs.
      - For example, smart phones have **EEPROM** to store their factory-installed programs.

Storage Structure

- 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:
    - ***load instruction*** moves a byte or word from main memory to an internal register within the CPU,
    - ***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.

Storage Structure

- We need to understand how a sequence of **main memory** addresses is generated by the running program.
- Before execution, the user program stays in the form of a **.exe** file (on a disk folder) and it must be loaded into the **main memory** for execution.
- Direct loading of the **.exe file** into memory is not possible due to the following two reasons:
  1. Main memory is usually too small to store all needed programs and data permanently.
  2. Main memory is a **volatile storage device that loses its contents when power is turned off**
- Thus, most computer systems provide **secondary storage** as an extension of **main memory**.

.

Storage Structure

- The most common **secondary-storage** device is a **Hard disk**, which provides storage for both programs and data.
  - Most programs (system and application) are stored on a disk until they are **loaded** into memory.
  - Many programs then use the disk as both the source and the destination of their processing.
    - The main differences among the various storage systems lie in speed, cost, size, and volatility.
- The wide variety of storage systems can be organized in a hierarchy (**Figure 1.4**) according to speed and cost.

Storage Structure

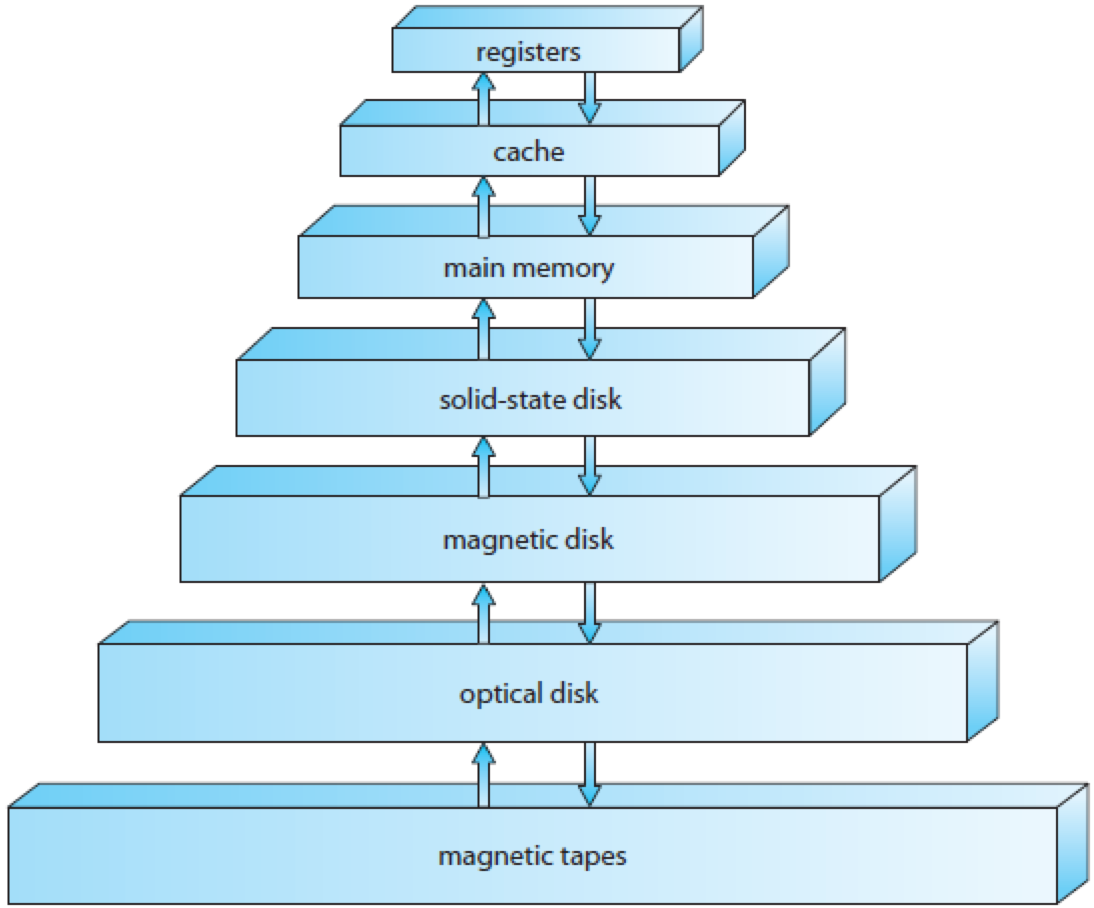


Figure 1.4 Storage-device hierarchy.

I/O Structure

- A large portion of **OS** code is dedicated to managing **I/O devices**.
- **Device controllers** that are connected to specific type of devices through a common bus.
  - The **device controller** is responsible for *moving the data between the peripheral devices that it controls and its local buffer storage*.
- Operating systems have **device drivers** (*device controlling SW*) for each **device controller.**
  - This **device driver** understands the **device controller** and provides the rest of the OS with a uniform interface to the device.

Computer-System Architecture

- **Figure 1.5** shows the interplay of all components of a computer system.

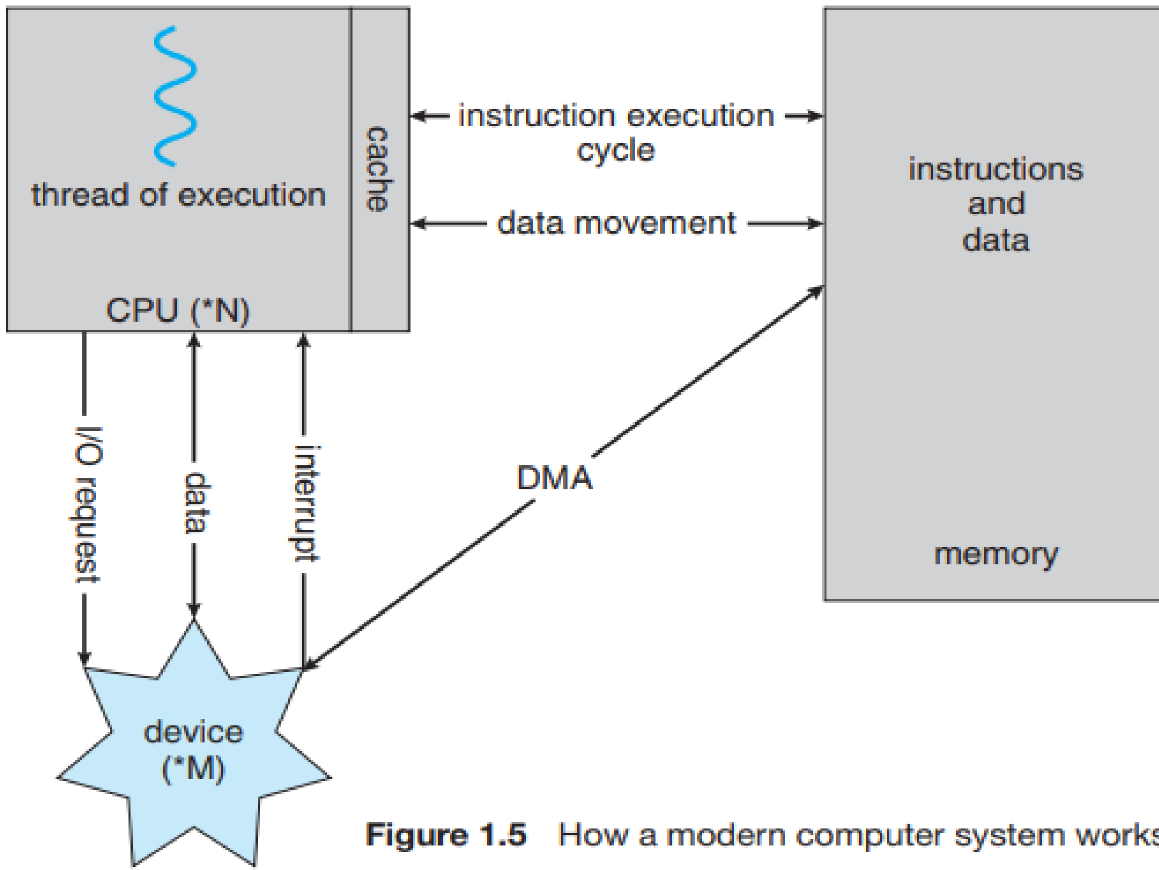


Figure 1.5 How a modern computer system works.

Computer-System Architecture

- A **computer system** can be organized in a number of different ways based on the number of **processors** used:
  - **Single-Processor** System
  - **Multiprocessor** System
  - **Clustered** System

Single-Processor Systems

- On a **single processor system**, there is **one** capable of executing a general-purpose instruction set (instructions from user process)
- Almost all single CPU systems have **other special-purpose Processors** as well:
    - such as disk, keyboard, graphics controllers, or I/O processors that move data rapidly among the components of the system.
  - All of these **special-purpose Processors** run a limited instruction set and do not run user processes (program).
  - For example, a *disk-controller* processor receives a sequence of requests from the CPU based on the scheduling algorithm.

Multiprocessor Systems

- Nowadays, **multiprocessor systems** (or **multi-core systems**) have begun to dominate computing.
  - Such systems have two or more Processors
    - sharing the computer bus and sometimes the clock, memory, and peripheral devices.
    - multiple processors have appeared on mobile devices such as *smartphones* and *tablet* computers.
- **Multiprocessor** systems have three main advantages:
  - **Increased throughput:** Get more work done in less time.
  - **Economy of scale:** Multiprocessor systems can cost less because they can share peripherals, mass storage, and power supplies.
  - **Increased reliability:** If functions can be distributed properly, then the failure of one Processor will not halt the system, only slowing it down.

Multiprocessor Systems

- The **multiprocessor** systems in use today are of two types:
  - **Asymmetric multiprocessing (AMP):**
    - in which each processor is assigned a specific task.
    - A master processor controls the system; the other processors either look to the boss for instruction or have predefined tasks.
    - There is no common or shared memory.
  - **Symmetric multiprocessing (SMP):**
    - in which each processor performs all tasks within the OS.
    - SMP means that all processors are peers; no master–worker relationship exists between processors.
    - sharing the system bus and sometimes the clock, memory, and peripheral devices.

Multiprocessor Systems

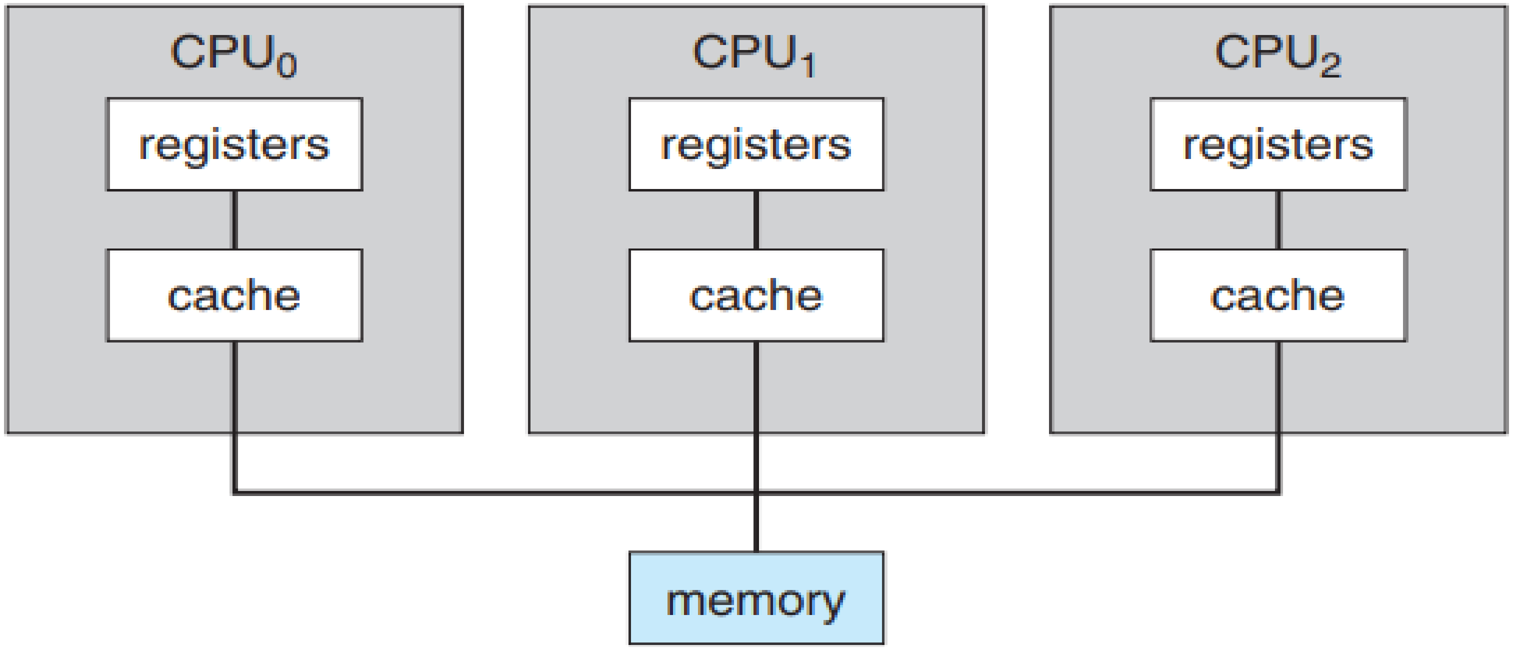


Figure 1.6 Symmetric multiprocessing architecture.

Multiprocessor Systems

- **Figure 1.7** shows a dual-core design with two cores on the same chip:
  - In this design, each core has its register set and local cache.
  - Other designs might use a shared cache or a combination of local and shared caches.
  - Aside from architectural considerations, such as cache, memory, and bus contention, these multicore CPUs appear to the operating system as ***N standard processors***.
  - This characteristic puts pressure on operating system designers and application programmers to use those processing cores.

Multiprocessor Systems

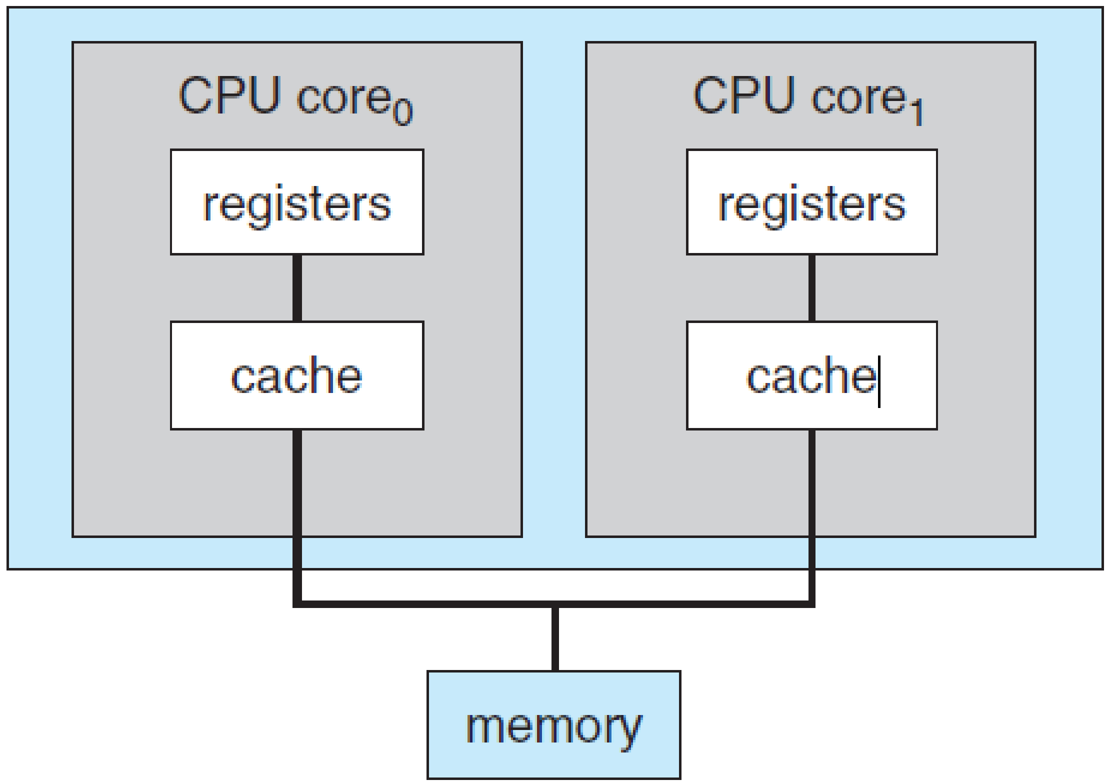


Figure 1.7 A dual-core design with two cores placed on the same chip.

Multiprocessor Systems

- Multiprocessing can cause a system to change its memory access model from **uniform memory access** (UMA) to **non-uniform memory access** (NUMA).
  - UMA is defined as the situation in which access to any RAM from any CPU takes the same amount of time.
- With **NUMA**, some parts of memory may take longer to access than others, creating a performance penalty.
  - Operating systems can minimize the NUMA penalty through resource management.

Multiprocessor Systems

- A recent trend in CPU design is to include multiple computing cores on a single chip.
- Such multiprocessor systems are termed **multi-core** (see **Figure 1.6**).
  - They 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.

Multiprocessor Systems

- In **multi-core** design, each core has its *register set* and *local cache*.
- Other designs might use a shared cache or a combination of local and shared caches.
- Aside from architectural considerations, such as cache, memory, and bus contention, these multi-core CPUs appear to the operating system as ***N*** standard processors.
- This characteristic pressures operating system designers and application programmers to use those processing cores.

Clustered Systems

- Another type of **multiprocessor** system is a **clustered system**, which gathers multiple CPUs.
- Clustered systems differ from multiprocessor systems in that they are composed of two or more individual systems—or nodes—joined together.
    - Such systems are considered **loosely coupled**.
- Each node may be a single processor system or a **multi-cor**e system.
    - The generally accepted definition is that clustered computers share storage and are closely linked via a local-area network LAN or a faster interconnect, such as **InfiniBand**

OS and Language Implementation

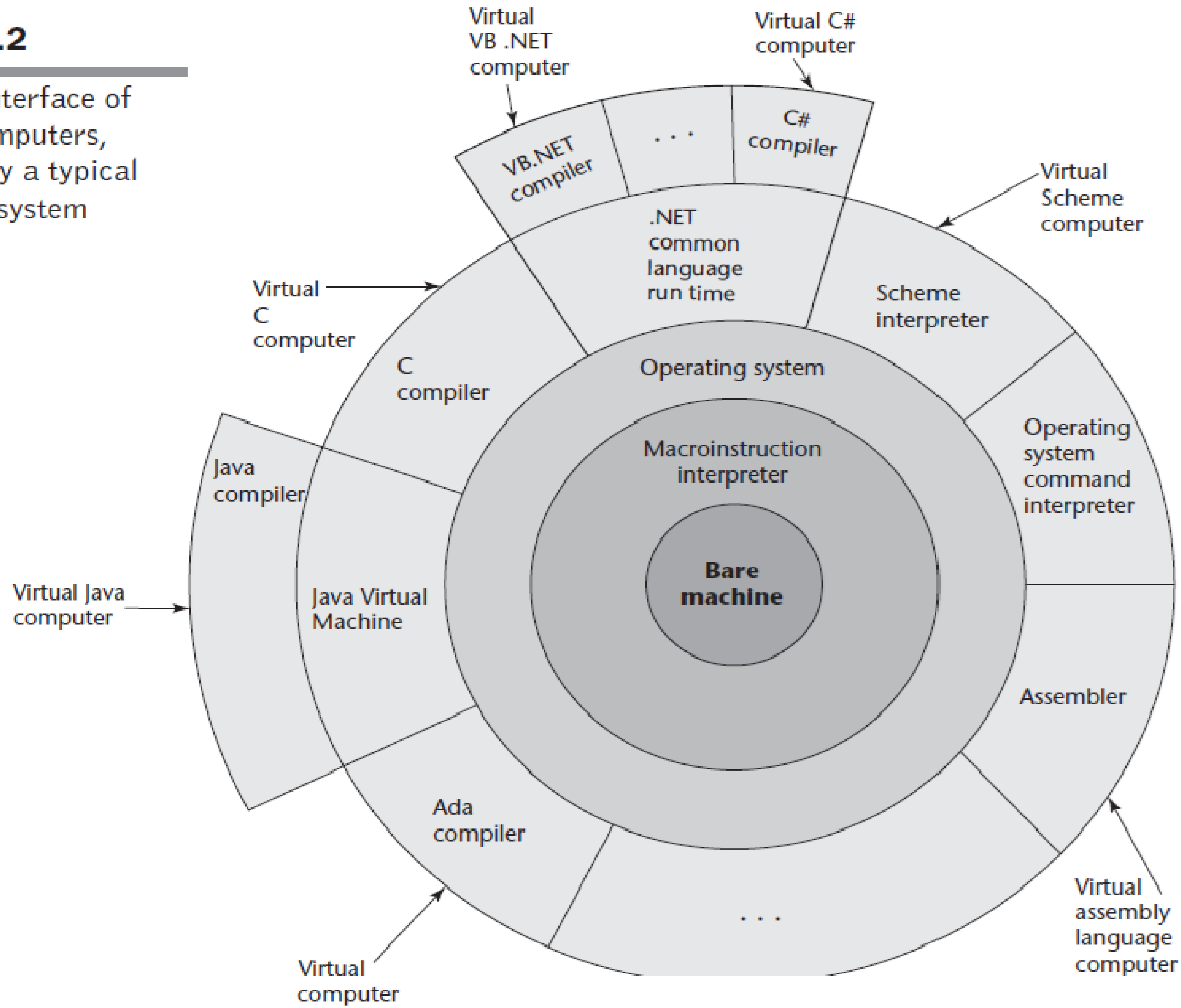
- A language implementation system in a computer requires **operating system** (OS) support, which supplies higher-level primitives than **machine language**.
  - The OS provides *resource management*, *input and output operations*, *a file management system*, *text and/or program editors*, and other commonly needed functions.
  - The language implementation systems need operating system facilities; they interface with the OS rather than directly with the processor (in machine language).

OS and Language Implementation

- The **OS** and language implementations are layered over the machine language interface of a computer.
- These layers can be considered **virtual computers**, providing interfaces to the user at higher levels:
  - For example, an OS and a **C compiler** provide a **virtual C computer**.
  - With other compilers, a machine can become another **virtual computer**.
  - Most computer systems provide several different virtual computers.
  - User programs form another layer over the top of the layer of virtual computers.
  - The layered view of a computer is shown in **Figure 1.2**.

Figure 1.2

Layered interface of virtual computers, provided by a typical computer system



Multiprogramming

- In **multiprogramming,** the OS keeps several jobs in **main memory** simultaneously for a single CPU (**Figure 1.9**).
- Since the main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool.**
  - This pool act as a **job queue** for all processes awaiting allocation of **main memory**.
- There must be enough memory to hold the OS (here the OS is called **resident monitor**) and one user program.
- **Multiprogramming** increases CPU utilization by organizing jobs so that the CPU always has one to execute.

Multiprogramming

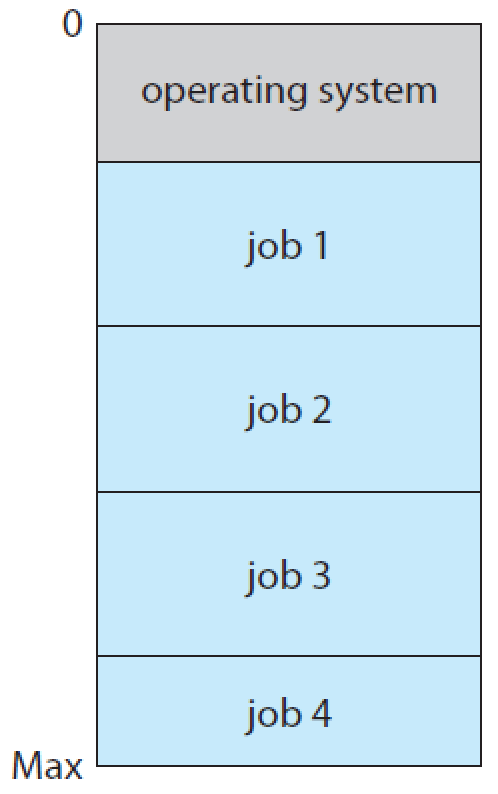


Figure 1.9 Memory layout for a multiprogramming system.

Multi-programming

- When one job needs to **wait for I/O**, the processor can switch to the other job which is not waiting for I/O
- This approach is called **multiprogramming** or **multitasking**
- When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O.

(a) Uniprogramming

Multi-programming

```
{
printf("\nEnter the first integer: ");
scanf("%d", &a);
printf("\nEnter the second integer: ");
scanf("%d", &b);
```

I/O cycle

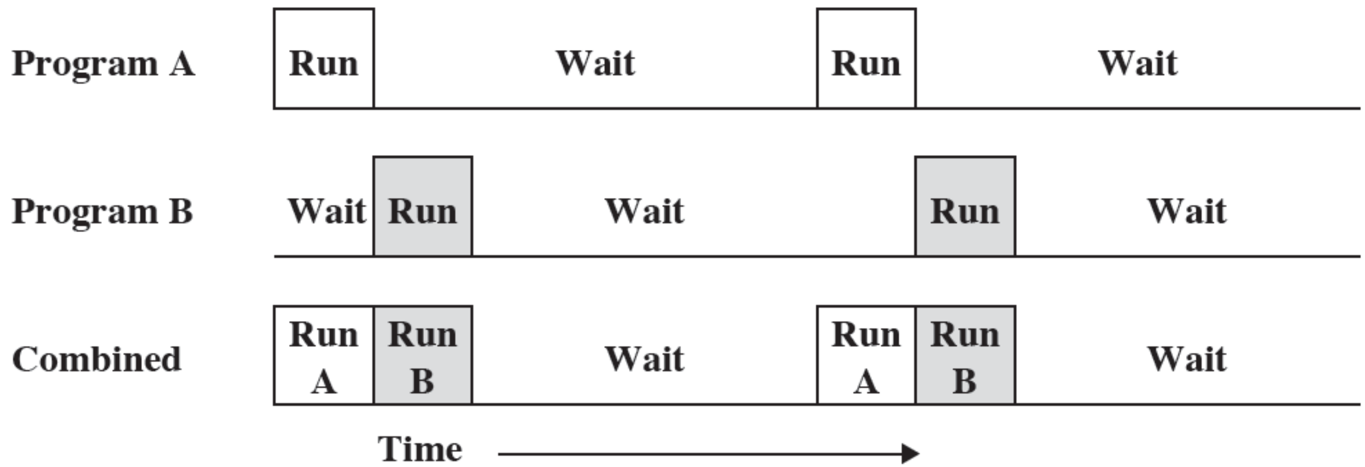
```
c = a+b;
d = (a*b)-c;
e = a-b;
f = d/e;
```

CPU cycle

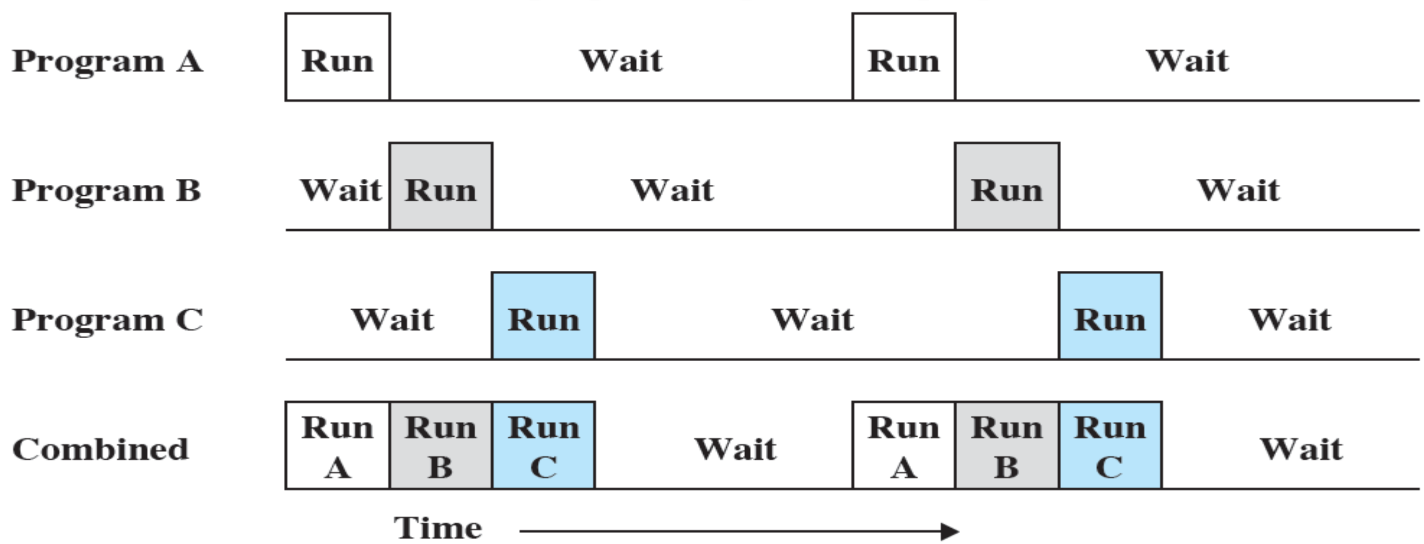
```
printf("\n a+b= %d", c);
printf("\n (a*b)-c = %d", d);
printf("\n a-b = %d", e);
printf("\n d/e = %d", f);
}
```

I/O cycle

Multi-programming



(b) Multiprogramming with two programs



(c) Multiprogramming with three programs

Time sharing

- **Time sharing** is a logical extension of **multiprogramming**.
- In **time-sharing systems**, the single CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.
- As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

Time sharing

- A **time-shared OS** uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer.
- Each user has at least one separate program in memory.
- A program loaded into memory and executing is called a **process.**
- When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O.
- I/O may be interactive; output goes to a display for the user, and input comes from a user's keyboard, mouse, or another device.

Job Scheduling vs. CPU Scheduling

- **Time sharing** and **multiprogramming** require that several jobs be kept simultaneously in **main memory**.
- If **several jobs** are ready to be brought into memory, and if there is not enough room for all of them, then the system keeps some of them in a **job pool** (or **job queue**) in a **disk memory**: Making this decision involves **job scheduling.**
- In addition, if several jobs are ready to run at the same time, the system must choose which job will run first. Making this decision is **CPU scheduling**.
  - Running multiple jobs concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management.

Virtual Memory in Time Sharing OS

- The time-sharing OS must ensure reasonable response time.
- This goal is sometimes accomplished through **swapping,** whereby processes are swapped in and out of **main memory (MM)** to the disk:
  - A common method for ensuring reasonable response time is **virtual memory (VM)**, a technique that allows the execution of a process that is not completely in memory.
  - The main advantage of the **VM** scheme is that it enables users to run programs larger than actual **physical memory.**
  - **VM** abstracts **MM** into a large, uniform array of storage, separating **logical memory** as viewed by the user from **physical memory.** This arrangement frees programmers from concern over memory-storage limitations.

Operating-System Operations

- Modern operating systems are **interrupt-driven** systems.
  - If there are no **processes** to execute, no **I/O devices** to service, and no **users** to whom to respond, an OS will sit quietly, waiting for something to happen.
- Events are almost always signaled by the occurrence of an **interrupt** or a **trap, or** a **system call**.
- The **system call** is a **software-generated interrupt** caused either by an error (for example, *division by zero* or *invalid memory access*) or a *specific request* from a user program that an OS service is performed.
- An **interrupt service routine** is an OS-predefined program that will deal with the interrupt.

Dual-Mode and Multimode Operation

- To ensure the proper execution of the OS, it is to distinguish between the execution of **operating-system code** and **user-defined code**.
- Based on these, we need **two separate modes of operation:**
  - **user mode** and
  - **kernel mode**
- A **bit**, called the **mode bit,** is added to the HW of the computer to indicate the current mode:
  - **kernel mode (0)** or **user mode (1).**

Dual-Mode and Multimode Operation

- With the **mode bit**, we can distinguish between a task that is executed either in **User mode** or in **Kernel mode**:
  - **User mode**: When the computer system is executing on behalf of a user application, the system is in user mode.
  - **Kernel mode**: when a user application requests a service from the OS (via a **system call**), the system must transition from ***user*** to ***kernel*** mode to fulfill the request.
- The mode transition of OS is shown in **Figure 1.10**.

Dual-Mode and Multimode Operation

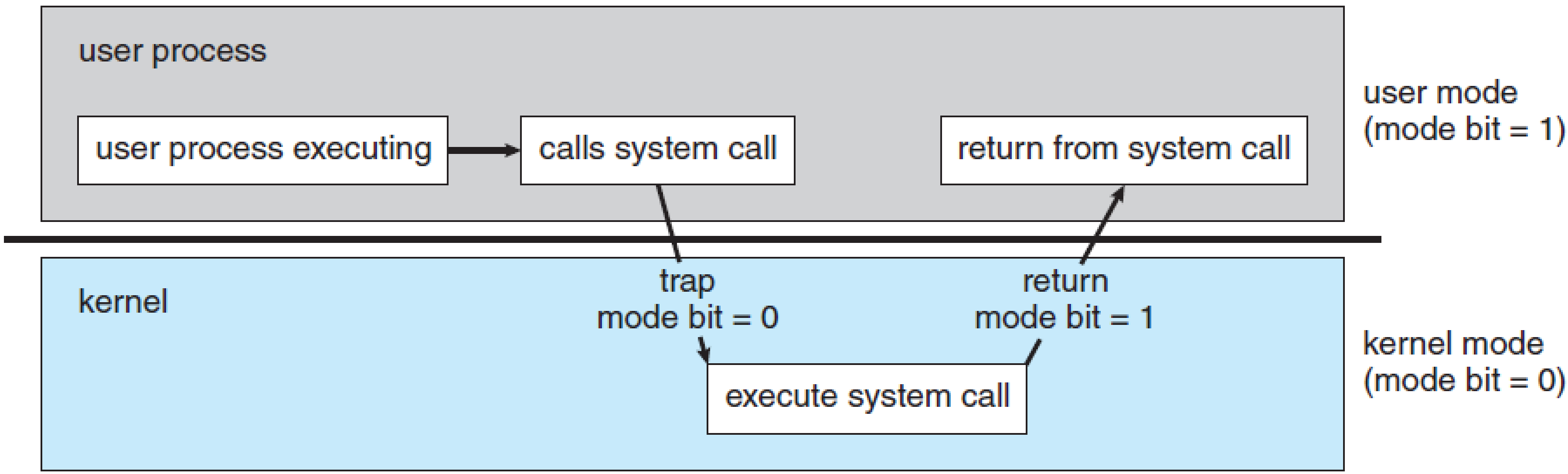


Figure 1.10 Transition from user to kernel mode.

Dual-Mode and Multimode Operation

- At system **boot** time, the system is in **kernel mode**.
- The OS is then loaded into **main memory** and starts user applications in **user mode**.
- Whenever a **trap** or **system call** occurs, the HW switches from ***user mode*** to ***kernel mode*** as shown in **Figure 1.10**
  - Thus, whenever the operating system gains control of the computer, it is **in kernel mode**.
- The system always switches to **user mode** (by setting the mode bit to 1) before passing control to a **user program**.

Dual-Mode and Multimode Operation

- The **dual mode** of operation provides us with the means for protecting the OS by designating some of the machine instructions called **privileged instructions.**
- The hardware allows **privileged instructions** to be executed only in **kernel mode**.
- If an attempt is made to execute a privileged instruction in **user mode**, the hardware does not execute the instruction but treats **it as illegal** and **traps** it to the OS.

System Call

- **System calls** provide the means for a *user program* to ask the OS to perform tasks reserved for the OS on the user program's behalf.
  - A **system call** is invoked in various ways, depending on the functionality provided by the underlying processor.
    - It is the method used by a process to request action by the OS.
- A **system call** usually takes the form of a **trap** to a specific location in the **interrupt vector**.
  - The **system-call service routine** is a part of the OS.
  - The **kernel** examines the interrupting instruction to determine what **system call** has to occur.

Process Management

- A program in execution, is called a **process**.
  - A compiler is a process.
  - A word-processing program being run by an individual user on a PC is a process.
  - A system task, such as sending output to a printer, can also be a process (or at least part of one).
- A process needs certain resources—including *CPU time*, *memory*, *files*, and *I/O devices*—to accomplish its task.
  - These resources are either given to the process when it is created or while it is running.
  - When the process terminates, the OS will reclaim any **reusable resources**.

Process Management

- The CPU executes one instruction at a time form the process (it can be called a ***thread***) one after another, until the process completes.
  - A **single-threaded process** has one **program counter** specifying the next instruction to execute.
- Thus, although two processes may be associated with the same program, they are considered two separate execution sequences.
  - A multithreaded process has multiple **program counters**, each pointing to the next instruction to execute for a given thread.

Process Management

- A system consists of a **collection of processes**, some of which are **OS processes** (running- OS code) and the rest of which are **user processes.**
  - All these processes can potentially execute concurrently—by multiplexing on a single CPU.
- The OS is responsible for the following activities in connection with process management:
  - **Scheduling** processes and threads on the CPU.
  - **Creating** and **deleting** both user and system processes.
  - **Suspending** and **resuming** processes.
  - Providing mechanisms for **process synchronization**.
  - Providing mechanisms for **process communication**.

Memory Management

- The **main memory** is central to the operation of a computer system (von Neumann architecture).
  - **Main memory** is a large array of bytes, each byte has its own address.
  - **Main memory** is a repository of quickly accessible data shared by the CPU and I/O devices.
    - The **main memory** is only the large storage device that the CPU is able to address and access directly.
- The CPU reads instructions from **main memory** during the **instruction-fetch cycle** and both reads and writes data from main memory during the **data-fetch cycle**.
  - **The data which is stored on disk must be transferred to main memory by CPU-generated I/O calls**.

Memory Management

- For a program to be executed, it must be mapped from the **disk address** to the **main memory address** (or **absolute addresses**) and then loaded into the **main memory.**
  - As the program executes, the CPU accesses program instructions and data from memory by these **absolute addresses**.
- Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

Memory Management

- To improve both the utilization of the CPU and the performance of the computer system, a need for **memory management** is important:
  - Many different memory management schemes are used.
- The OS is responsible for the following activities as part of **memory management**:
  - ***Keeping track of which parts of memory are used and who is using them.***
  - ***Deciding which processes (or parts of processes) and data to move into and out of memory.***
  - ***Allocating and de-allocating memory space as needed.***

File-System Management

- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, is the **file** unit.
  - The OS maps **files** onto physical media and accesses these files via the storage devices.
- The OS is responsible for the following activities in connection with **file management**:
  - Creating and deleting files.
  - Creating and deleting directories to organize files.
  - Supporting primitives for manipulating files and directories.
  - Mapping files onto secondary storage.
  - Backing up files on stable (nonvolatile) storage media.

Mass-Storage Management

- The OS is responsible for the following activities in connection with **disk management**:
  - Free-space management
  - Storage allocation
  - Disk scheduling

Caching

- **Caching** is an important principle of computer systems.
- Information is normally kept in some storage system (such as main memory). As it is used, it is temporarily copied into a faster storage system—the cache.
- When we need particular information, we first check whether it is in the cache.
- If it is, we use the information directly from the cache.
- If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.

Caching

- Main memory can be viewed as a fast cache for secondary storage since data in secondary storage must be copied into main memory for use.
- Data must be in the **main memory** before being moved to secondary storage for safekeeping.
- The file-system data, which resides permanently in secondary storage, may appear on several levels in the storage hierarchy.

Storage Management

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Figure 1.11 Performance of various levels of storage.

I/O Systems

- One of the purposes of an OS is to hide the peculiarities of specific hardware devices from the user.
  - For example, in UNIX, the peculiarities of I/O devices are hidden from the bulk of the OS itself by the **I/O subsystem.**
- The **I/O subsystem of the OS** consists of several components:
  - A **memory-management component** that includes buffering, caching, and spooling
  - A general **device-driver** interface
  - Drivers for **specific hardware** devices
- Only the device driver knows the peculiarities of the specific device to which it is assigned.

End of Chapter 1