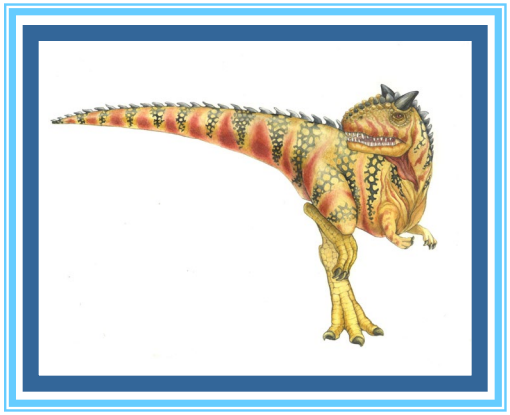
Chapter 2: Operating-System Structures



Chapter 2: Operating-System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Operating System Debugging
- Operating System Generation
- System Boot

Objectives

- To describe the services an OS provides to users, processes, and other systems.
- To discuss the various ways of structuring an OS.
- To explain how operating systems are installed and customized and how they boot.

Operating System Services

- Operating systems provide an environment for executing programs and services to programs and users.
- The major **OS services** that helpful to the user are:
  - **User interface** - Almost all operating systems have a user interface (UI).
    - **Command-Line Interface (CLI)**, **Graphics User Interface (GUI)**, and **Batch**
  - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error).
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device.

Operating System Services (Cont.)

- **File-system manipulation** - Programs need to read and write files and directories, create and delete them, search them, list file Information, and manage permissions.
- **Communications** – Processes may exchange information on the same computer or between computers over a network
  - Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.
- **Error detection** – OS needs to be constantly aware of possible *errors*
  - Errors may occur in the CPU and memory hardware, in I/O devices, and in the user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user's and programmer's abilities to use the system efficiently.

Operating System Services (Cont.)

- **Resource allocation -** When multiple users or multiple jobs are running concurrently, resources must be allocated to each of them
  - Many types of resources - *CPU cycles*, *main memory*, *file storage,* I/O *devices*, etc.
- **Accounting -** To track which users use how much and what kinds of computer resources, etc.
- **Protection and security -** The owners of information stored in a multiuser or networked computer system control the use of that information;
  - **Concurrent processes** should not interfere with each other.
  - **Protection** involves ensuring that all access to system resources is controlled.
  - **Security** of the system from outsiders requires user authentication, including I/O devices, to prevent invalid access attempts

A View of Operating System Services

Figure 2.1 shows one view of the various operating-system services and how they interrelate.

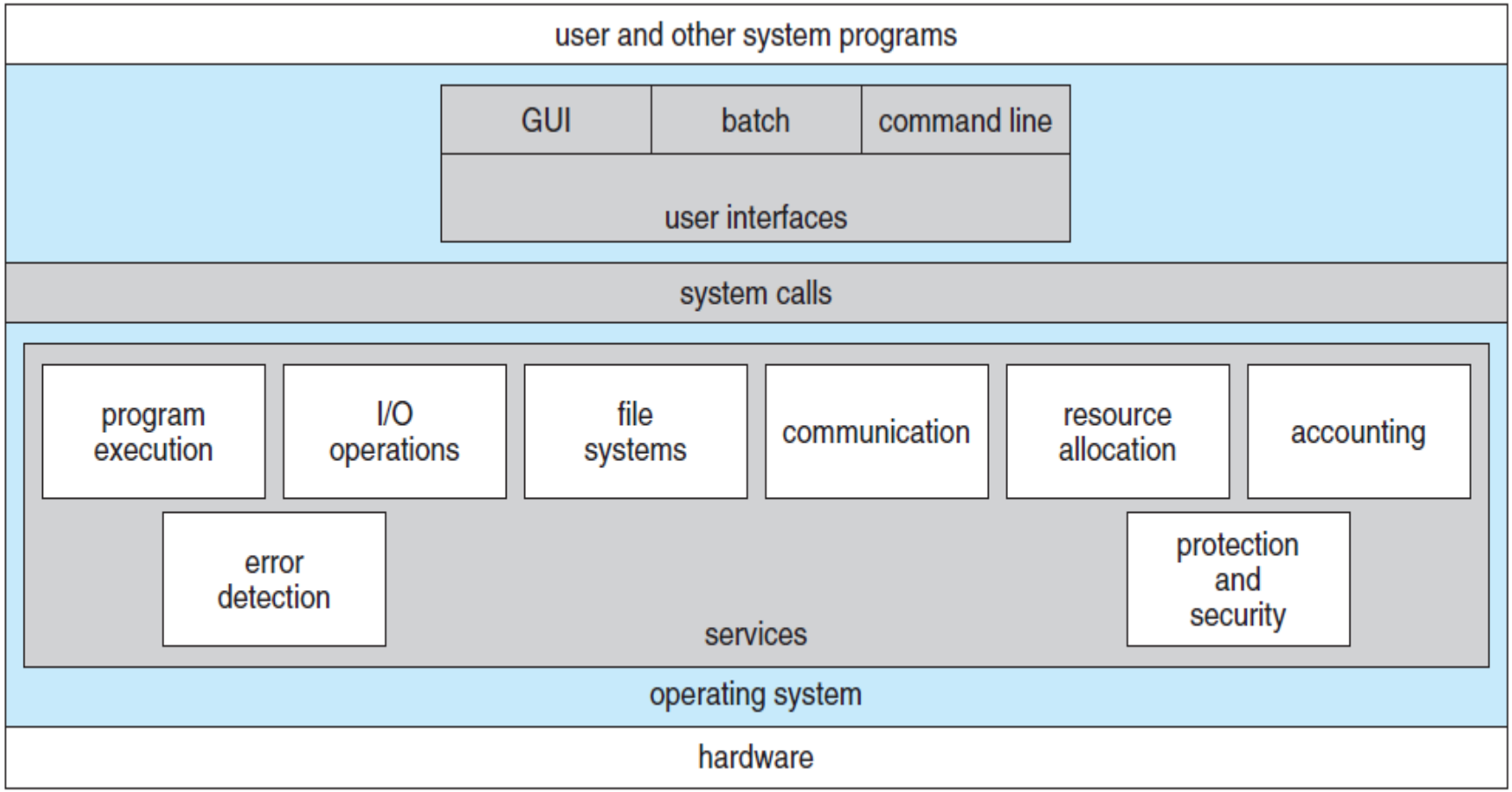


Figure 2.1 A view of operating system services.

User and Operating-System Interface

- Two fundamental **user-OS interface** approaches are:
  - A **command-line** or **command interpreter** interface
    - that allows users to directly enter commands to be performed by the operating system.
    - For example, the command line by DOS, Linux, etc
  - A **graphical user interface** (**GUI**)
    - For example, Windows

Command Interpreters

- Most operating systems, including **Linux**, **UNIX**, and **Windows**, treat the **command interpreter** as a special program that is running when **a process is initiated** or when a user **first logs on** (on interactive systems).
- On systems with **multiple command interpreters** to choose from, the interpreters are known as **shells**.
  - For example, on **UNIX** and **Linux** systems, a user may choose among several different shells, including the ***C shell***, ***Bourne-Again shell (bash)***, ***Korn shell***, and others.
  - **Figure 2.2** shows the ***bash shell*** command interpreter being used on **macOS**.
- The main function of the **command line** is to get and execute the given user-specified command.
    - *create*, *delete, list*, *print*, *copy*, *execute*, and so on.
  - The MS-DOS and UNIX shells operate in this way.

Command Interpreters

- These commands can be implemented in **two general ways**:
  - In **one approach**, the command interpreter itself contains the code to execute the command. For example, a command to ***delete a file*** may cause the **command interpreter** to jump to a section of its **code** that sets up the parameters and makes the appropriate **system call**.
  - **An alternative approach—used by UNIX**—implements most commands through **system programs:**
    - In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed (**without a system call**).
    - Thus, the **UNIX command** to **delete** a file: **rm file.txt** would search for a pre-defined file called **rm**, load the file into memory, and execute it with the parameter **file.txt**.

Graphical User Interface (GUI)

- In **GUI**, instead of entering commands directly via a **command-line** interface, users employ a mouse-based window and menu system characterized by a **desktop metaphor**:
  - The user moves the mouse to position its pointer on images or icons **on the screen (the desktop)** that represent programs, files, directories, and system functions.
  - **The first GUI appeared on the Xerox Alto computer in 1973**.
  - However, the GUI became more widespread with the advent of **Apple Macintosh computers** in the **1980s**.
  - **Microsoft's** first version of **Windows—Version 1.0**—was based on the addition of a GUI interface to MS-DOS.
  - Traditionally, **UNIX systems** have been dominated by **command-line interfaces**.

Choice of Interface

- Because either a **command-line interface** or a **mouse-and-keyboard system** is impractical for most mobile systems, smartphones and handheld tablet computers typically use a **touch-screen** interface.
  - Both the **iPad** and the **iPhone** use the **Springboard** touch-screen interface.
- **Command-line interfaces** usually make repetitive tasks easier, in part because they have their own programmability.
  - For example, if a frequent task requires a set of **command-line steps**, those steps can be recorded into a file, and that file can be run just like a program.
  - This program **is not compiled into executable code** but rather is **interpreted by the command-line interface**.
  - These **shell scripts** are very common on systems that are command-line oriented, such as **UNIX** and **Linux**.

System Calls

- **System calls** provide an interface to the services made available by an OS.
- Let us see how **system calls** are used in file transfer using the command line:
  - Assume a simple program to read data from **one file** and copy them to **another file** (the output file does not exist)
  - The first name of the input and the output files:
    - **One approach** is for the program to ask the user for the names.
    - In an interactive system, this approach will require a sequence of **system calls**, first to write a prompting message on the screen and then to read names from the keyboard.
  - Once the two file names have been obtained, the program must **open** the **input file** and **create** the **output file**.

System Calls

- Each of these operations requires another **system call**.
  - Possible **error conditions** for each operation can require additional system calls.
    - If there is no file of that name or the file is protected against access.
    - In these cases, the program prints an error message on the console and then terminates.
- When both files are set up, a loop statement that reads from the **input file** (a system call) and writes to the **output file** (another system call).
- Finally, after the entire file is copied, the program may **close** both files (another system call), write a message to the console or window (one more system call), and finally **terminate** normally (the final system call). This system-call sequence is shown in **Figure 2.5.**

System Calls

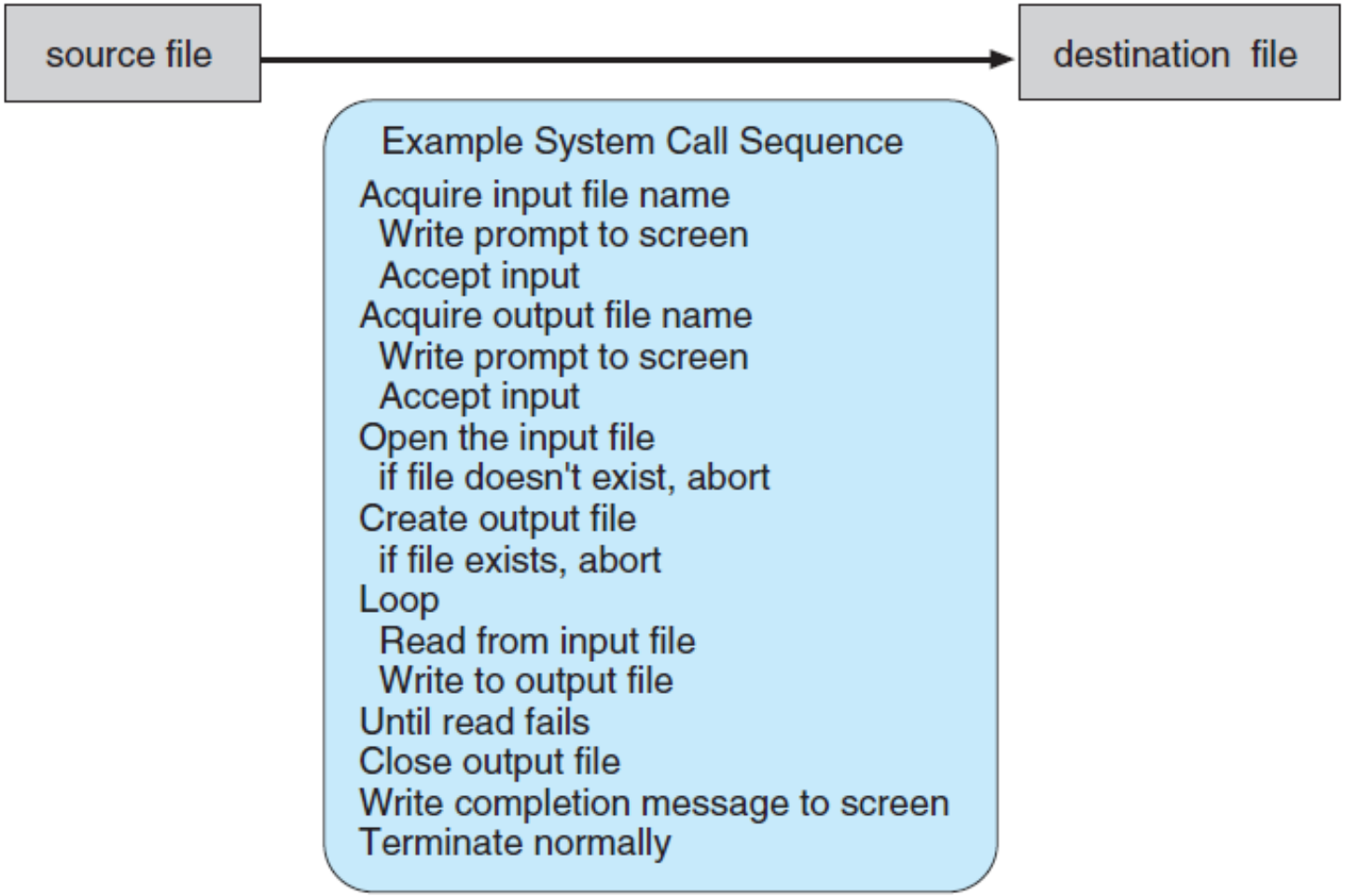


Figure 2.5 Example of how system calls are used.

Application Programming Interface(API)

- The **API** specifies a set of functions that are available to an application programmer. The interface can be thought of as a contract of service between two applications.
  - This contract defines how the two communicate with each other using requests and responses. **Google Maps API** is the most widely used API
- The most common **APIs** are the **Windows API** for Windows systems, the **POSIX API** for UNIX, Linux, and Mac OSX systems, and the **Java API** for the **Java virtual machine (JVM)** and **Android OS**.
- A programmer accesses an **API** via a library of code provided by the **OS**.
  - In the case of **UNIX** and **Linux** for programs written in the **C** language, the library is called **libc**.
- The **Operating System API** enables you to access various features provided by the **operating system** of the device.

System Calls

- Another important factor in handling **system calls** is the **run-time environment** (**RTE**).
- The **RTE** provides a **system-call interface** that serves as the link to **system calls** made available by the **OS**.
- The **system-call interface** intercepts function calls in the **API** and invokes the necessary **system calls** within the **OS**.
- A **number** is associated with each **system call**, and the **system-call interface** maintains a **table indexed** according to these numbers.
- The **system call interface** then invokes the intended **system call** in the OS kernel and **returns** the status of the system call.
- The relationship among an **API**, **the system-call interface**, and the **OS** is shown **in Figure 2.6**, which illustrates how the OS handles a user application invoking the **open()** system call.

System Calls

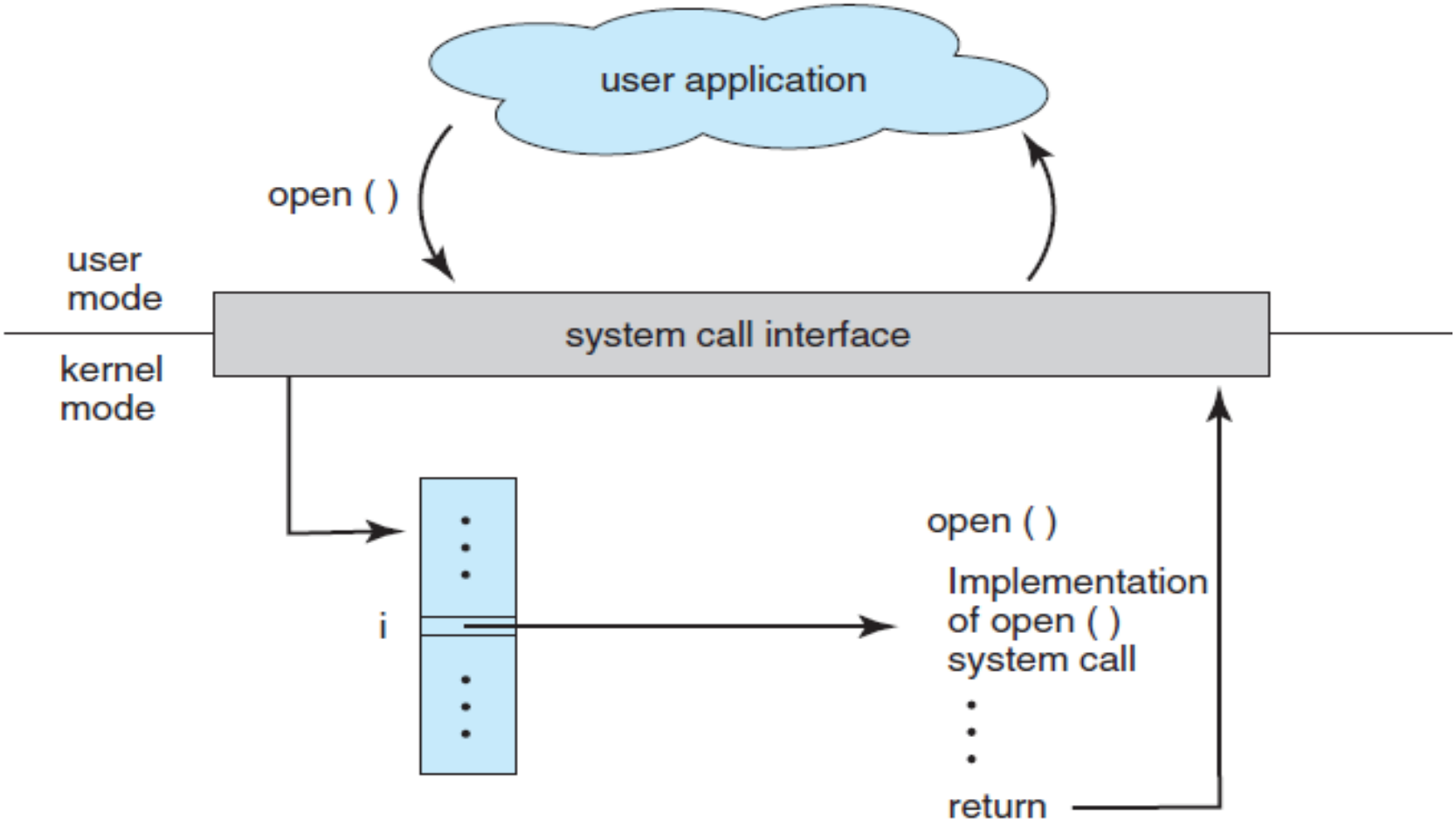


Figure 2.6 The handling of a user application invoking the `open()` system call.

System Calls

- **Three general methods** are used to **pass parameters** to the OS during a system call.
- The **simplest approach** is to pass the parameters in **registers**.
- **More complex approach**:
  - In some cases, there may be **more parameters** than registers.
  - In such cases, the **parameters** are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a **register** (see **Figure 2.7**).
  - **Linux** uses a combination of these two approaches.

System Calls

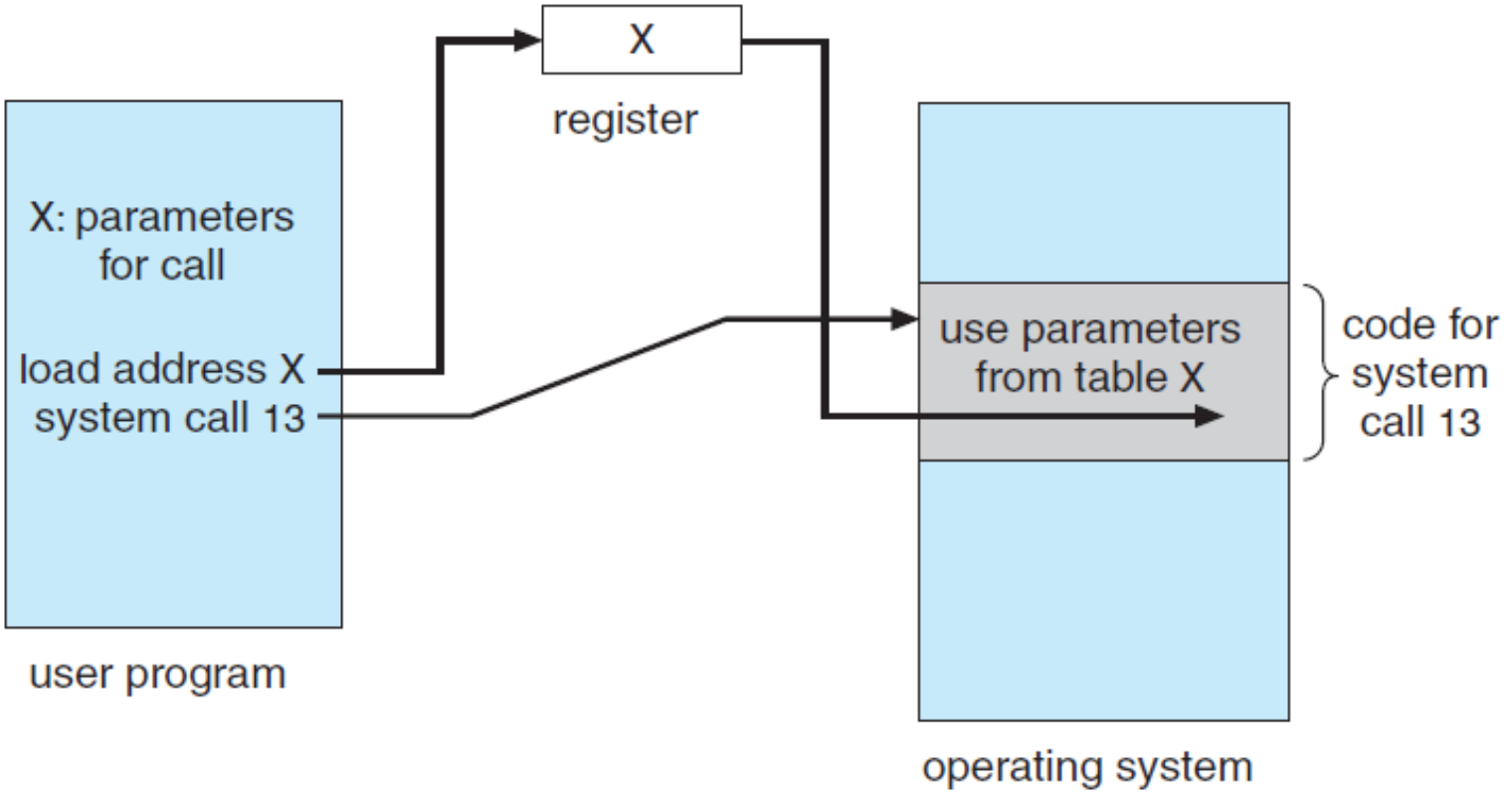


Figure 2.7 Passing of parameters as a table.

Types of System Calls

- Process control
  - create process, terminate process
  - load, execute
  - get process attributes, set process attributes
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices
- Protection
  - get file permissions
  - set file permissions

Figure 2.8 Types of system calls.

Types of System Calls

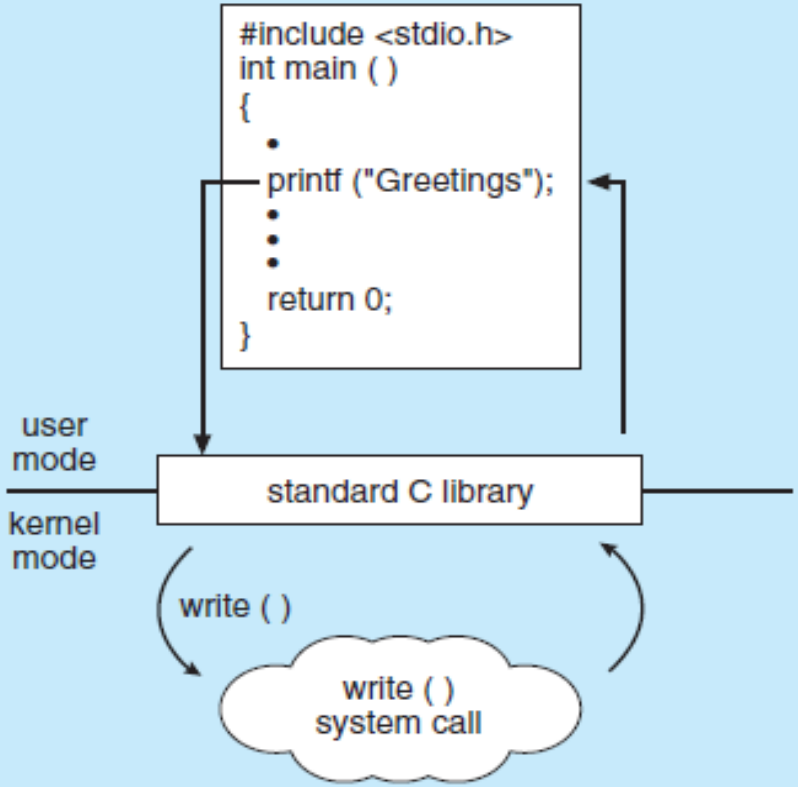
EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shm_open()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

process, control – System Calls

EXAMPLE OF STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program. This is shown below:



process, control – System Calls

- In a computer system, it is quite often, two or more processes may **share data**.
- To ensure the integrity of the data being shared, an OS often provide **system calls** allowing a process to **lock** shared data.
  - Then, no other process can access the data until the **lock** is **released.**
- Typically, such system calls include **acquire lock()** and **release lock().**

process, control – System Calls

- The **MS-DOS** is an example of a **single-tasking system**.
- It has a **command interpreter** that is invoked when the computer is started (**Figure 2.9(a)**).
  - Because MS-DOS is single-tasking, it uses a simple method to run a program and does not create a new process.
- It **loads** the program into **main memory**, writing over most of itself to give the program as much memory as possible (**Figure 2.9(b)**).

process, control – System Calls

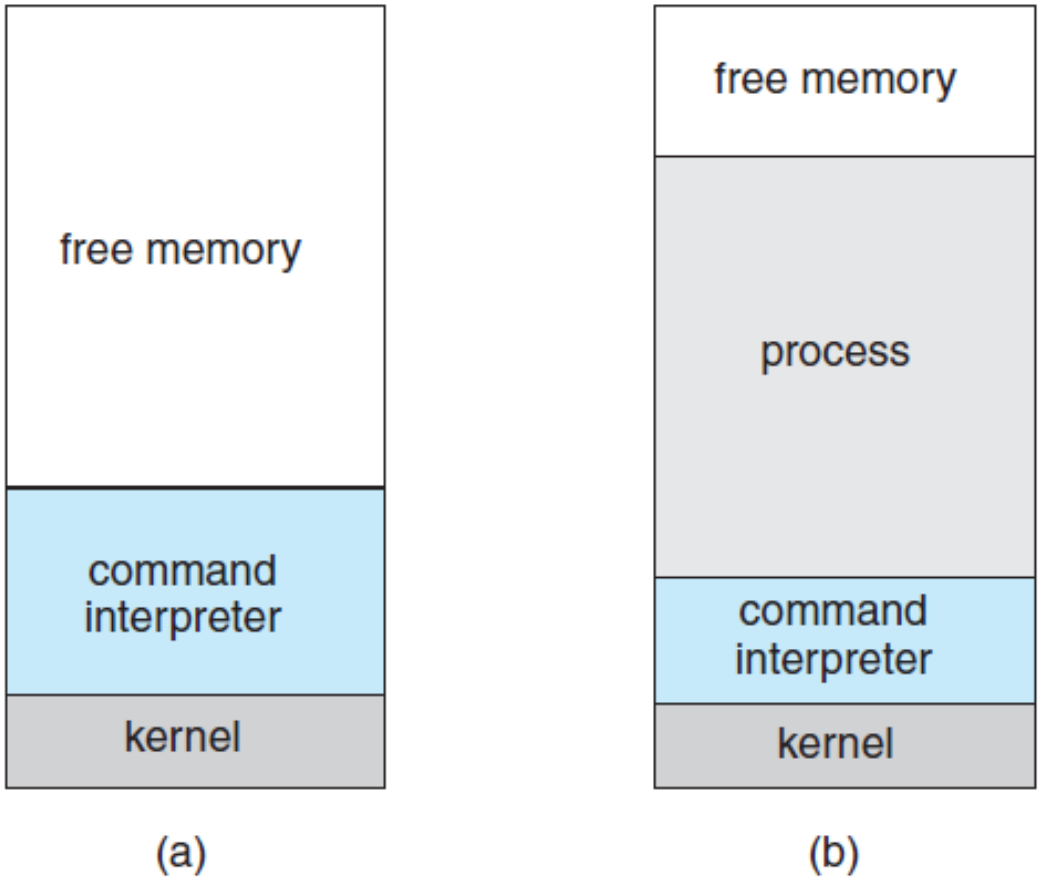


Figure 2.9 MS-DOS execution. (a) At system startup. (b) Running a program.

process, control – System Calls

- Next, it sets the **instruction pointer (**also called a **program counter)** to the first instruction of the program.
- The program then runs, and either an error causes a ***trap***, or the program executes a **system call** to terminate.

process, control – System Calls (8)

- **FreeBSD OS** (derived from Berkeley UNIX) is an example of a **multitasking** operating system.
  - When a user logs on to the system, the shell of the user's choice is run.
  - This shell is similar to the **MS-DOS shell** in that it accepts commands and executes programs that the user requests.
  - However, since **FreeBSD** is a multitasking system, the command interpreter may continue running while another program is executed (**Figure 2.10**).

process, control – System Calls

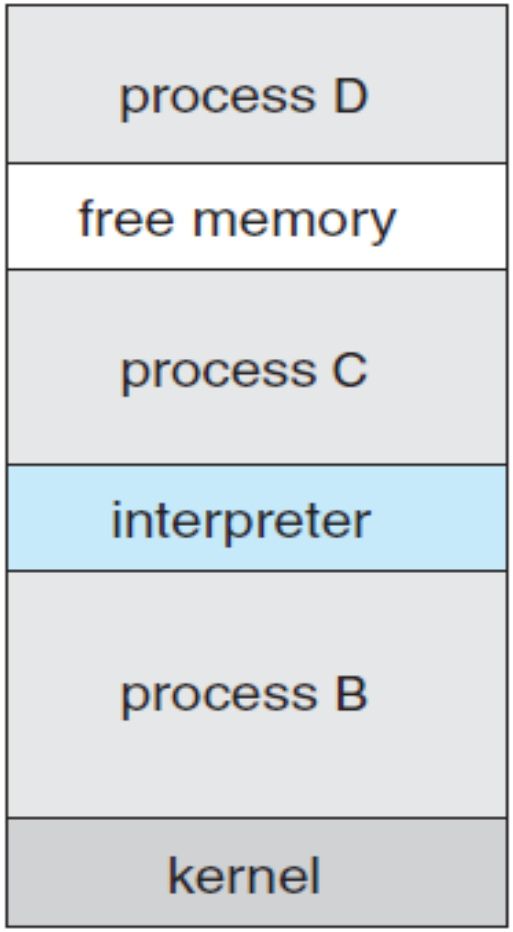


Figure 2.10 FreeBSD running multiple programs.

File Management – System Calls

- Several **system calls** dealing with files:
  - We first need to be able to **create()** file.
  - Once the file is created, we need to **open()** it and to use it.
  - We may also **read(), write(),** or **reposition()** (rewind or skip to the end of the file, for example).
  - Finally, we need to **close()** the file, indicating that we are no longer using it.
  - Later can **delete()** the selected file.

Device Management – System Calls

- A process may need several resources to execute—*main memory*, *disk drives*, *access to files*, and so on.
  - If the resources are available, they can be granted, and control can be returned to the user process.
  - Otherwise, the process will have to **wait** (called **blocked**) until resources are available.
- A system with multiple users may require us to first **request()** a device, to ensure exclusive use of it.
- After we are finished with the device, we **release()** it.

Information Maintenance

- Many **system calls** exist simply for the purpose of transferring information between the *user program* and the OS:
  - For example, most systems have a system call to return the current **time()** and **date().**
  - Other system calls may return information about the system, such as the *number of current users*, *the version number* of the OS, the *amount of free memory* or *disk space*, and so on.

Communication

- There are two common models of **inter-process communication (IPC)**:
  - the **message passing** model and
  - the **shared-memory** model.

Message-passing model (1)

- In the **message-passing model,** the communicating processes exchange messages with one another to transfer information.
- **Messages** can be exchanged between the processes either directly or indirectly through a **common mailbox**.
- The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network.

Message-passing model (2)

- Each computer in a network has a **host name** and a **network identifier**, such as an **IP address**.
- Similarly, each process has a **process ID**.
- The **get_hostid()** and **get_processid()** system calls do this translation.
- The network identifiers are then use the **open()** and **close()** calls provided by the file system or to specific **open_ connection()** and **close_connection()** system calls, depending on the system's model.
- The recipient process usually must give its permission for communication to take place with an **accept_connection()** system call.
- The source, known as the **client,** and the receiver, known as a **server,** then exchange messages by using **read_message()** and **write_message()** system calls.
- At the end the **close_connection()** system call terminates the communication

Shared-memory Model

- In the **shared-memory model,** processes use **shared_memory_create()** and **shared_memory_attach()** system calls to create and gain access to regions of memory owned by other processes.
  - **The OS tries to prevent one process from accessing another process's memory.**
  - **Shared memory requires that two or more processes agree to remove this restriction.**
- They can then exchange information by reading and writing data in the shared areas.
- The form of the data is determined by the processes and is not under the operating system's control.

Pro and Cons

- Both of the models just discussed are common in operating systems:
  - **Message passing** is useful for exchanging smaller amounts of data, because no conflicts need be avoided.
    - It is also easier to implement than shared memory for *inter-computer* communication.
  - **Shared memory** allows maximum speed and convenience of communication, since it can be done at memory transfer speeds when it takes place within a computer.
    - Problems exist are, in the areas of protection and synchronization between the processes sharing memory.

System Programs (1)

- Another aspect of a modern system is its collection of **system programs**.
- **System programs,** also known as **system utilities,** provide a convenient environment for *program development* and *execution.*
- Some of the **System programs** are simply user interfaces to **system calls**.
  - Others are considerably more complex.

System Programs (2)

- **System programs** can be divided into these categories:
  - **File management**: These programs *create*, *delete, copy, rename*, *print*, *dump*, *list*, and generally *manipulate files and directories*.
  - **Status information**: Some programs simply ask the system for the *date, time, amount of available memory or disk space, number of users, or similar status information*.
  - **File modification:** Several text editors may be available to *create and modify the content of files stored on disk or other storage devices*.
  - **Programming-language support**: Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, Python, PERL, etc) are often provided with the OS or available as separate download form.

System Programs (3)

- **Program loading and execution:** Once a program is assembled or compiled, it must be loaded into memory to be executed.
- **Communications**: These programs provide the mechanism for creating virtual connections among processes, users, and computer systems.
- **Background services**: All general-purpose systems have methods for launching certain system-program processes at boot time.
  - Some of these processes terminate after completing their tasks, while others continue to run until the system is halted.

- The view of the OS seen by most users is defined by the application and system programs, rather than by the actual **system calls**
  - Consider a user is running the Mac OS X on his/her PC, the user might see its GUI, alternatively, the user might have a command-line UNIX shell. Both **GUI** and **command line** use the same set of **system calls**, but the system calls look different and act in different ways.

OS Design Goals

- The first problem in designing an OS is to define **goals** and **specifications**:
  - At the **highest level**, the design of the OS will be affected by the following choices of a system:
    - batch,
    - time sharing,
    - single user,
    - multiuser,
    - distributed,
    - real time, or
    - general purpose.
- The requirements can be divided into two basic groups:
  - **user goals** and **system goals**.

Design Goals - User goals

- **Users goal** of an OS can be based on the following aspects:
  - The OS should be *convenient to use*, *easy to learn and to use*, *reliable*, *safe*, and *fast*.
    - These specifications are not particularly useful in the system design, since there is no general agreement on how to achieve them.

Design Goals - System goals

- **System goal** of an OS can be based on the following aspects:
  - The system should be *easy to design*, *implement*, *and maintain*; and it should be *flexible*, *reliable*, *error free, and efficient*.
    - Again, these requirements are vague and may be interpreted in various ways.

OS Mechanisms and Policies

- One important principle of OS design is the separation of **policy** from **mechanism.**
  - **Mechanisms** determine ***how*** to do something;
  - **Policies** determine ***what*** will be done.
    - For example, **the timer construct is a mechanism for ensuring CPU protection** but deciding **how long the timer is to be set for a particular user is a policy decision**.
- The separation of **policy** and **mechanism** is important for OS design flexibility.
  - **Microkernel-based** operating systems take the separation of **mechanism** and **policy** by implementing a set of basic **functional blocks**:

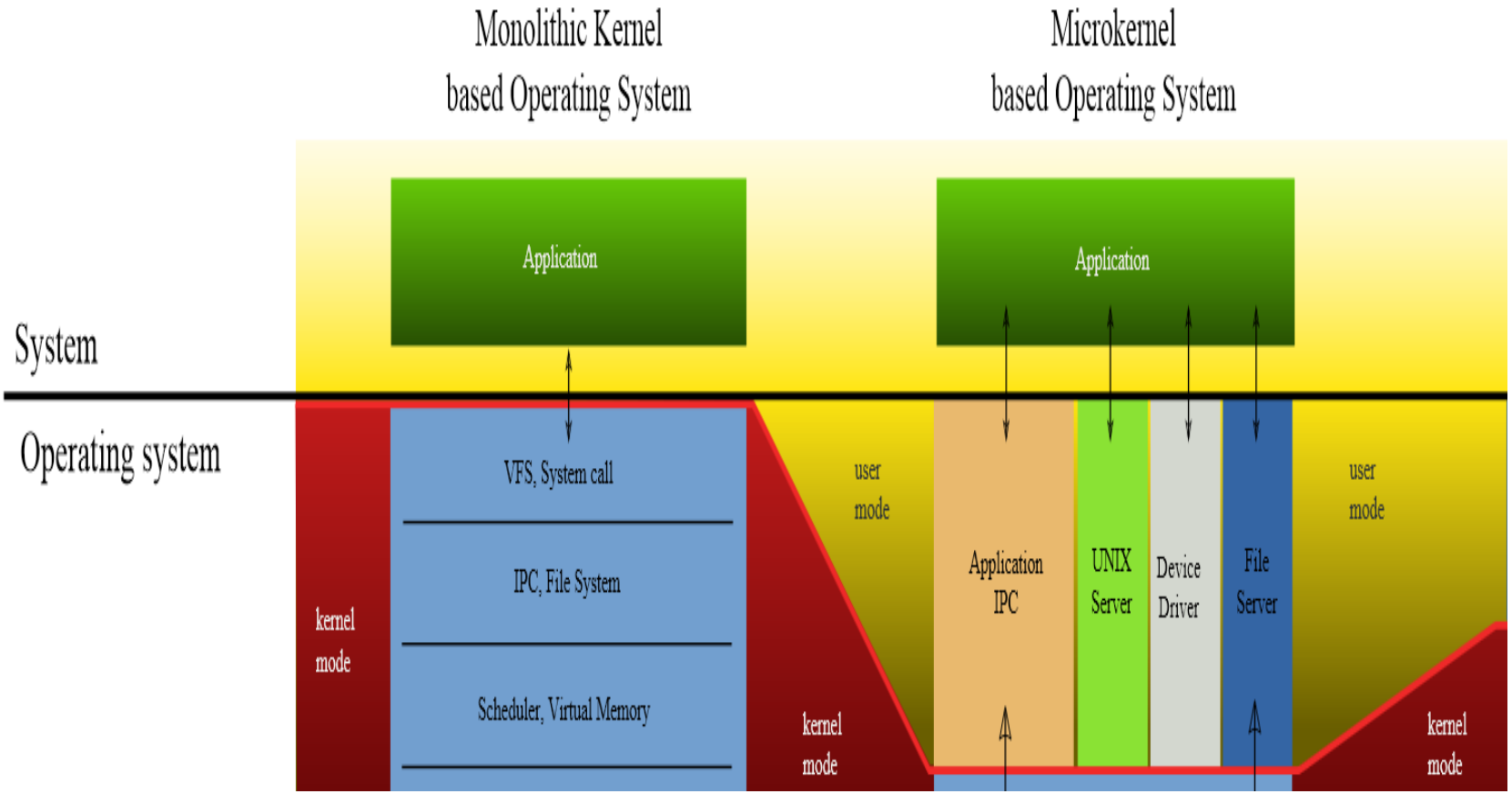
OS Mechanisms and Policies

- These **micro-kernel functional blocks** are almost **policy-free**
  - **because** allowing more advanced mechanisms and policies to be added via **user-created kernel modules** or user programs themselves.
- **Microsoft** has closely encoded both **mechanism** and **policy** into the system to enforce a global look and feel across all devices that run the **Windows**.
  - **Windows** have similar interfaces to all applications, because the interface itself is built into the **kernel** and **system libraries**.
  - **Apple** has adopted a similar strategy with its **macOS** and **iOS** systems.
- The "standard" **Linux kernel** has a specific **CPU scheduling algorithm**
  - which is a **mechanism** that supports a **certain policy**; anyone is free to modify or replace the **scheduler to support a different policy**.

OS Mechanisms and Policies

- **Policy decisions** are important for all **resource allocation**
  - Whenever it is necessary to decide whether to **allocate a resource**, a **policy decision** must be made.
  - Whenever the question is ***how*** rather than ***what***, it is a **mechanism** that must be determined.
- Once an OS is designed, it must be implemented.
  - Because **operating systems** are collections of many programs, written by many people over a long period, it isn't easy to make general statements about how they are implemented.
  - Early operating systems were written in **assembly language**.
  - Now, most are written in higher-level languages such as **C** or **C++**

OS Mechanisms and Policies



Monolithic OS Structure

- The simplest structure for organizing an OS is no structure at all.
  - That is, place all the **kernel's functionality into a single, static binary file** that runs in a single address space.
- This approach—known as a **monolithic** structure—is a common technique for designing operating systems.
  - An example of such **monolithic structuring** is the original **UNIX** operating system, which consists of two separable parts: the **kernel** and the **system programs**.
  - The kernel is further separated into a series of **interfaces** and **device drivers**, which have been added and expanded over the years as UNIX has evolved, as shown in **Figure 2.12**.

Monolithic OS Structure

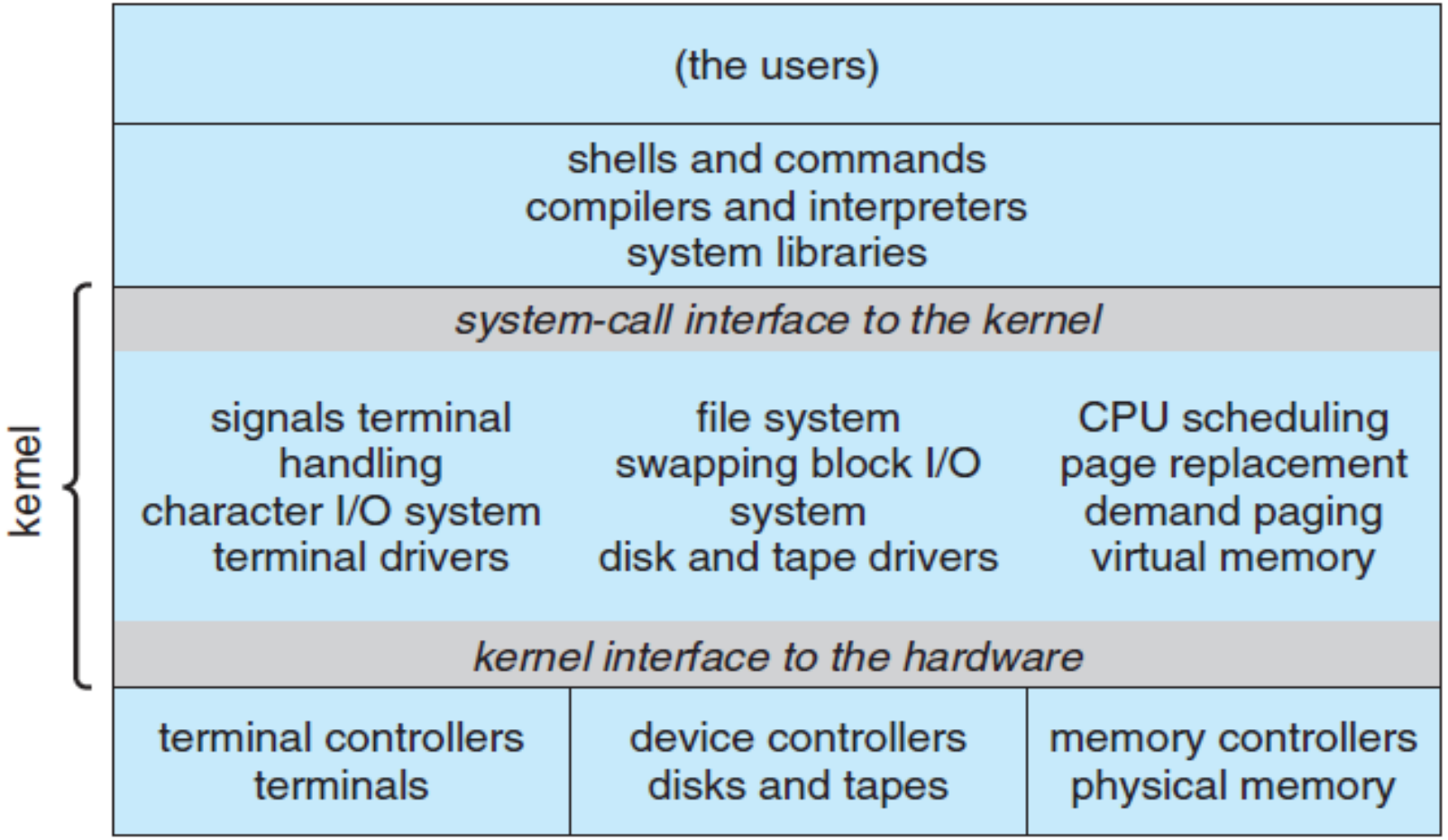


Figure 2.12 Traditional UNIX system structure.

Monolithic OS Structure

- The **Linux operating system** is based on **UNIX** and is structured similarly, as shown in **Figure 2.13**.
- Applications typically use the **glibc standard C library** when communicating with the **system call interface** to the **kernel**.
- The **Linux kernel is monolithic** in that it runs entirely **in kernel mode** in a single address space.
- Despite the apparent simplicity of monolithic kernels, they are difficult to implement and extend.
  - Monolithic kernels have a **distinct performance advantage**; there is very little overhead in the system-call interface, and communication within the kernel is fast.
  - Therefore, despite the implementation drawbacks of **monolithic kernels**, their speed and efficiency explain why such a design is used in the **UNIX**, **Linux**, and **Windows** operating systems.

Monolithic OS Structure

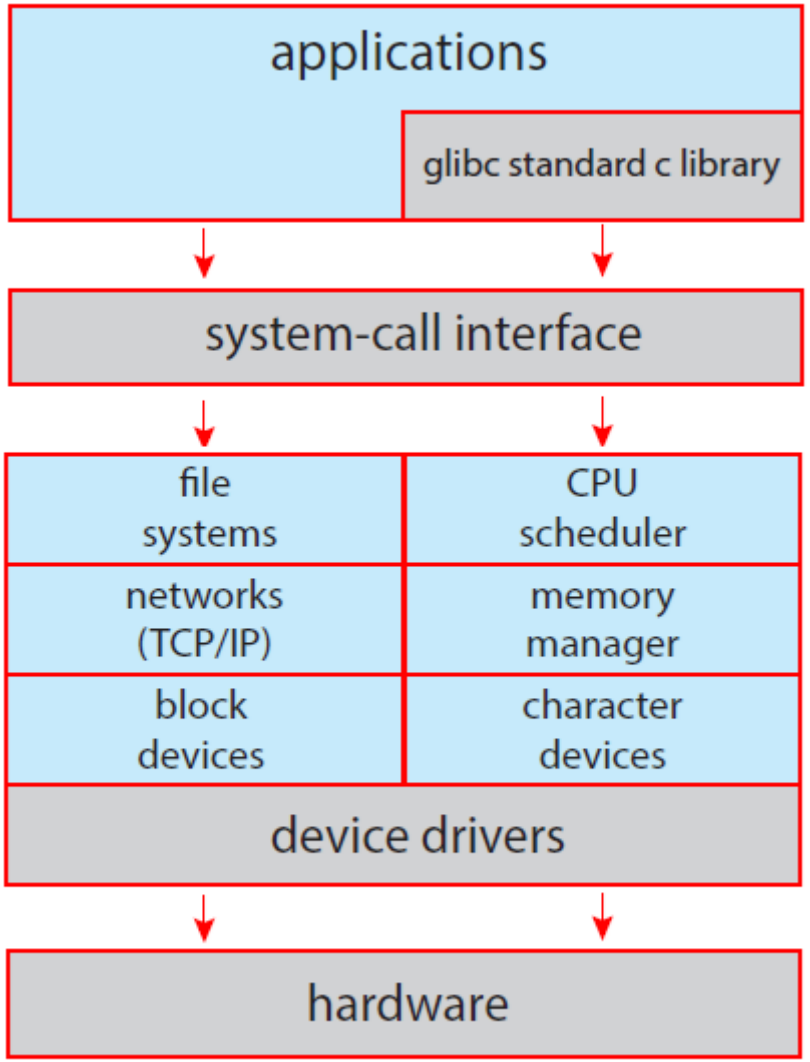


Figure 2.13 Linux system structure.

Layered Approach

- **The monolithic** design approach is often known as a **tightly coupled system** *because changes to one part of the system can have wide-ranging effects on other parts*.
- The **alternative OS design approach** is modular and is called a **loosely coupled** system.
  - Such a system is divided into separate, smaller components that have specific and limited functionality.
  - All these components together comprise the **kernel**.
- The **advantage of this modular approach** is that:
  - changes in one component affect only that component, allowing system implementers more freedom in creating and changing the inner workings of the system.
- One method is the **layered approach**, in which the OS is broken into several layers (levels). The bottom layer (**layer 0**) is the **hardware**, and the **highest layer** (**layer *N***) is the **user interface**. This layering structure is depicted in **Figure 2.14.**

Layered Approach

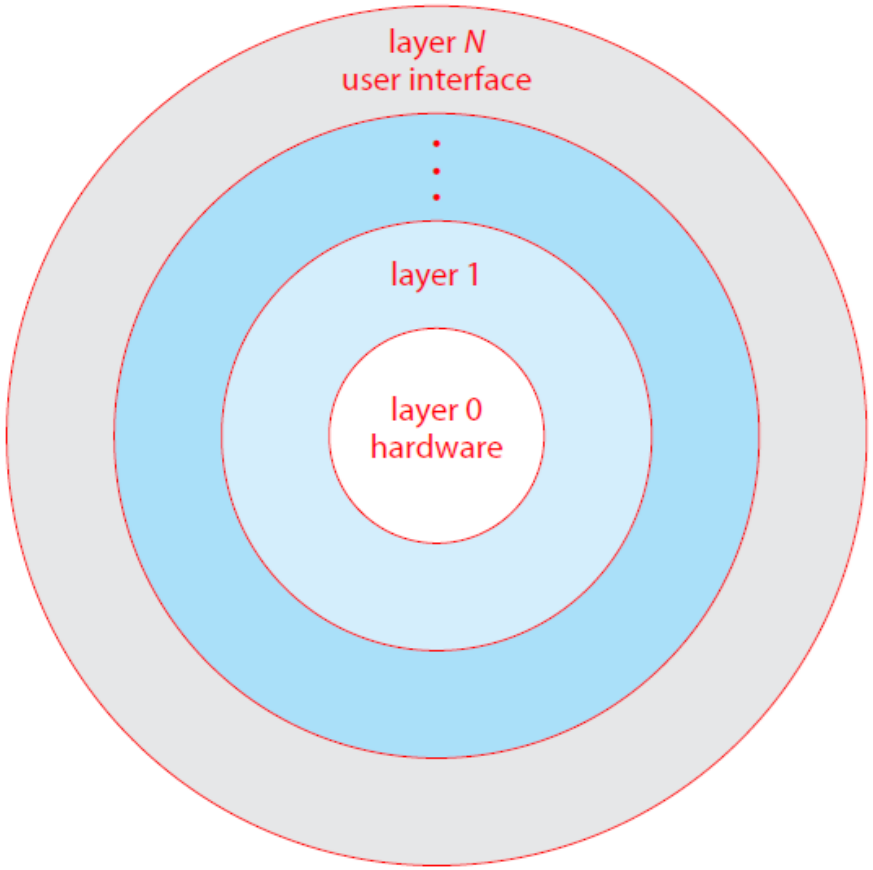


Figure 2.14 A layered operating system.

Layered Approach

- The **main advantage of the layered approach** is simplicity of construction and debugging.
  - The layers are selected so that each uses functions (operations) and services of only lower-level layers.
  - This approach simplifies debugging and system verification.
  - The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions.
  - **Layered systems** have been successfully used in computer networks (such as **TCP/IP**) and web applications.
- A few operating systems use a pure layered approach.
  - One reason involves *the challenges of appropriately defining the functionality of each layer*. The overall performance of such systems is poor due to the overhead of requiring a user program to traverse through multiple layers to obtain an operating-system service.

Micro-kernels

- In the **mid-1980s**, researchers at **Carnegie Mellon University** developed an OS called **Mach** that modularized the kernel using the **microkernel** approach.
  - This design approach structures the OS by removing all nonessential components from the kernel and implementing them as **user-level programs** that reside in separate address spaces.
  - The result is a **smaller kernel**.
  - Typically, **microkernels** provide minimal process and memory management, in addition to a communication facility.
  - **Figure 2.15** illustrates the architecture of a **typical microkernel**.

Micro-kernels

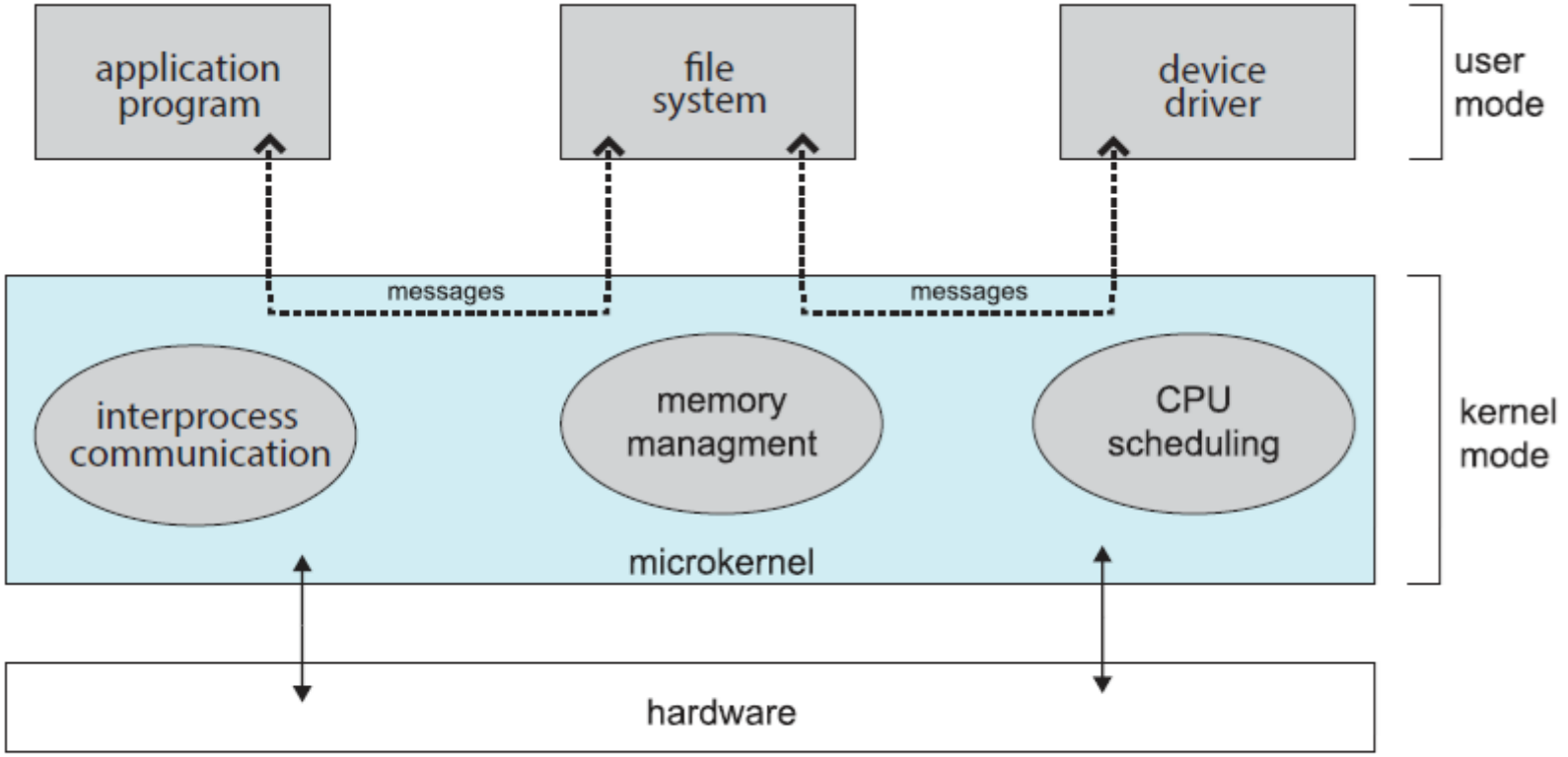


Figure 2.15 Architecture of a typical microkernel.

Micro-kernels

- The main function of the **microkernel** is to provide communication between the **client program** and the **various services** that are also running in the **user space**.
- Communication is provided through **message passing:**
  - For example, *if the **client program** wishes to access a file, it must interact with the **file server***.
  - *The **client program** and service never interact directly*.
  - *Rather, they communicate indirectly by **exchanging messages with the microkernel***.
- **One benefit of the microkernel approach** is that it makes extending the OS easier
  - All new services are added to the **user space** and consequently do not require modification of the **kernel**.

Micro-kernels

- The **microkernel** also provides more security and reliability, since most services are running as user applications—rather than kernel processes.
- **If a service fails, the rest of the OS remains untouched**.

- The best-known illustration of a **microkernel OS** is ***Darwin***, the kernel component of the **macOS** and **iOS** operating systems.
- Another example of microkernel design is **QNX**, a real-time operating system for **embedded systems**.
- **Drawback:** The performance of **microkernels** can suffer due to increased **system-function overhead**.
  - When two user-level services communicate, messages must be copied between the services, which reside in separate address spaces.
  - The OS may have to switch from one process to the next to exchange the messages, which causes kernel overhead.
- **Windows NT**: The first release had a layered microkernel organization.
- **Windows XP** was more **monolithic** than a **microkernel**.

Modules

- Perhaps the best current methodology for OS design involves using **loadable kernel modules** (**LKMs**).
  - The **kernel** has a set of core components and can link in **additional services** via modules, either at **boot time** or **during run time**.
  - This type of design is common in modern implementations of **UNIX**, such as **Linux**, **macOS**, Solaris, and **Windows**.
- The idea of **LKM design** is for the kernel to provide **core services**, while **other services are implemented dynamically**, **as the kernel is running**.
  - **Linking services dynamically** is preferable to adding new features directly to the kernel, *which would require recompiling the kernel every time a change is made*.
  - For example, we add new CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules.

Modules

- The **overall result of LKM** resembles a **layered system** in that each kernel section has defined, protected interfaces.
  - Still, **LKM is more flexible than a layered system**, because any module can call any other module.
- The **LKM approach** is also like the **microkernel approach**
  - In that the **primary module has only core functions** and knowledge of how to load and communicate with other modules.
- But LKM is more efficient, *because modules do not need to invoke message passing to communicate*.
- **Linux** uses **LKMs**, primarily for supporting *device drivers* and *file systems*.
  - **LKMs** can be “inserted” into the kernel as the system is started (or ***booted***) or during run time.
  - If the **Linux kernel** does not have the necessary **driver**, it can be dynamically loaded. LKMs can also be removed from the kernel during runtime.

Modules

- The **Solaris OS** structure, shown in **Figure 2.16**, is organized around a core kernel with **seven types of loadable kernel modules**:

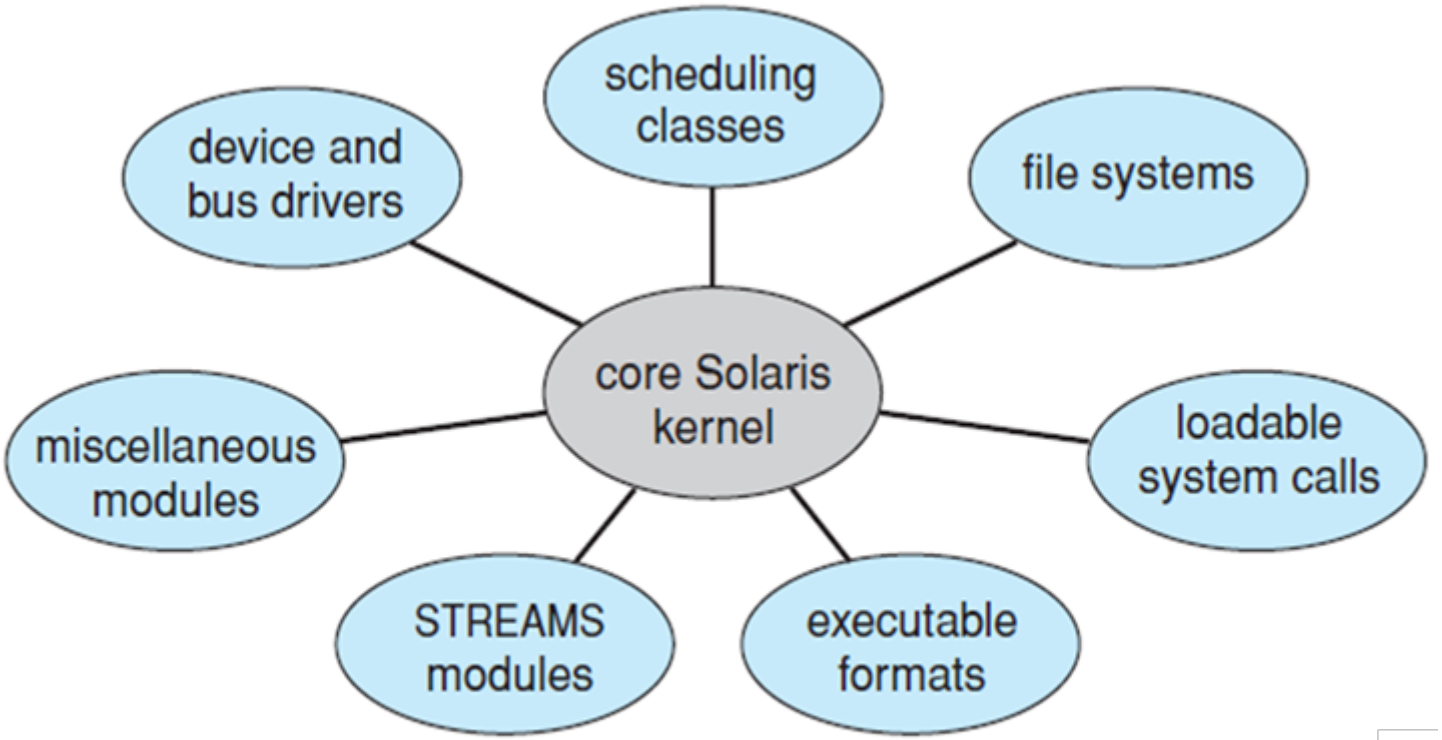


Figure 2.16 Solaris loadable modules.

System Boot

- After an OS is generated, it must be made available for use by the hardware.
- But how does the hardware know where the kernel is or how to load that kernel? *The process of starting a computer by loading the kernel is known as **booting** the system*.
- On most systems, the **boot process** proceeds as follows:
  1. A small piece of code known as the **bootstrap program** or **boot loader** locates the **kernel**.
  2. The **kernel** is loaded into main memory and started.
  3. The **kernel** initializes hardware.
  4. The **root file** system is mounted.

System Boot

- Some computer systems use a multistage boot process: When the computer is first powered on, a small boot loader located in nonvolatile firmware known as **BIOS** is run.
- This initial boot loader usually does nothing more than load a second boot loader, which is located at a fixed disk location called the **boot block**.
- The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution.
- More typically, it is simple code (as it must fit in a single disk block) and knows only the address on disk and the length of the remainder of the bootstrap program.

System Boot

- Many recent computer systems have replaced the **BIOS-based boot process** with **UEFI** (Unified Extensible Firmware Interface).
- **UEFI** has several advantages over **BIOS**, including better support for 64-bit systems and larger hard disk drives.
- Perhaps the greatest advantage is that **UEFI** is a single, complete boot manager and therefore, is faster than the multistage BIOS boot process.

System Boot

- In addition to loading the file containing the kernel program into memory, it also runs diagnostics to determine the state of the machine —for example*, inspecting memory* and the *CPU*, and *discovering devices*.
- If the **diagnostics pass**, the program can continue with the **booting steps**.
- The **bootstrap** can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory.
- Sooner or later, it starts the operating system and mounts the root file system. It is only at this point that the system is said to be **running**.