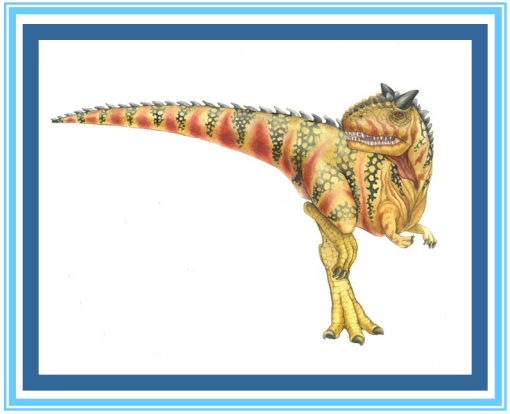
Chapter 2: Operating-System Structures



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 permission management.
- **Communications** – Processes may exchange information on the same computer or between computers over a network
- **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 running concurrently, resources must be allocated to each of them
  - Many types of resources - *CPU cycles*, *main memory*, *file storage,* I/O *devices*, etc.
- **Accaccounting -** 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 from 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.

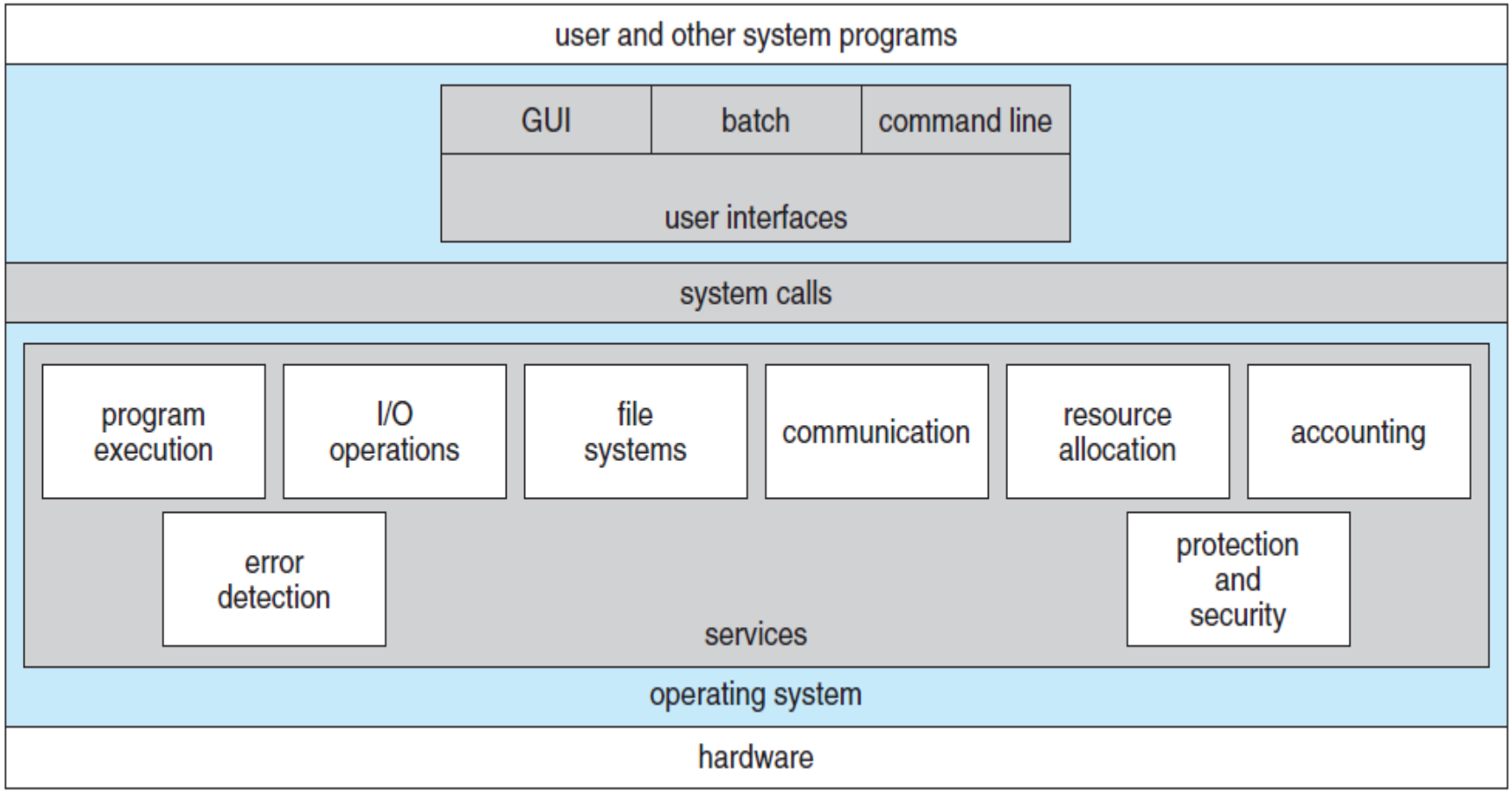


Figure 2.1 A view of operating system services.

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 command line by DOS, Linux, etc
  - A **graphical user interface** (**GUI**)
    - For example Windows

Command Interpreters

- The main function of the **command line** is to get and execute the given user-specified command.
  - Many of the commands given at this level manipulate files:
    - *create*, *delete, list*, *print*, *copy*, *execute*, and so on.
  - The MS-DOS and UNIX shells operate in this way.
- These commands can be implemented in two general ways:
  - **One approach** is the **command interpreter** or **command line** 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**.

Command Interpreters

- **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**

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, 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 the MS-DOS.

Choice of Interface

- The choice of whether to use a **command-line** or **GUI** interface is mostly one of personal preference.
  - 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- **shell programming**.
  - 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.
- Most **Windows** users are happy to use the **Windows GUI** environment.
  - .

System Calls (1)

- **System calls** provide an interface to the services made available by an OS.
- Let us see how system calls are used:
  - Assume a simple program to read data from **one file** and copy them to **another file** (the output file is not existing)
  - The first name 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 (2)

- 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 (3)

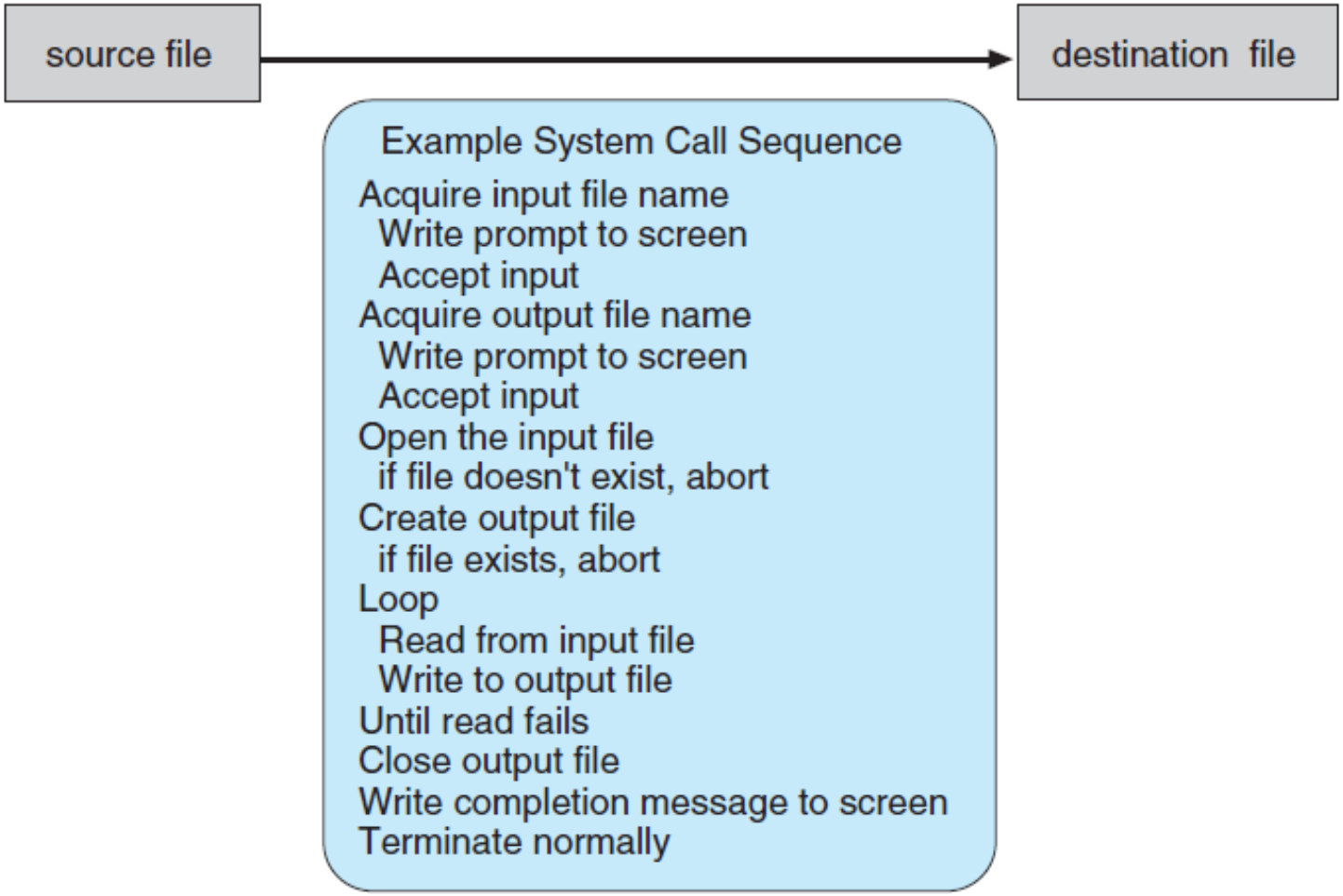


Figure 2.5 Example of how system calls are used.

System Calls (4)

- For most programming languages, there is a **run-time support system** (*a set of functions built into libraries included with a compiler*) provides a **system-call interface** that serves **system calls**:
  - The **system-call interface** intercepts function calls in the **API** (**Application Programming Interface**) and invokes the necessary **system calls** within the OS.
  - 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.
    - Typically, a **number** is associated with each **system call**, and the **system-call interface** maintains a **table indexed** according to these numbers.

System Calls (5)

- The **caller** (a *user program*) needs only obey the **API** and understand what the OS will do as a result of the execution of that **system call**.
  - Most of the details of the OS interface are hidden from the programmer by the API and are managed by the run-time **support library**.
- The relationship between 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 (6)

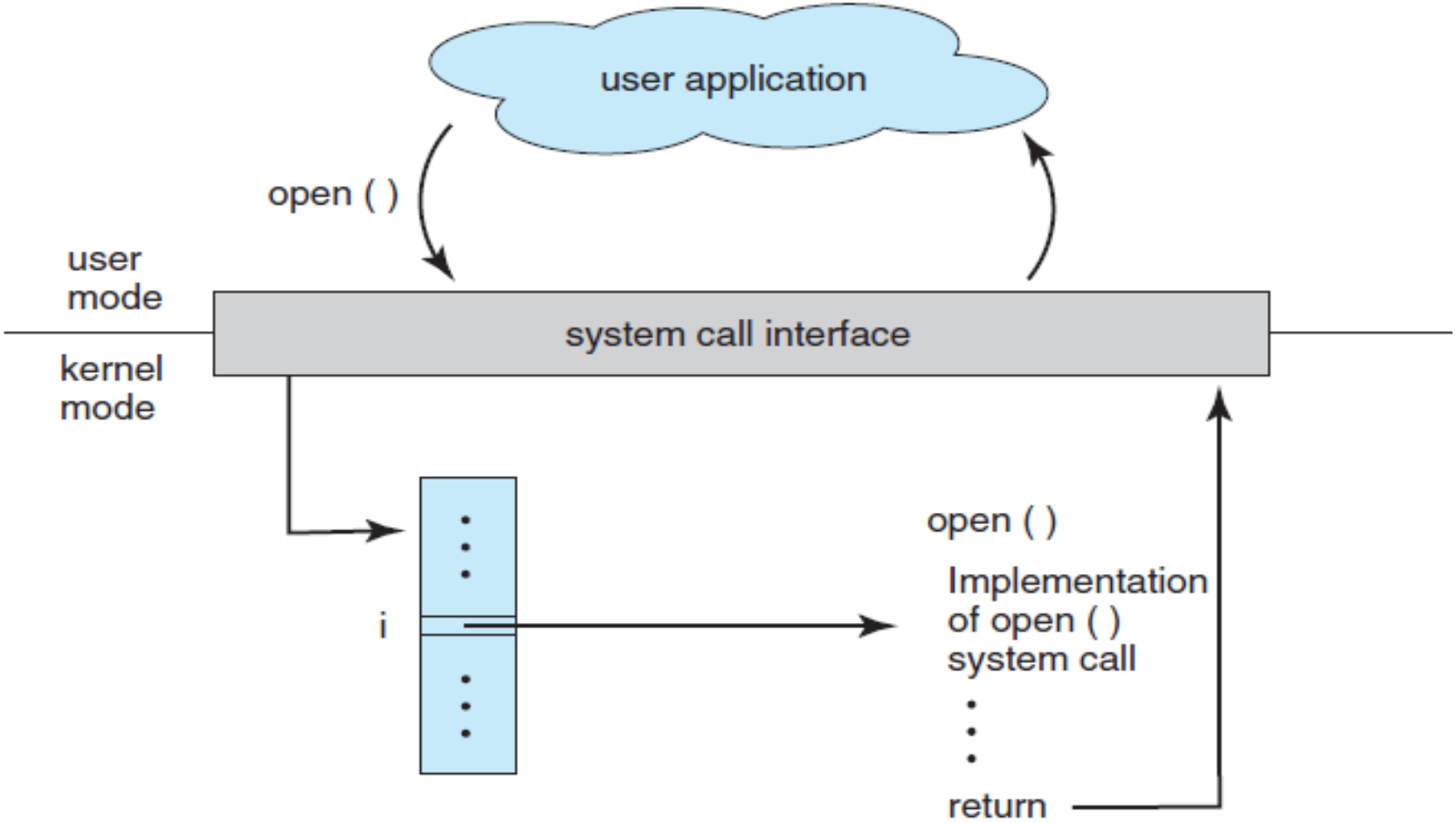


Figure 2.6 The handling of a user application invoking the `open()` system call.

System Calls (7)

- **Three** general methods used to **pass parameters** to the **OS** during a **system call** are:
  - The **simplest approach** is to pass the **parameters in registers** (**Figure 2.7**).
    - This is the approach taken by Linux and Solaris.
  - In **other cases**, the parameters also can be placed, or **pushed**, onto the **stack** by the program and **popped off** the stack by the OS.

System Calls (8)

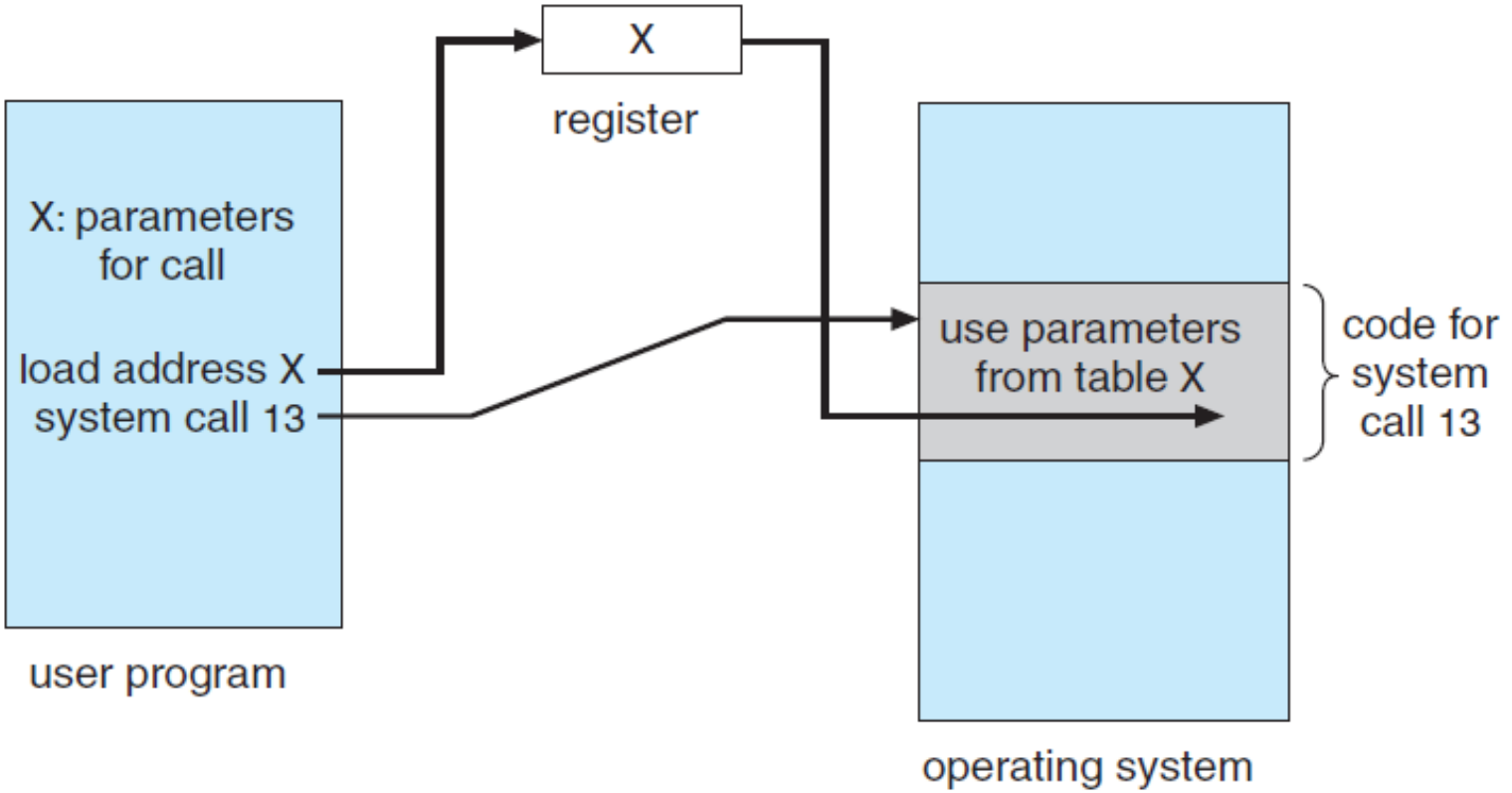


Figure 2.7 Passing of parameters as a table.

Types of System Calls

- **System calls** can be grouped roughly into **six** major categories:
  - process
  - control
  - file manipulation
  - device manipulation
  - information maintenance
  - communication
  - protection

Types of System Calls

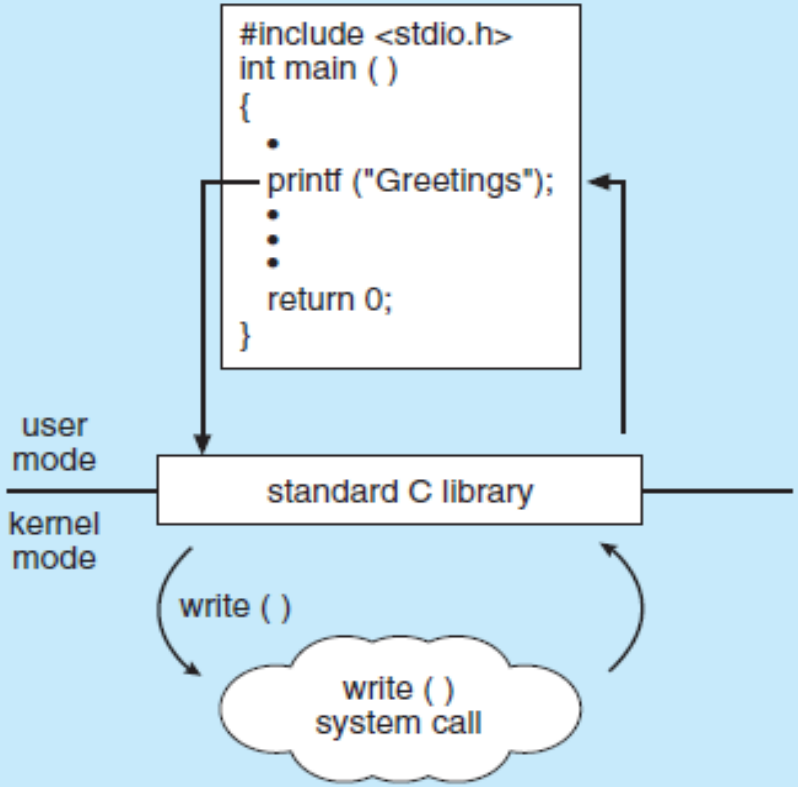
EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	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 (1)

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 (2)

- 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 (3)

- 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 (4)

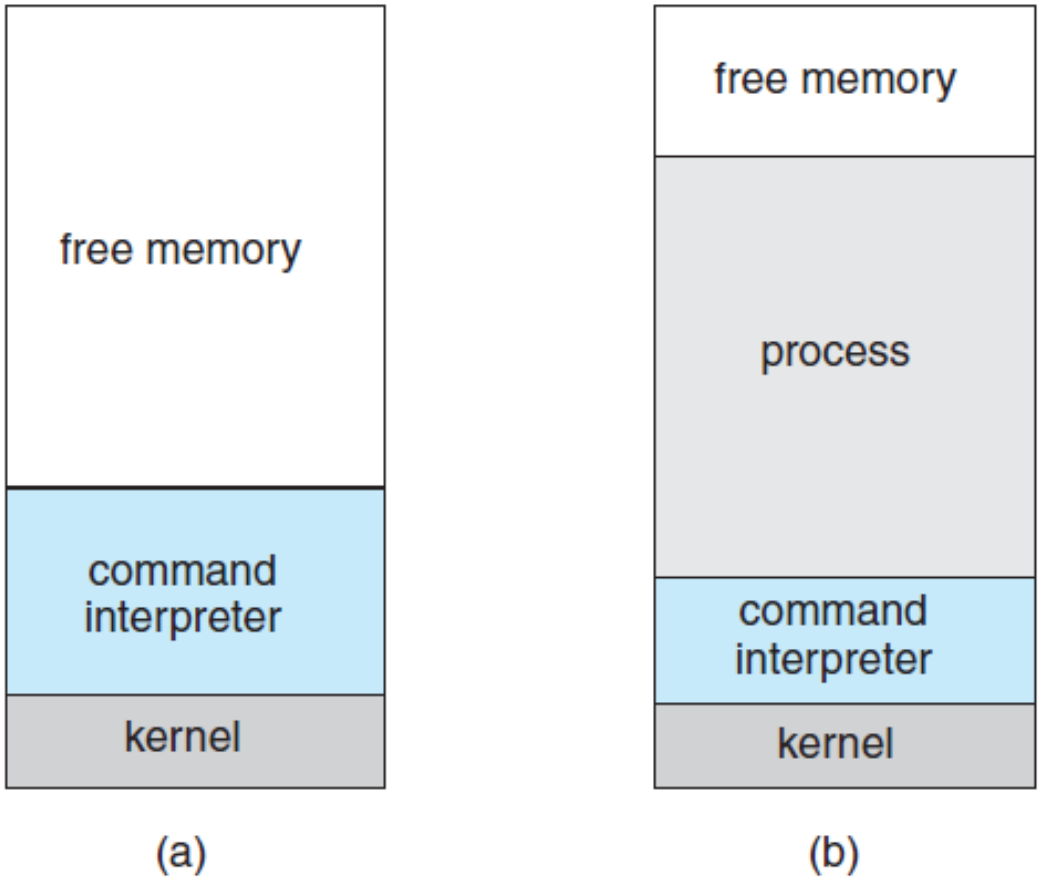


Figure 2.9 MS-DOS execution. (a) At system startup. (b) Running a program.

process, control – System Calls (5)

- 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 (9)

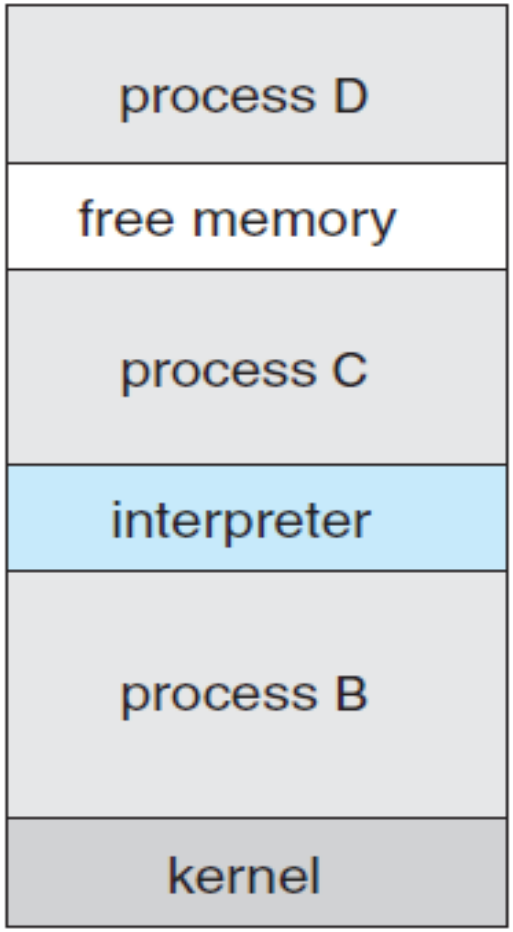


Figure 2.10 FreeBSD running multiple programs.

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.

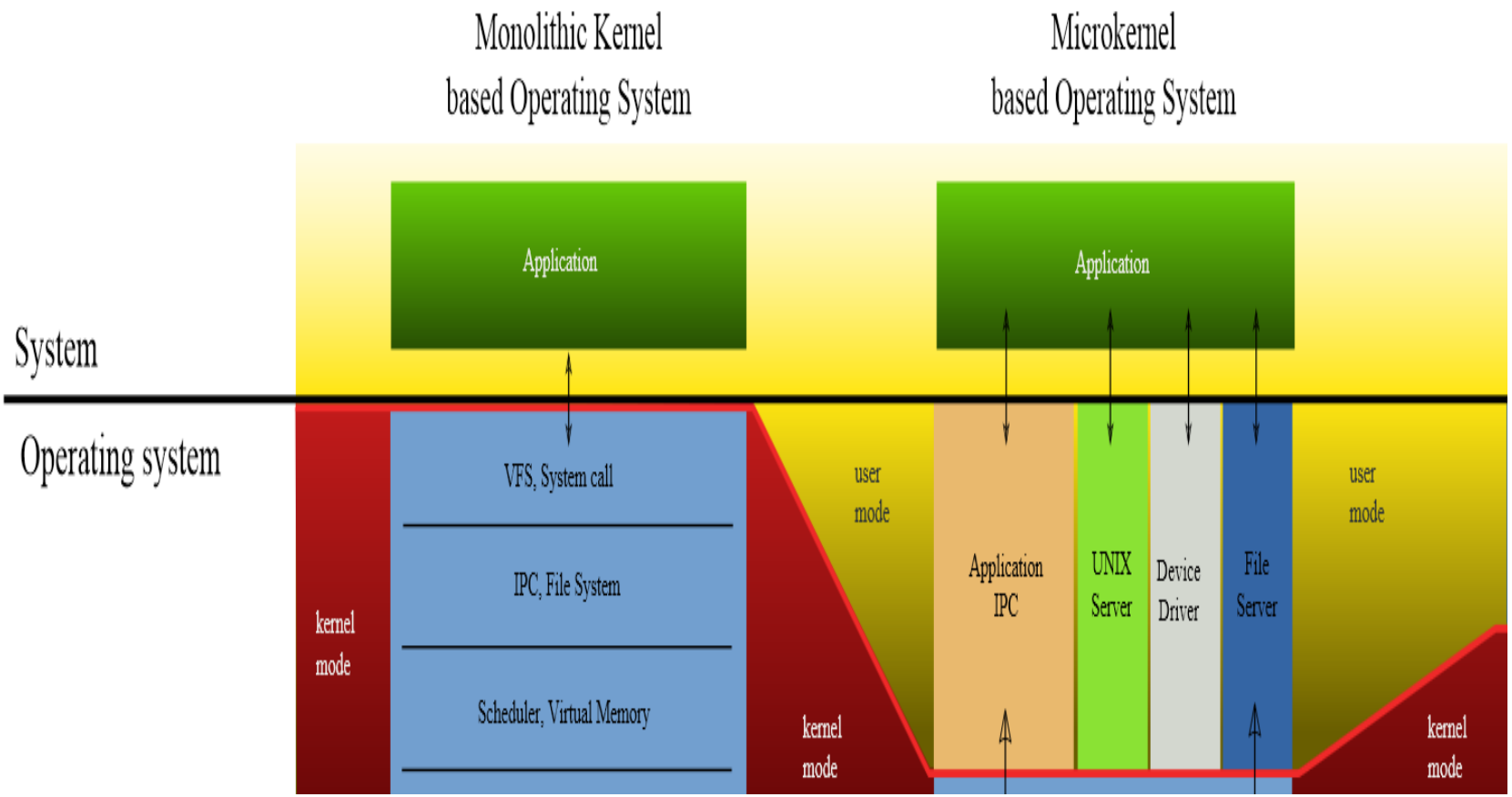
Mechanisms and Policies (1)

- 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.

Mechanisms and Policies (2)

- **Microkernel-based** operating systems take the separation of **mechanism** and **policy** by implementing a set of functional blocks:
  - **kernel** is a collection of **primitive facilities** or **system calls** for the rest of the OS
  - a **microkernel** is the near-minimum amount of software that can provide the **mechanisms** needed to implement an OS
  - The **microkernel** is the only software section of the OS running at the most privileged functions of the OS level (***supervisor*** or ***kernel mode***) at any given time.
- A **monolithic kernel** is an OS architecture where the *entire OS is working in the **kernel space** and alone as supervisor mode .*
  - Here the functional mechanisms and policies are not separated.

Mechanisms and Policies (3)



Mechanisms and Policies (4)

- In a **microkernel-based** OS, its function-blocks are almost **policy free**;
  - consider the history of UNIX. At first, it had a **time-sharing** scheduler.
- **Policy decisions** are important for all **resource allocation**.
  - Whenever it is necessary to decide whether or not 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.

Implementation

- Early operating systems were written in **assembly language**.
  - Now, although some operating systems are still written in *assembly language*, most are written in a higher-level language such as C or C++.
- Actually, an OS can be written in more than one language.
  - The *lowest levels of the kernel* might be assembly language.
  - Higher-level routines might be in C, and system programs might be in C or C++, in interpreted scripting languages like PERL or Python, or in shell scripts.

Operating-System Structure

- The **OS design structures** can be classified into **five** forms:
  - **Simple Structure**
  - **Layered Approach**
  - **Micro-kernels**
  - **Modules**
  - **Hybrid Systems**

Simple Structure (1)

- In **simple OS structure**, an OS does not have a well-defined structure.
  - **MS-DOS** is an example of such a system.
  - It was written to provide the most functionality in the least space, so it was not carefully divided into modules. **Figure 2.11** shows a **simple OS structure**.
  - In **MS-DOS**, the **interfaces** and **levels of functionality** are not well separated (**monolithic in nature**).
  - **Application programs** are able to access the basic **I/O routines** to write directly to the display and disk drives.
    - Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail.

Simple Structure (2)

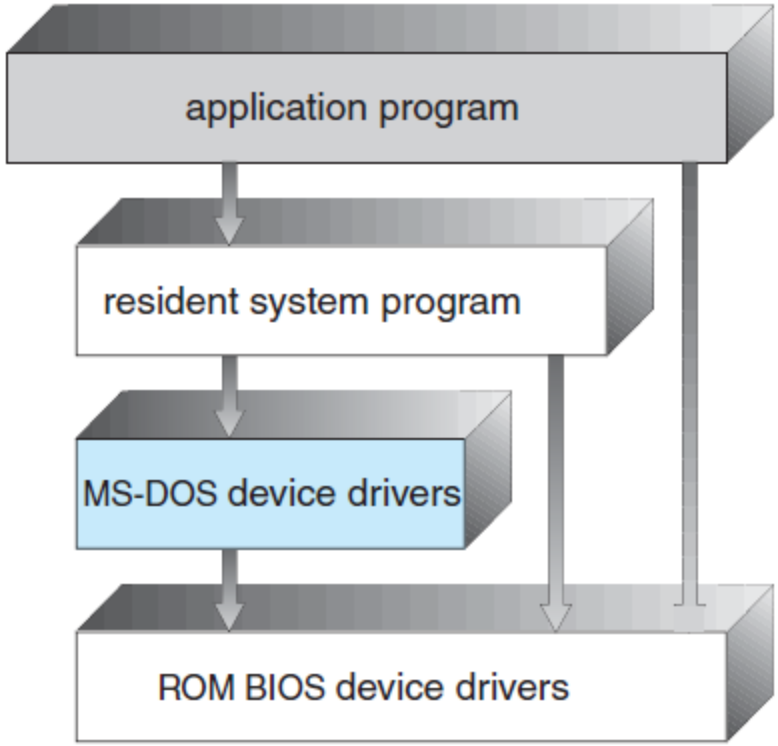


Figure 2.11 MS-DOS layer structure.

Simple Structure (3)

- Another example of simple structuring is the **original UNIX** system.
  - Like MS-DOS, UNIX initially was limited by hardware functionality.
  - It consists of two separable parts:
    - the **kernel** and the **system programs**.
- We can view the **traditional UNIX OS** (**monolithic design**) as being layered to some extent, as shown in **Figure 2.12**.
  - Everything below the **system-call** interface and above the physical hardware is the kernel.
  - The **kernel** provides the *file system*, *CPU scheduling, memory management*, and other OS functions through system calls.
- It had a distinct performance advantage, there is very little overhead in the system call interface or in communication within the kernel.

Simple Structure (4)

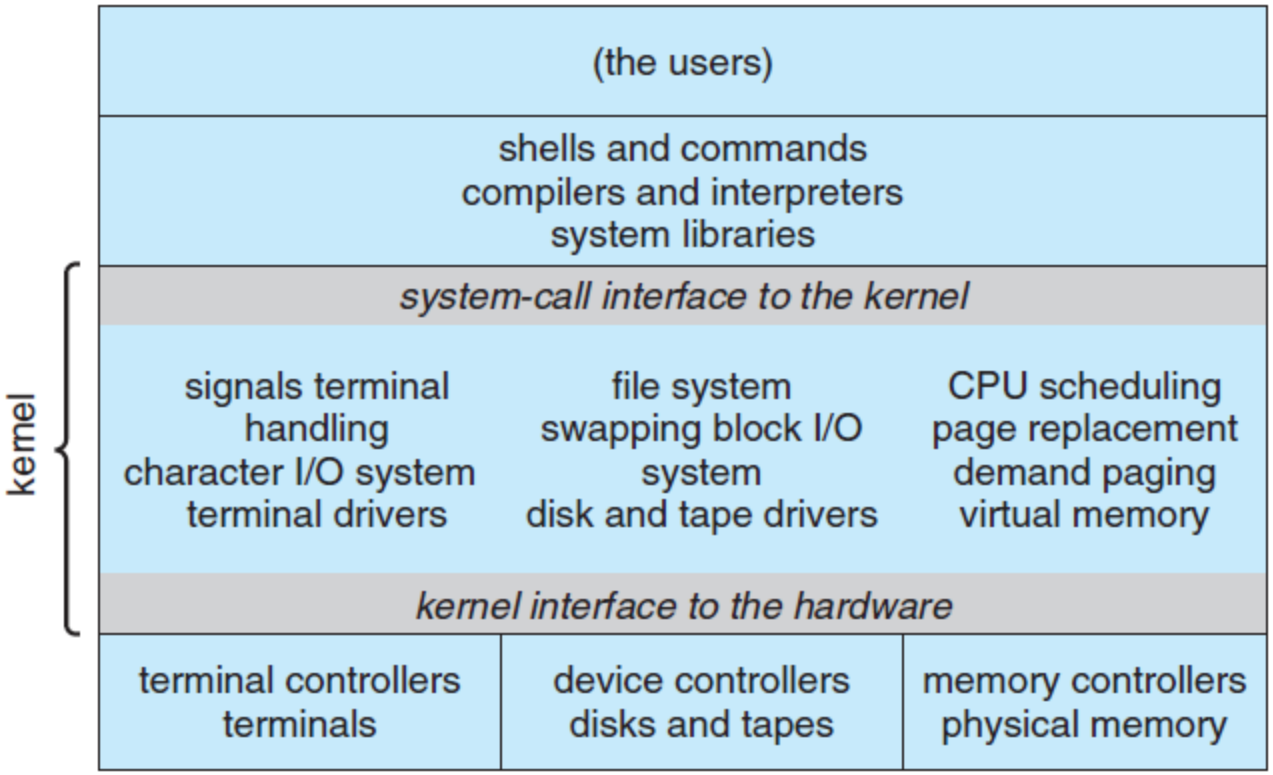


Figure 2.12 Traditional UNIX system structure.

Layered Approach (1)

- In **layered approach**, an OS can be made **modular** in many ways:
  - One method is the **layered** approach, in which the OS is broken into a number of **layers** (called levels).
  - The bottom layer (layer 0) is the **hardware**; the highest (layer *N) is the* **user interface** (**Figure 2.13)**.
- A typical OS layer—say, layer *M* —consists of data structures and a set of routines that can be invoked by higher-level layers.
  - The **main advantage** of the layered approach is simplicity of construction and debugging.
  - **Disadvantage**: Each layer adds overhead to the **system call**.

Layered Approach (2)

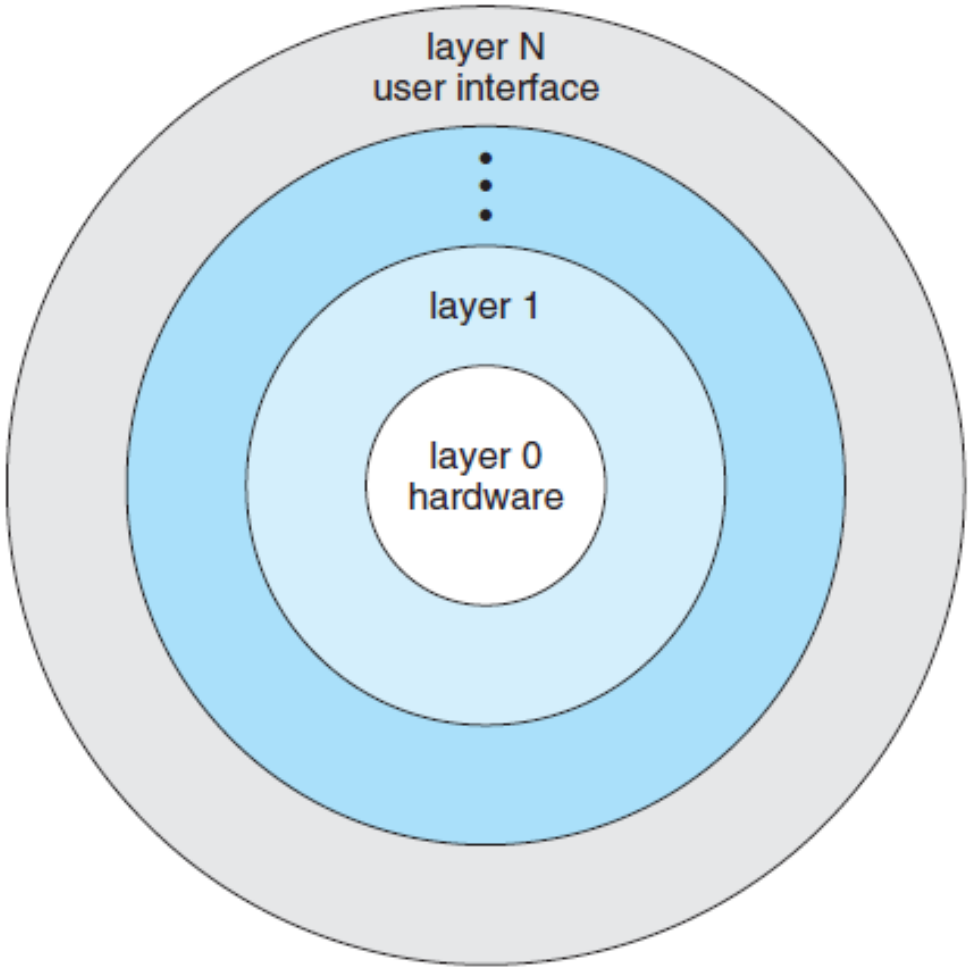


Figure 2.13 A layered operating system.

Micro-kernels (1)

- In the mid-1980s, researchers at Carnegie Mellon University developed an OS called **Mach** that modularized the kernel using the **microkernel** approach.
  - ***This method structures the OS by removing all nonessential components from the kernel and implementing them as system and user-level programs.***
  - The result is a **smaller *kernel***.
    - There is little consensus regarding which services should remain in the kernel and which should be implemented in user space.
  - **Figure 2.14** illustrates the architecture of a typical microkernel.

Micro-kernels (2)

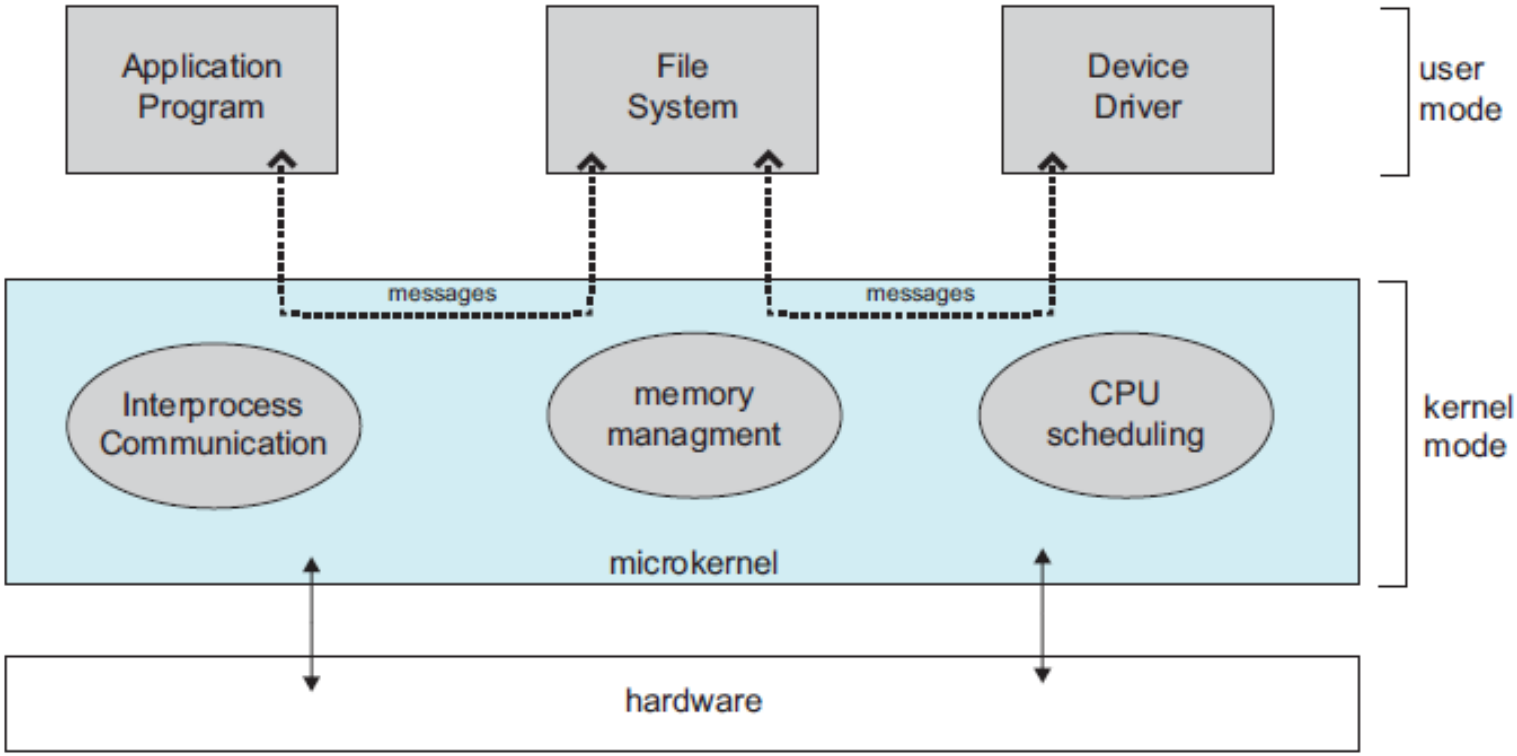


Figure 2.14 Architecture of a typical microkernel.

Micro-kernels (3)

- The main function of the **microkernel** is to provide communication between the **client program** and the various services that are also running in **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 user space and consequently do not require modification of the kernel.
  - The microkernel also provides more **security and reliability**, since most services are *running as user* rather than *kernel* processes.

Micro-kernels (4)

- Unfortunately, the performance of **micro-kernels** can suffer due to increased **system-function overhead**.
  - Consider the history of **Windows NT**:
    - The first release had a **layered microkernel** organization.
    - This version's performance was low compared with that of **Windows 95**.
    - **Windows NT 4.0** partially corrected the performance problem by moving layers from user space to kernel space and integrating them more closely.
  - By the time **Windows XP** was designed, Windows architecture had become more *monolithic* than *microkernel.*

Modules (1)

- Perhaps the best current methodology for OS design involves using **loadable kernel modules**:
  - Means, *the kernel has a set of core components and links in additional services via modules, either at boot time or run time*.
    - This type of design is common in modern implementations of **UNIX**, such as **Solaris**, **Linux,** and **Mac OS X**, and **Windows.**
  - The idea of the **modular OS design** is for the **kernel** to provide ***core services*** while other services are implemented dynamically, as the kernel is running.

Modules (2)

- **Linking** services dynamically is preferable to adding new features directly to the **kernel**,
  - Thus, it is possible to build a **CPU scheduling** and **memory management** algorithms directly into the **kernel**.
- The overall result from a **modular OS** resembles a *layered system* in that each kernel section has defined, protected interfaces;
- It is more flexible than a layered system, because any module can call any other module.
- This approach is similar to the **microkernel** approach in that the primary module has only core functions and it involves the knowledge of how to load and communicate with other modules;
  - but **it is more efficient**, *because modules do not need to invoke message passing in order to communicate other modules*.

Modules (3)

- The **Solaris OS** structure is shown in **Figure 2.15**, is organized around a core kernel with **seven types of loadable kernel modules**:
  1. **Scheduling classes**
  2. **File systems**
  3. **Loadable system calls**
  4. **Executable formats**
  5. **STREAMS modules**
  6. **Miscellaneous**
  7. **Device and bus drivers**
  - **Linux** also uses **loadable kernel modules**, primarily for supporting *device drivers* and *file systems.*

Modules (4)

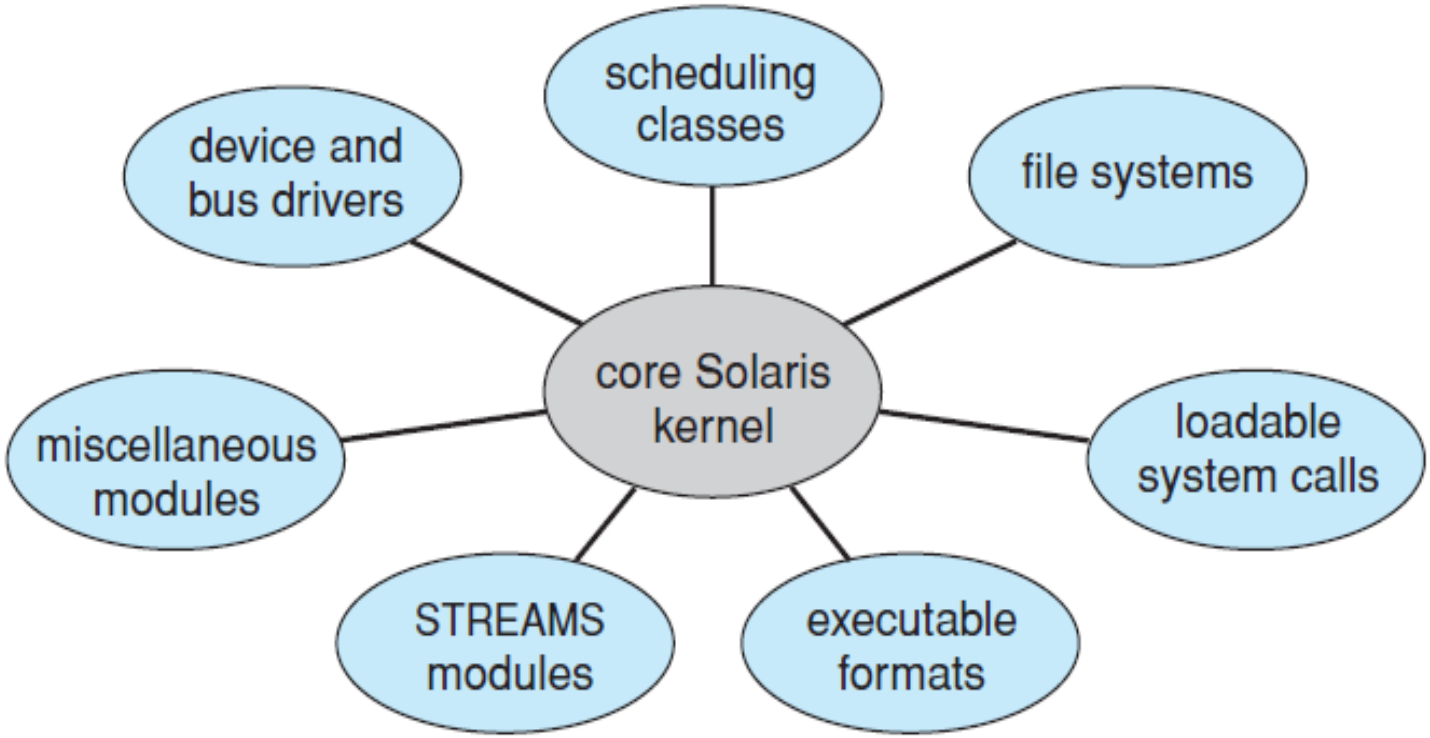


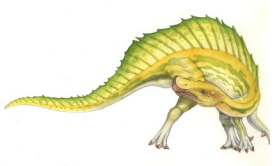
Figure 2.15 Solaris loadable modules.

Hybrid Systems (1)

- In practice, a modern OS structure is a **hybrid system** that address performance, security, and usability issues.
  - For example, both **Linux** and **Solaris** are ***monolithic***, because the OS is in a single address space to provide very efficient performance.
  - **Windows** is largely **monolithic** in nature, but it has features of **microkernel** systems, includes providing support for separate subsystems (known as ***OS personalities***) that run as **user-mode** processes.
    - **Windows** systems also provide support for dynamically **loadable kernel** modules.

Hybrid Systems (2)

- Three famous hybrid OS are:
  - Apple Mac OS X
  - iOS and
  - Android.



iOS

- Apple mobile OS for ***iPhone***, ***iPad***
  - Structured on Mac OS X, added functionality
  - Does not run OS X applications natively
    - Also runs on different CPU architecture (ARM vs. Intel)
  - **Cocoa Touch Objective-C** API for developing apps
  - **Media services** layer for graphics, audio, video
  - **Core services** provides cloud computing, databases
  - Core operating system, based on Mac OS X kernel

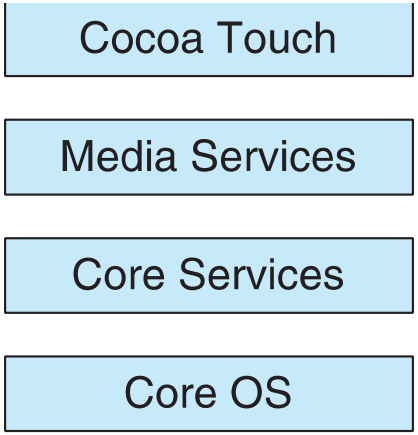


Figure 2.17 iOS Architecture

Android

- The **Android OS** was designed by the **Open Handset Alliance** (led primarily by Google) and was developed for Android smartphones and tablet computers.
  - Whereas **iOS** is designed to run only on Apple mobile devices and is a **close-sourced** one
  - **Android** OS runs on a variety of mobile platforms and is **open-sourced**, partly explaining its rapid rise in popularity.
  - The structure of Android appears in **Figure 2.18**.

Android (1)

- **Linux** is used primarily for process, memory, and device-driver support for hardware and has been expanded to include power management.
- The Android runtime environment includes a core set of libraries as well as the **Dalvik virtual machine**.
- Software designers for Android devices develop applications in the **Java**.
- However, rather than using the standard *Java API*, Google has designed a separate Android API for Java development.

Android (2)

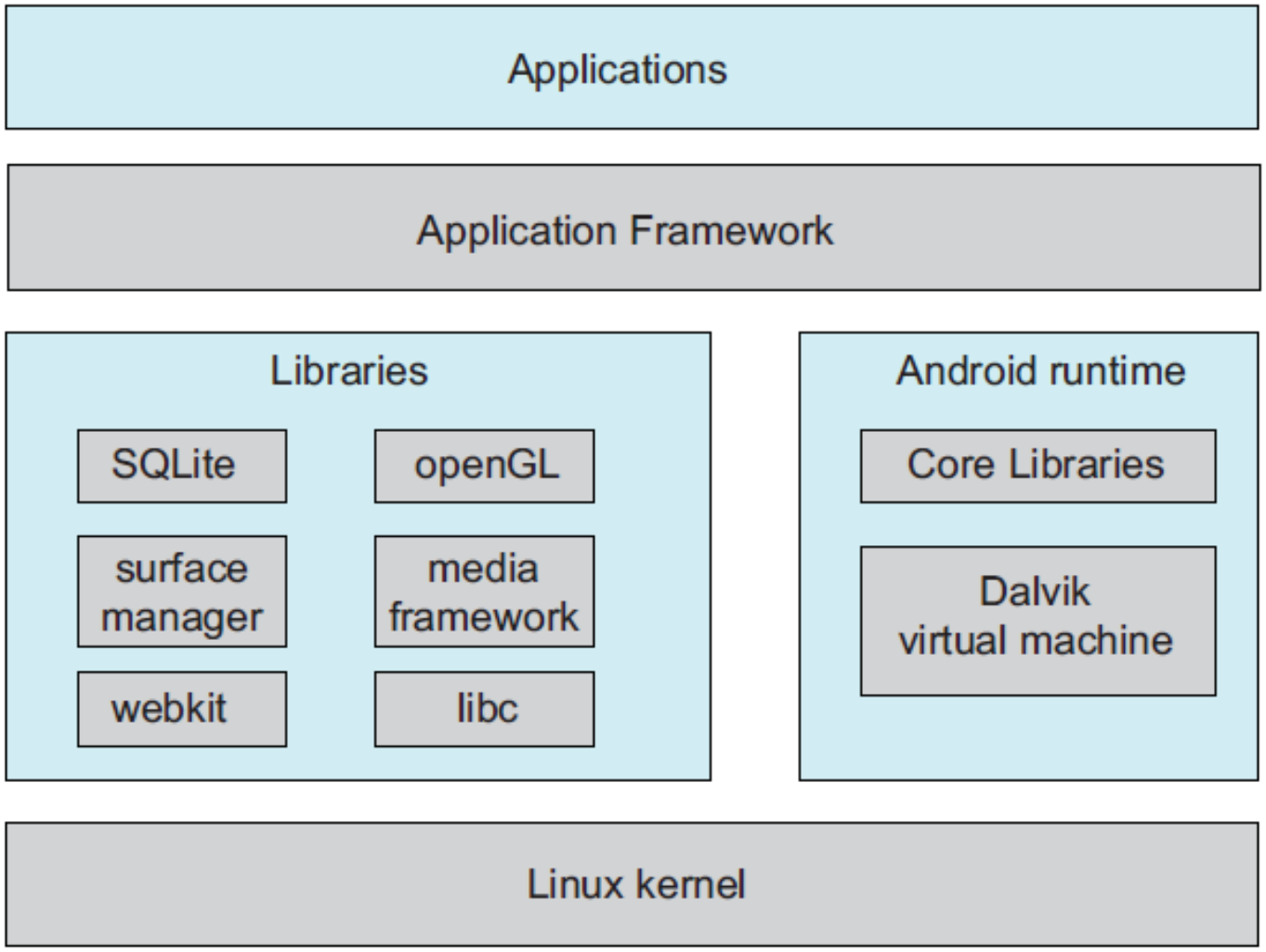
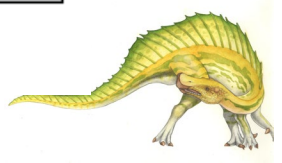
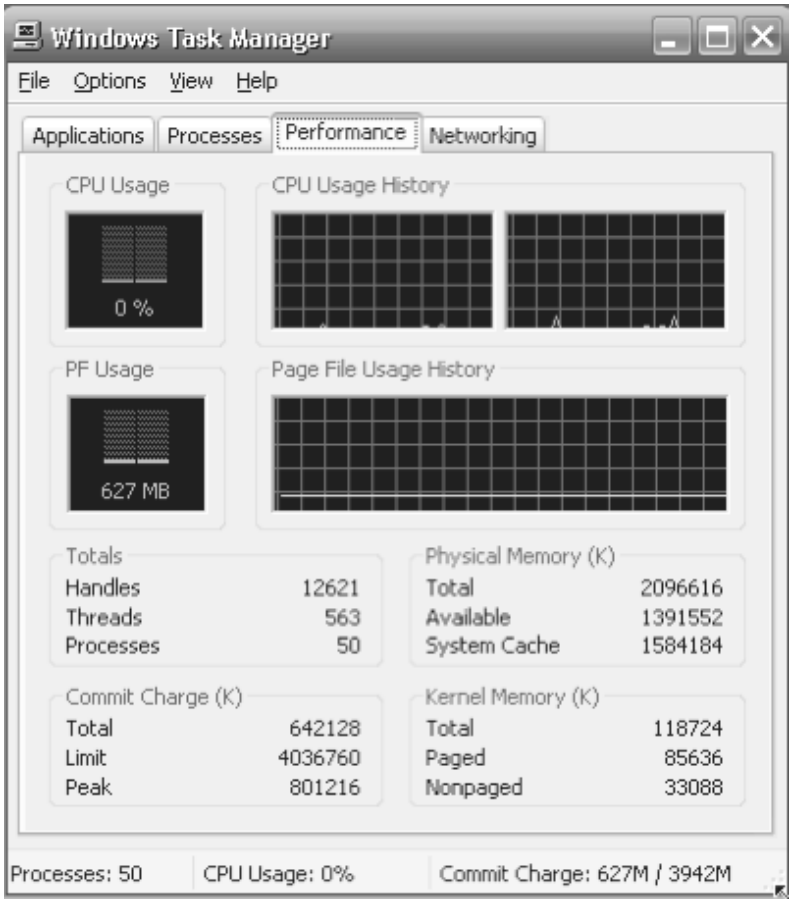


Figure 2.18 Architecture of Google's Android.



Performance Tuning

- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- For example, “top” program or Windows Task Manager



End of Chapter 2