BACSE106: Operating Systems

Module 1:

Operating Systems and Structure

Course Objectives

1. To introduce the fundamental concepts, design issues, and core functionalities of operating systems.

2. To build skills in managing processes, memory, files, devices, and ensuring protection and security, using algorithms and practical simulations.

3. To introduce modern developments like virtualization, distributed operating systems, and cloud-based systems.

Course Outcomes

1. Understand the structure, functionalities, and foundational concepts of operating systems.

2. Apply process scheduling algorithms and analyze thread models and deadlock-handling methods.

3. Implement synchronization mechanisms and solve concurrency problems in multi-process systems.

4. Evaluate memory management and file system strategies used in operating systems.

5. Analyze device management, protection mechanisms, and virtualization techniques in operating systems.

Module 1: Operating Systems and Structure

Operating system basics - History of operating systems - OS structure: Monolithic, Layered, Microkernel, Modular - OS services - System calls and types - Interrupts - OS design issues – Building and booting an OS.

Text Book:

Abraham Silberschatz, Peter B. Galvin, Greg Gagne, "Operating System Concepts", Wiley, 10th Edition, 2018

Reference Books:

1. Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, "Operating Systems: Three Easy Pieces", Arpaci Dusseau Books, 1st Edition, 2023
2. Andrew S. Tanenbaum, Herbert Bos, "Modern Operating Systems", Pearson Education, 4th Edition, 2015
3. William Stallings, "Operating Systems: Internals and Design Principles", Pearson, 9th Edition, 2018

Operating System Basics

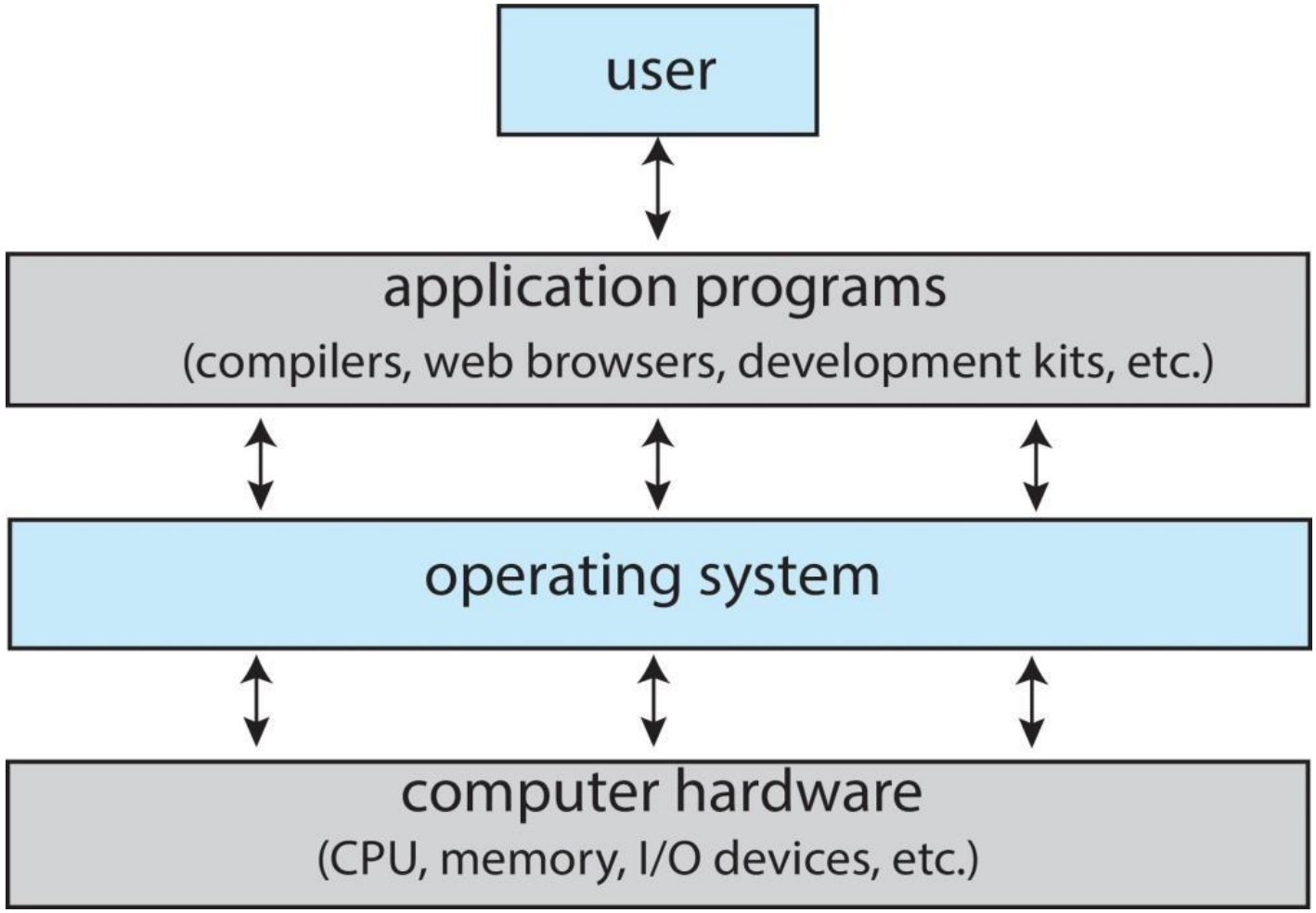
What is an Operating System (OS)?

- OS is a program that acts as an intermediary between a user of a computer and the computer hardware
- The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner.
- An operating system is software that manages the computer hardware.
- 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 Components

- Computer system can be divided into four components:
  - Hardware – provides basic computing resources
    - CPU, memory, I/O devices
  - Operating system
    - Controls and coordinates use of hardware among various applications and users
  - Application programs – define the ways in which the system resources are used to solve the computing problems of the users
    - Word processors, compilers, web browsers, database systems, video games
  - Users
    - People, machines, other computers

Abstract View of Computer



VIEWS OF OS

Two Views:

- User View
- System View

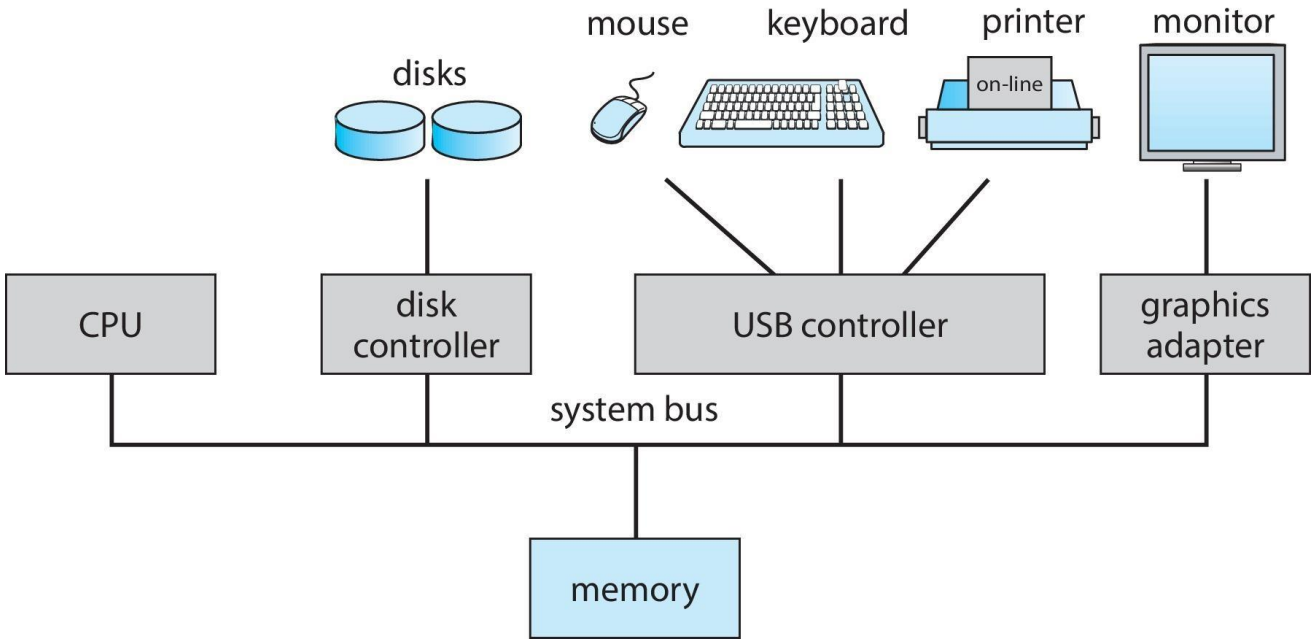
- **User View**
  - The user's view of the computer varies according to the interface being used (E.g. Mobile devices, touch screens, embedded systems, etc).
  - Users want convenience, ease of use and good performance. So, the operating system is designed mostly for ease of use
    - Don't care about resource utilization

VIEWS OF OS (CONTD…)

- **System View**
  - From the computer's point of view, the operating system is the program most intimately involved with the hardware.
  - Operating system can be viewed as a resource allocator
  - Computer system has many resources that may be required to solve a problem: CPU time, memory space, storage space, I/O devices, and so on. The operating system acts as the manager of these resources.
  - An operating system is a control program.
  - A control program manages the execution of user programs to prevent errors and improper use of the computer

Computer System Organization

- Computer-system operation
  - One or more CPUs, device controllers connect through common
    bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



Operating System Operations

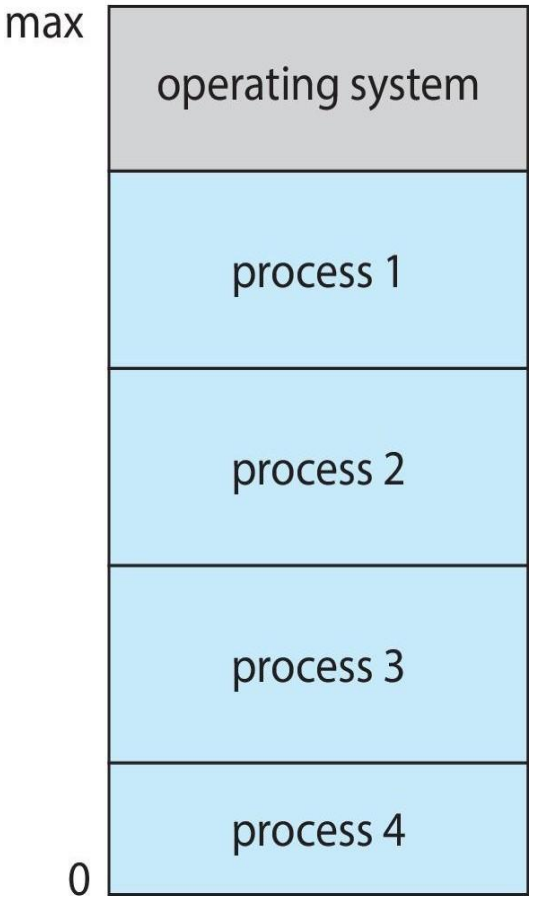
- An operating system provides the environment within which programs are executed
- When the computer is powered up or rebooted the bootstrap program must locate the operating-system kernel and load it into memory.
- The Bootstrap program initializes all aspects of the system, from CPU registers to device controllers to memory contents.

- Various OS operations include:
  - Multiprogramming and multitasking
  - Dual mode and multimode operations
  - Timer

Multiprogramming (Batch system)

- Single user cannot always keep CPU and I/O devices busy
- Multiprogramming organizes jobs (code and data) so CPU always has one to execute
- A subset of total jobs is kept in memory
- One job selected and run via job scheduling
- When job has to wait (for I/O for example), OS switches to another job
- In a non-multiprogrammed system, the CPU would sit idle.
- In a multiprogrammed system, the operating system simply switches to, and executes, another process.
- Multiprogramming increases CPU Utilization
- In a multiprogrammed system, a program in execution is termed a process

Memory Layout



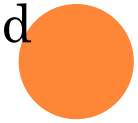
MULTITASKING (TIMESHARING)

- A logical extension of Batch systems– the CPU switches jobs so frequently that users can interact with each job while it is running, creating interactive computing
  - If several processes are ready to run at the same time, the system must choose which process will run next. Making this decision is called CPU scheduling.
  - The operating system must ensure reasonable response time. A common method for doing so is virtual memory, a technique that allows the execution of a process that is not completely in memory.

Dual-mode or Multimode Operation

- Dual-mode operation allows OS to protect itself and other system components
  - User mode and kernel mode (also called supervisor mode, system mode, or privileged)
- Mode bit provided by hardware
  - Provides ability to distinguish when system is running user code or kernel code.
  - The mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1).

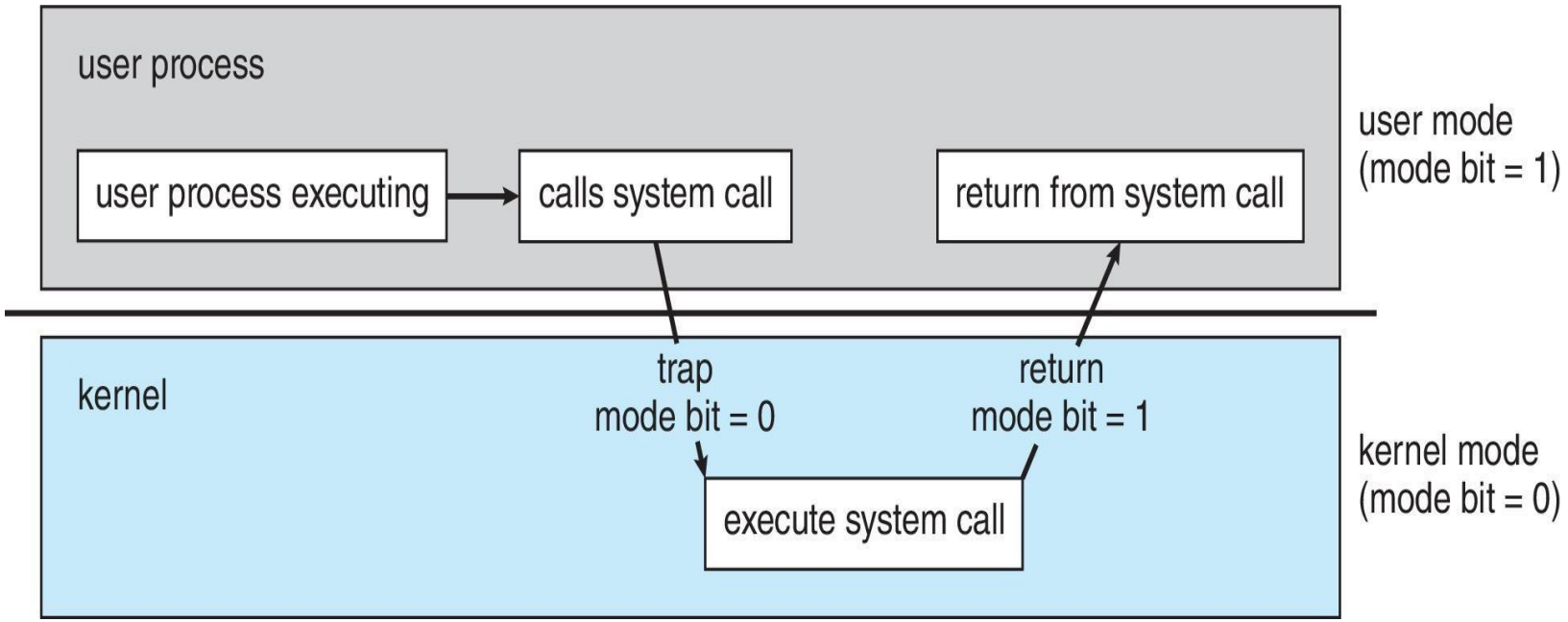
- At system boot time, the hardware starts in kernel mode.
- The operating system is then loaded and starts user applications in user mode.
- Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0).
- The hardware allows privileged instructions to be executed only in kernel mode.



Multi Mode Operation

- The concept of modes can be extended beyond two modes.
- For example, Intel processors have four separate **protection rings**, where ring 0 is kernel mode and ring 3 is user mode.
- Rings 1 and 2 could be used for various operating-system services.

Transition from User to Kernel Mode



Timer

Timer to prevent infinite loop or to fail to call system services and never return control to the operating system. To achieve this:

- Timer is set to interrupt the computer after some time period
- Keep a counter that is decremented by the physical clock
- Operating system sets the counter.
- When the counter reaches zero, an interrupt occurs.
- Timer will be set up before scheduling process to regain control or terminate program that exceeds allotted time

Storage Structure

- Main memory – only large storage media that the CPU can access directly.
  - Random access
  - Typically volatile.
  - Loses its content when power is turned off or otherwise lost
  - Typically random-access memory in the form of Dynamic Random-access Memory (DRAM)
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity. In power off, data will not be lost.
- The CPU can load instructions only from memory, so any programs must first be loaded into memory to run.
- The first program to run on computer power-on is a bootstrap program, which then loads the operating system.

Storage Notations

- The basic unit of computer storage is the bit, 0 and 1.
- A byte is 8 bits.
- A word is made up of one or more bytes.
- For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words.

- A kilobyte, or KB, is 1,024 bytes;
- a megabyte, or MB, is 1,024^2 bytes;
- a gigabyte, or GB, is 1,024^3 bytes;
- a terabyte, or TB, is 1,024^4 bytes;
- and a petabyte, or PB, is 1,024^5 bytes.

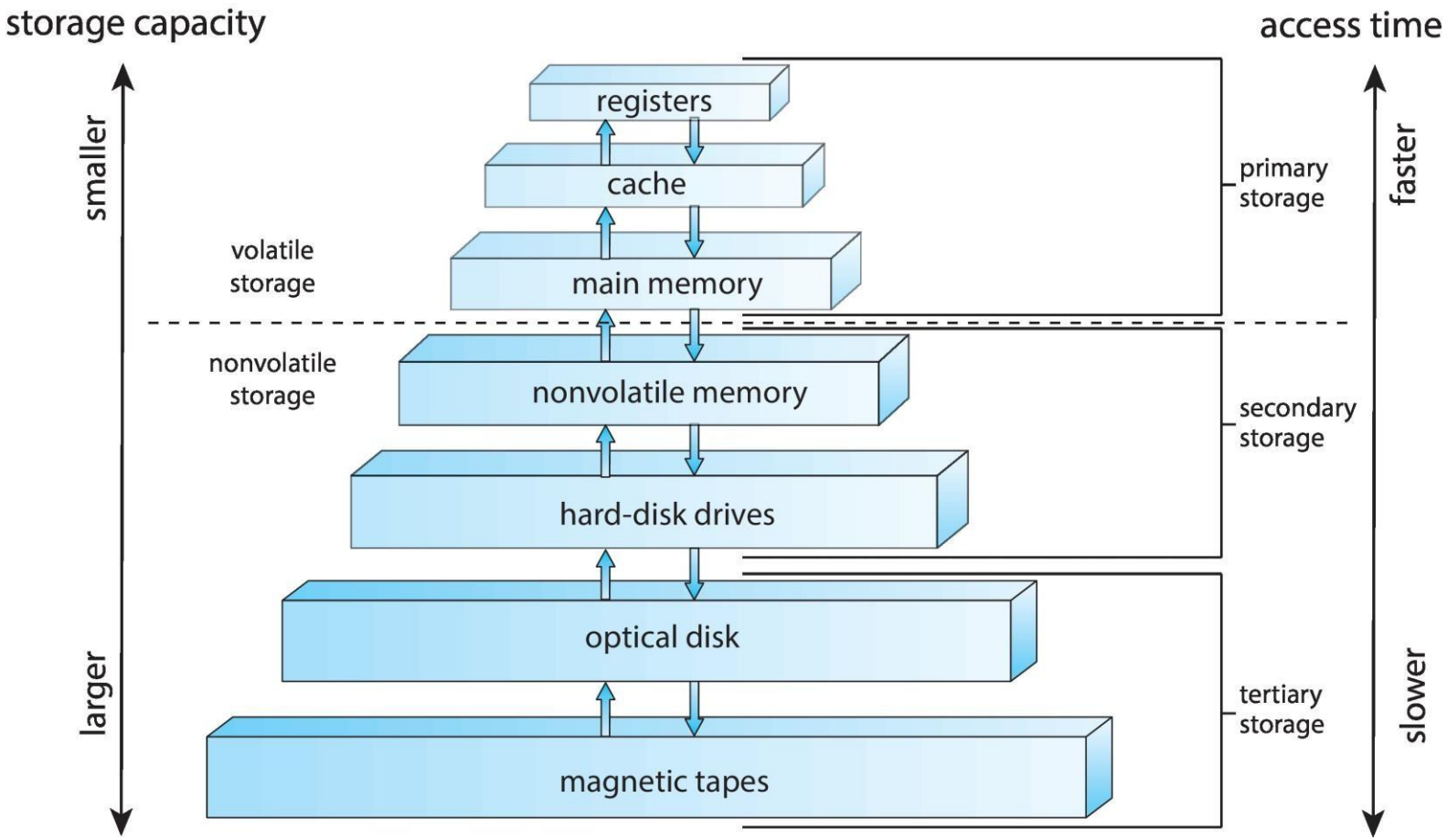
- Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes.

Storage Structure (Cont.)

- Hard Disk Drives (HDD) – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into tracks, which are subdivided into sectors
  - The disk controller determines the logical interaction between the device and the computer
- Non-volatile memory (NVM) devices– faster than hard disks, nonvolatile
  - Various technologies
  - Becoming more popular as capacity and performance increases, price drops

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility

Storage-Device Hierarchy

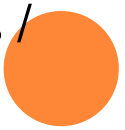


OS: Resource Management

- An operating system is a resource manager. It manages the following:
  - Process Management
  - Memory Management
  - File System Management
  - Storage Management
  - Cache Management
  - I/O System Management

Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a passive entity; process is an active entity.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one program counter specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads



Process Management Activities

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Memory Management

- To execute a program all (or part) of the instructions must be in memory
- All (or part) of the data that is needed by the program must be in memory
- Memory management determines what is in memory and when
  - Optimizing CPU utilization and computer response to users
- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into
    - and out of memory
  - Allocating and deallocating memory space as needed

File-system Management

- File-System management
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
  - OS activities include
    - Creating and deleting files and directories
    - Primitives to manipulate files and directories
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media

Mass-Storage Management

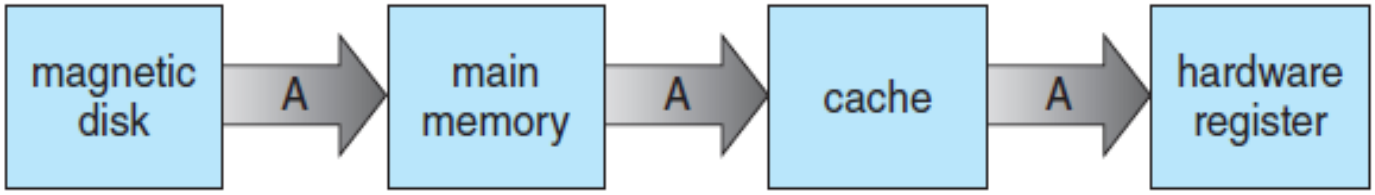
- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time.
- Entire speed of computer operation hinges on disk subsystem and its algorithms
  - Most programs—including compilers, web browsers, word processors, and games—are stored on these devices until loaded into memory.
  - Backups of disk data, storage of seldom-used data, and long-term archival storage are some examples.
  - Magnetic tape drives and their tapes and CD DVD and Blu-ray
  - drives and platters are typical tertiary storage devices.
- OS activities
  - Mounting and unmounting
  - Free-space management
  - Storage allocation
  - Disk scheduling
  - Partitioning
  - Protection

CACHING

- Caching is an important principle of computer systems Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy
- Most systems have an instruction cache to hold the instructions expected to be executed next.
- Caches have limited size.

CACHING

Migration of integer A from disk to register



- In a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache.
- In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute in parallel, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides.
- This situation is called **cache coherency**, and it is usually a hardware issue.

Characteristics of Storage Units

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 SRAM	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

I/O System Management

The I/O subsystem 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.

History of Operating Systems

History of OS

- **1940s-1950s:** Computers had **no operating systems**; programmers interacted directly with the machine, manually loading programs via switches or punched cards.
- **1956:** The first operating system, GM-NAA I/O, was developed by General Motors for the IBM 704. It introduced batch processing, where a series of jobs was run automatically.
- 1960s: Multiprogramming was developed to improve CPU efficiency by keeping the processor busy while a program was waiting for input/output. Time-sharing systems like CTSS and Multics emerged, allowing multiple users to interact with a single system simultaneously.
- 1970s: Unix, developed by Ken Thompson and Dennis Ritchie at Bell Labs, revolutionized OS design with its multitasking and portability, and became widely adopted by universities and research institutions.

History of OS (Cont.)

- 1981: MS-DOS (Microsoft Disk Operating System) was created, initially based on CP/M, to run on the new personal computers and introduced a command-line interface.
- 1984: The Apple Macintosh introduced the first widely popular graphical user interface (GUI) with a mouse, making computers much more user-friendly.
- 1985: Microsoft Windows was released, offering a GUI on top of the MS-DOS system.
- **1990s:** Open-source operating systems like Linux became popular, allowing for community-driven development and modification. Windows and Mac OS continued to refine their GUIs.
- **2000s:** Mobile operating systems like iOS (2007) and Android (2008) gained prominence with the rise of smartphones.
- **2010s-Present:** The focus shifted to cloud computing, the Internet of Things (IoT), and integrating artificial intelligence (AI) features into operating systems.

Operating Systems Structure

Operating System Structure

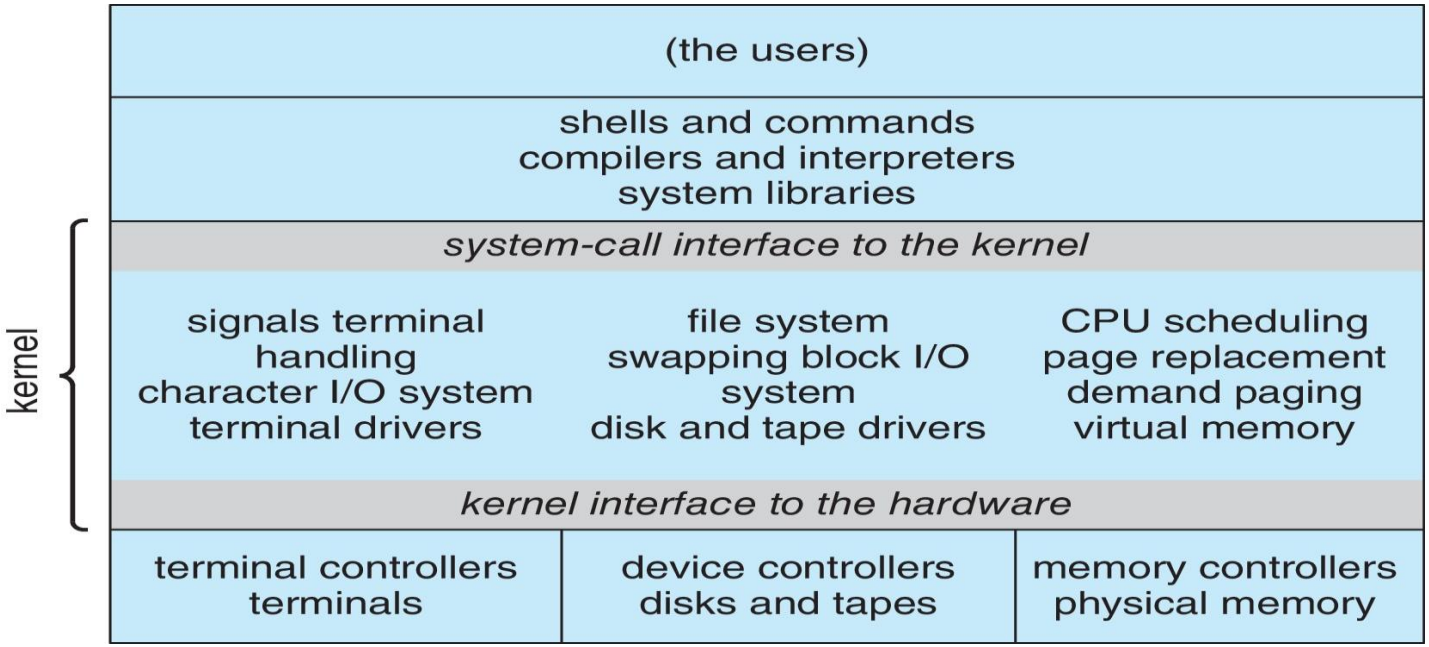
A common approach is to partition the task into small components, or modules, rather than have one single system. The various OS structures are:

- **Monolithic Structure**
- **Layered Approach**
- **Microkernels**
- **Modules**
- **Hybrid Systems**

MONOLITHIC STRUCTURE – ORIGINAL UNIX

- The simplest structure for organizing an operating system is no structure at all.
- That is, place all of the functionality of the kernel into a single, static binary file
- This runs in a single address space. This approach is known as a **monolithic** structure. This is a common technique for designing operating systems.
- An example of such limited 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.

TRADITIONAL UNIX SYSTEM STRUCTURE

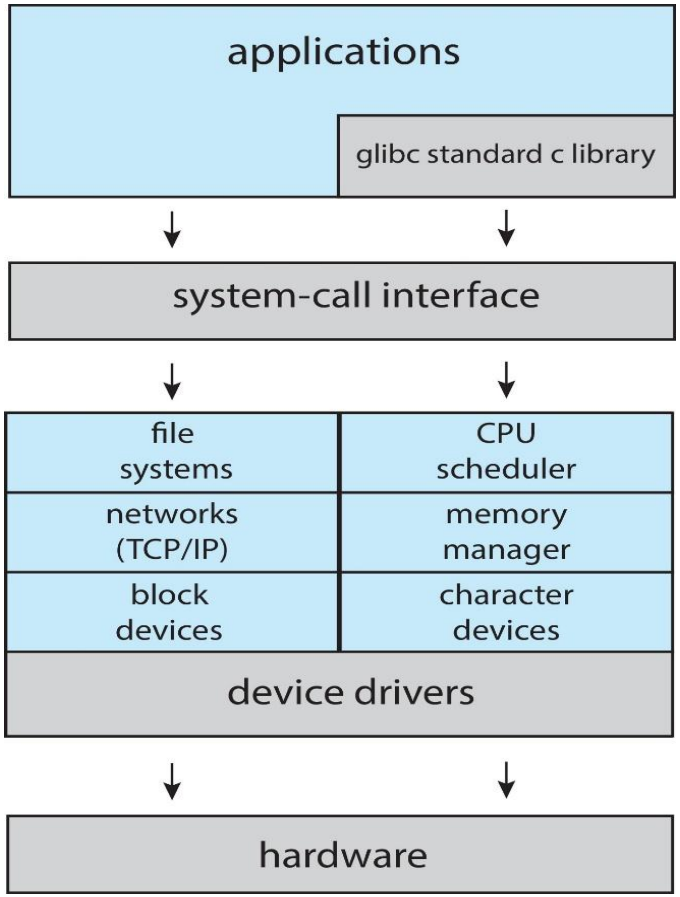


- Everything below the system-call interface and above the physical hardware is the kernel.
- The kernel provides the file system, CPU scheduling, memory management, and other operating system functions through system calls.
- An enormous amount of functionality to be combined into one single address space

LINUX SYSTEM STRUCTURE

Monolithic plus modular design

- 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.
- Uses a modular design that allows the kernel to be modified during run time.

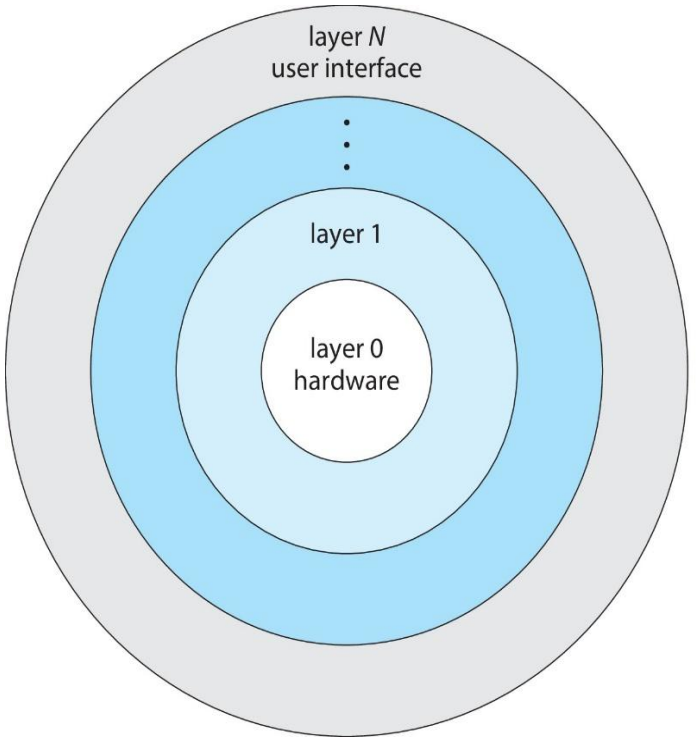


Drawback

- They are difficult to implement and extend

Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- The main advantage of the layered approach is simplicity of construction and debugging.
- The first layer can be debugged without any concern for the rest of the system.
- Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on.
- If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged.



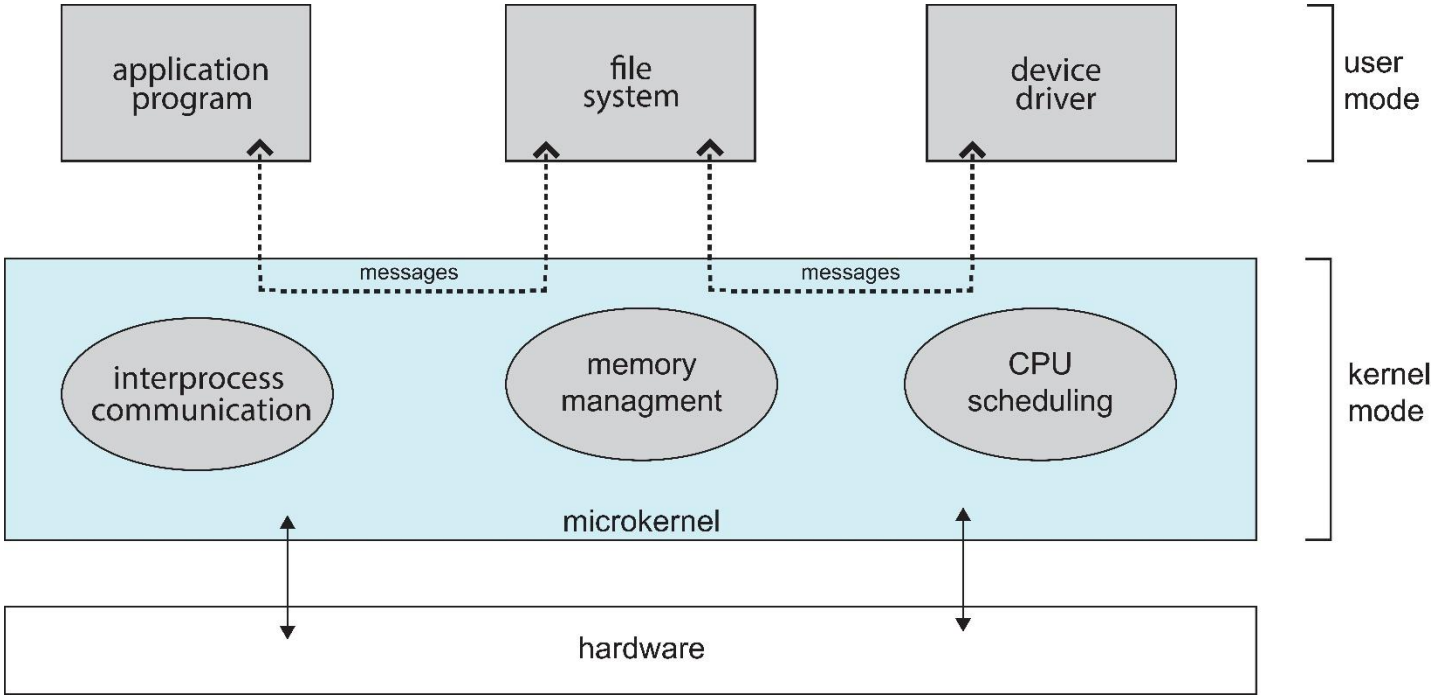
Drawback

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.

MICROKERNELS

- Microkernels provide minimal process and memory management, in addition to a communication facility
- This method structures the operating system 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.
- 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.
- The best-known illustration of a microkernel operating system is Darwin, the kernel component of the macOS and iOS operating systems.
- Darwin, consists of two kernels, one of which is the Mach microkernel.

MICROKERNEL SYSTEM STRUCTURE



- Another example is QNX, a real-time operating system for embedded systems.
- The QNX Neutrino microkernel provides services for message passing and process scheduling.
- It also handles low-level network communication and hardware interrupts.
- All other services in QNX are provided by standard processes that run outside the kernel in user mode

Drawbacks

- The performance of microkernels can suffer due to increased system-function overhead.
- When two user-level services must communicate, messages must be copied between the services, which reside in separate address space.
- The operating system may have to switch from one process to the next to exchange the messages.
- The overhead involved in copying messages and switching between processes has been the largest impediment to the growth of microkernel-based operating systems.

Modules

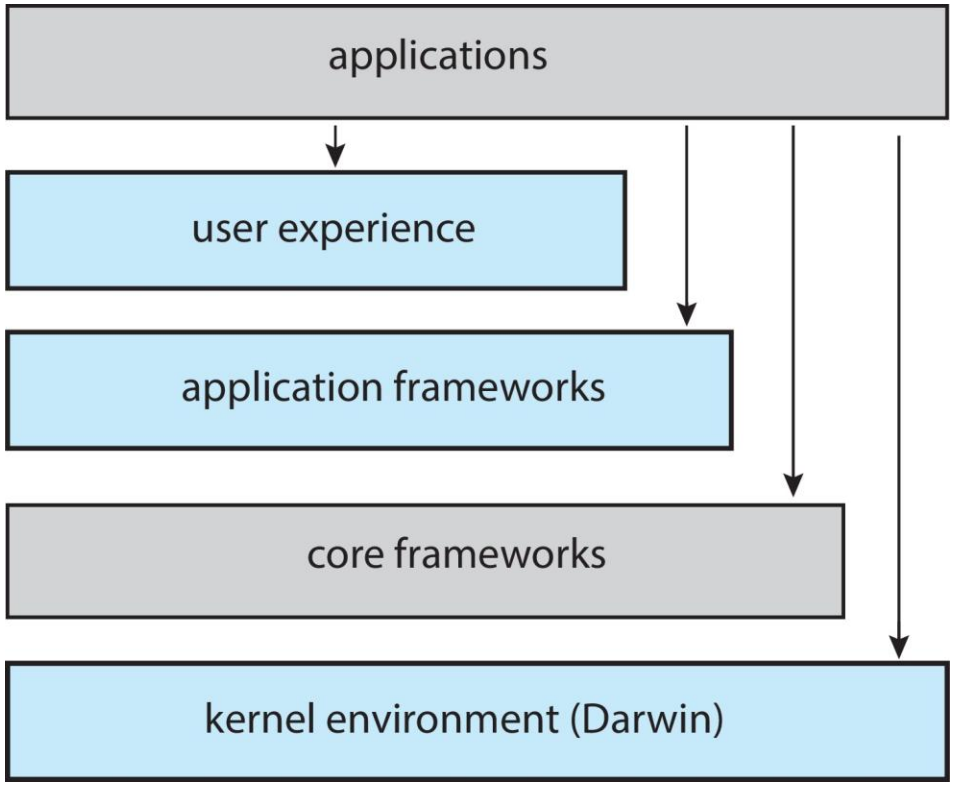
- Many modern operating systems implement **loadable kernel** mo**dules** (**LKMs**) the kernel has a set of core
- Components and can link in additional services via modules, either at boot time or during run time.
- For example, we might build CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules.
- Overall, similar to layers but with more flexible
  - Linux, Solaris, etc.

Hybrid Systems

- Most modern operating systems are not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
- Linux is monolithic, because having the operating system in a single address space.
- It also modular, so that new functionality can be dynamically added to the kernel.
- Windows is largely monolithic as well but it retains some behavior typical of microkernel systems, including providing support for separate subsystems
- Windows systems also provide support for dynamically loadable kernel modules.

MACOS AND IOS STRUCTURE

- Apple's macOS operating system is designed to run primarily on desktop and laptop computer systems, whereas iOS is a mobile operating system designed for the iPhone smartphone and iPad tablet computer.
- iOS is designed to run on Apple mobile devices.

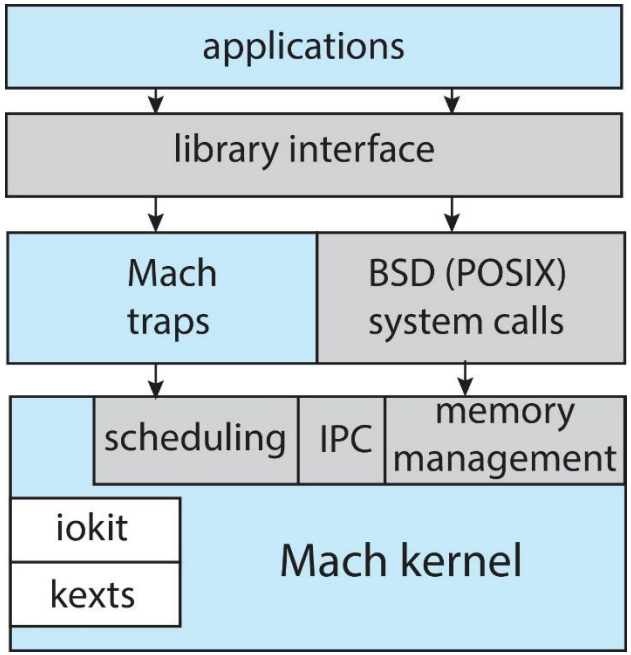


LAYER DETAILS….

- **User experience layer**.
  - This layer defines the software interface that allows users to interact with the computing devices.
  - macOS uses the ***Aqua*** user interface, which is designed for a mouse or trackpad, whereas iOS uses the ***Springboard*** user interface, which is designed for touch devices.
- **Application frameworks layer**.
  - This layer includes the ***Cocoa*** and ***Cocoa Touch*** frameworks, which provide an API for the Objective-C and Swift programming languages.
  - The primary difference between Cocoa and Cocoa Touch is that the former is used for developing macOS applications, and the latter by iOS to provide support for hardware features unique to mobile devices, such as touch screens.
- **Core frameworks**.
  - This layer defines frameworks that support graphics and media including, Quicktime and OpenGL.
- **Kernel environment**.
  - This environment, also known as **Darwin**, includes the Mach microkernel and the BSD UNIX kernel.

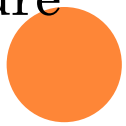
DARWIN

- DARWIN uses a hybrid structure.
- Darwin is a layered system that consists primarily of the Mach microkernel and the BSD UNIX kernel
- Darwin provides two system-call interfaces: Mach system calls (known as traps) and BSD system calls (which provide POSIX functionality).
- Mach provides fundamental operating system services, including memory management, CPU scheduling, and interprocess communication (IPC) facilities such as message passing and remote procedure calls (RPCs).
- Kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules (which macOS refers to as kernel extensions, or kexts).



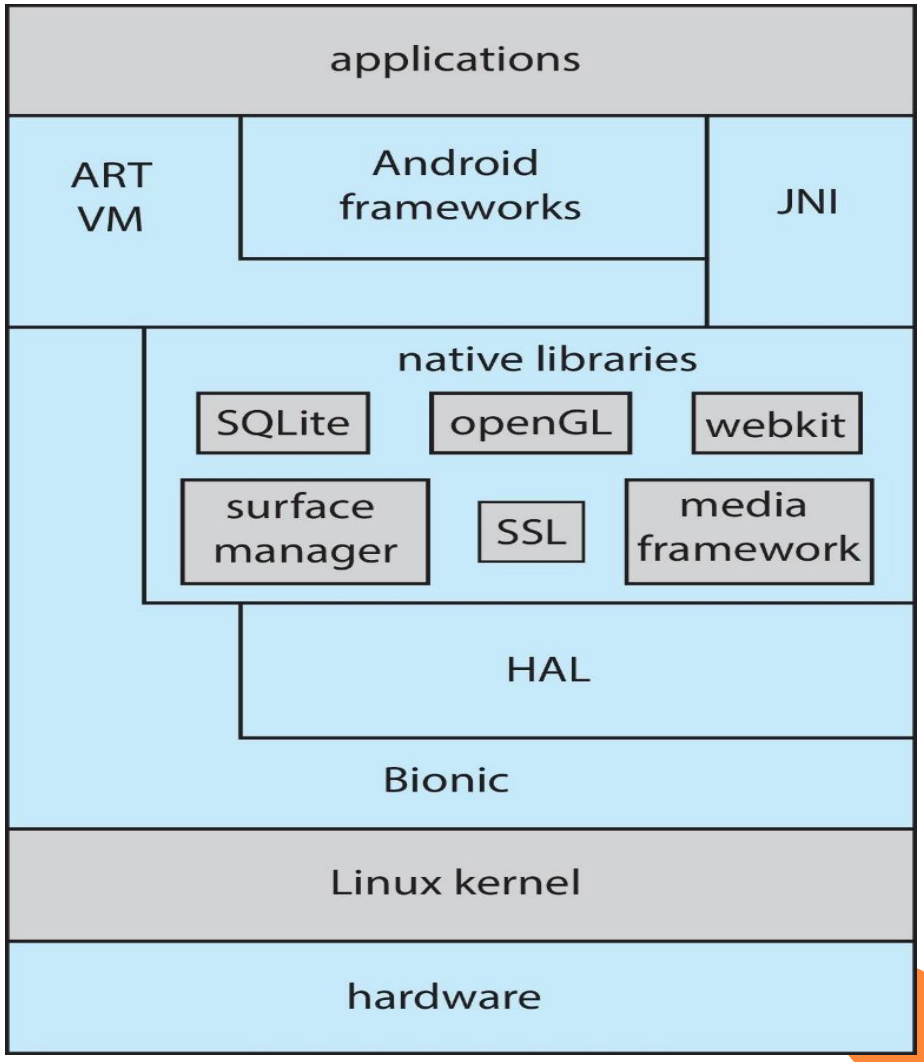
ANDROID

- The Android operating system was designed by the Open Handset Alliance (led primarily by Google) and was developed for Android smartphones and tablet computers.
- Android devices develop applications in the Java language, but they do not generally use the standard Java API.
- Google has designed a separate Android API for Java development.
- Java applications are compiled into a form that can execute on the Android RunTime ART, a virtual machine designed for Android and optimized for mobile devices with limited memory and CPU processing capabilities.
- Android developers can also write Java programs that use the Java native interface—or JNI
- The set of native libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and network support, such as secure sockets (SSLs).



ANDROID ARCHITECTURE

- Java programs are first compiled to a Java bytecode .class file and then translated into an executable .dex file.
- .dex files are compiled into native machine code when they are installed on a device, from which they can execute on the ART.
- Google developed the Bionic standard C library for Android.



Operating Systems Services

System Services

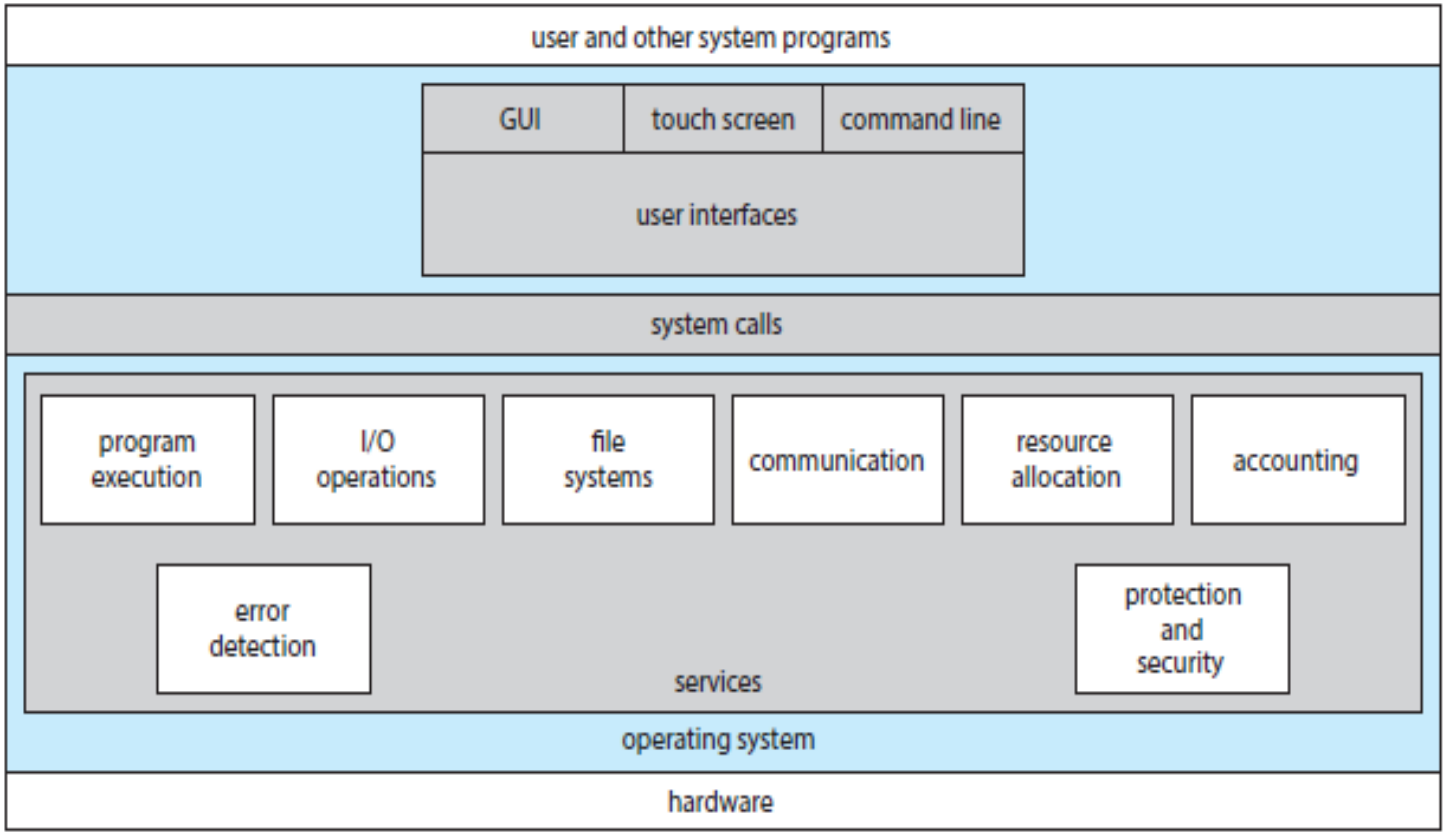
- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation
  - Status information sometimes stored in a file
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs
- Most users' view of the operating system is defined by system programs, not the actual system calls

System Services (Cont.)

- Operating system services provides functions that are helpful to the user. They are:
  - **User Interface**
  - **Program execution**
  - **I/O operations**
  - **File-system manipulation**
  - **Communications**
  - **Error detection**
  - **Resource allocation**
  - **Logging**
  - **Protection and security**

System Services (Cont.)

View of operating system services



System Services (Cont.)

- User interface
- Most commonly used interfaces,
  - Graphical user interface (GUI)
  - Touch Screen
  - Common Line Interface
- Program execution.
  - The system must be able to load a program into memory and to run that program.
- I/O operations.
  - A running program may require I/O, which may involve a file or an I/O device.
  - For efficiency and protection, users usually cannot control I/O devices directly.
  - Therefore, the operating system must provide a means to do I/O.

System Services (Cont.)

- File-system manipulation
  - Programs need to read and write files and directories.
  - They also need to create and delete them by name, search for a given file, and list file information.
- Communications.
  - There are many circumstances in which one process needs to exchange information with another process.
  - Such communication may occur between processes that are executing on the same computer or between processes.
  - 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.

System Services (Cont.)

- Error detection
  - The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware, in I/O devices, and in the user program.
  - For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.
- Resource allocation
  - When there are multiple processes running at the same time, resources must be allocated to each of them.
- Logging
- We want to keep track of which programs use how much and what kinds of computer resources.
- Protection and security
  - Protection involves ensuring that all access to system resources is controlled.
  - Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate itself, usually by means of a password, to gain access to system resources. It extends to defending external I/O devices, including network adapters, from invalid access attempts.

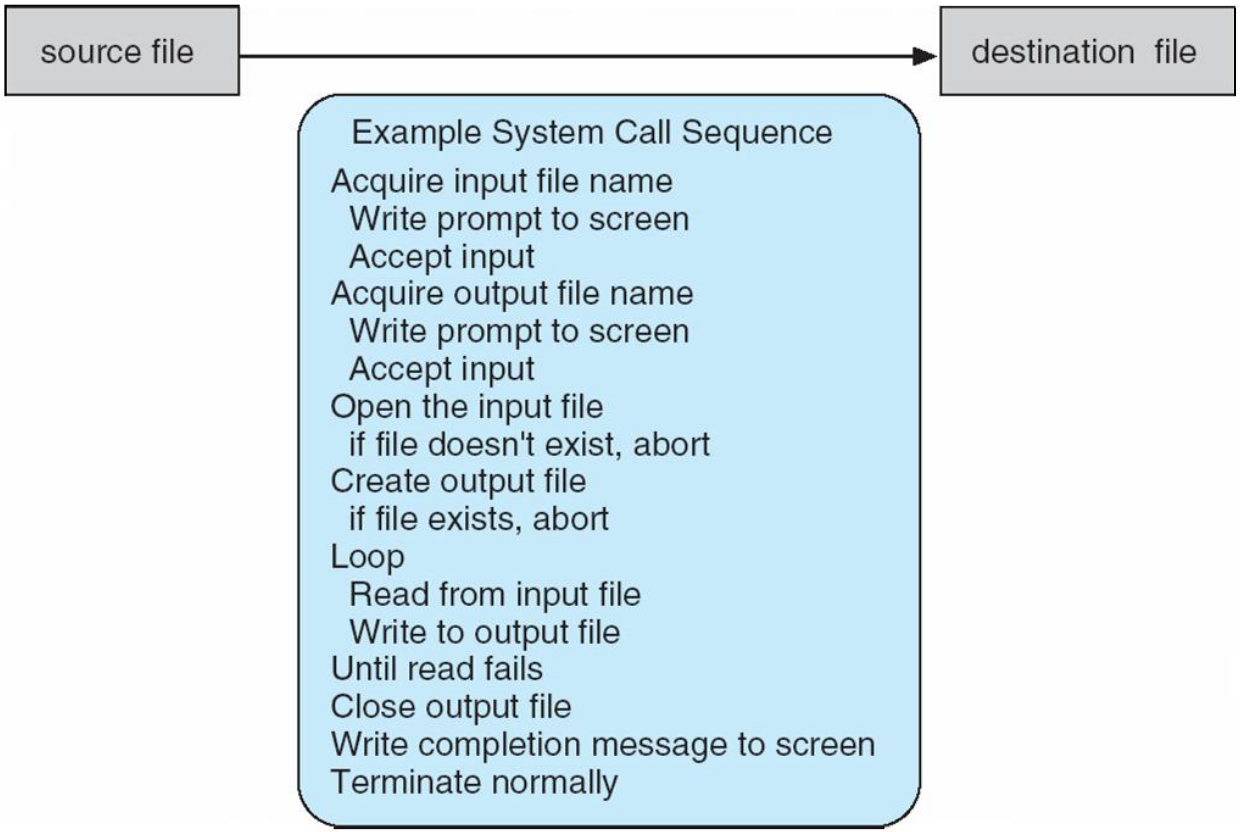
System Calls and Types

System Calls

- **System calls** provide an interface to the services made available by an operating system.
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Example of System Calls

- System call sequence to copy the contents of one file to another file



EXAMPLE OF STANDARD API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

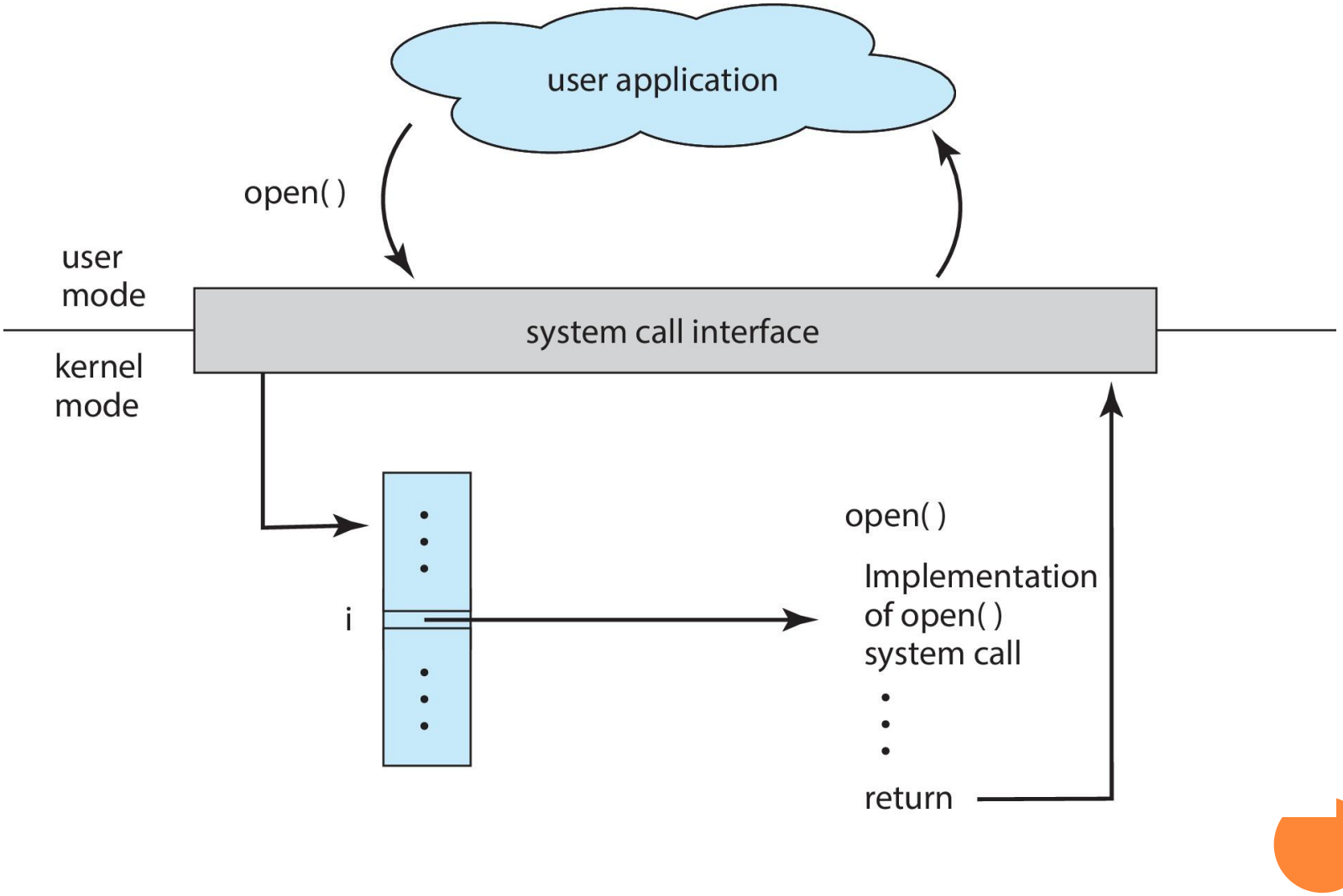
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

System Call Implementation

- Typically, a number is associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller needs to know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

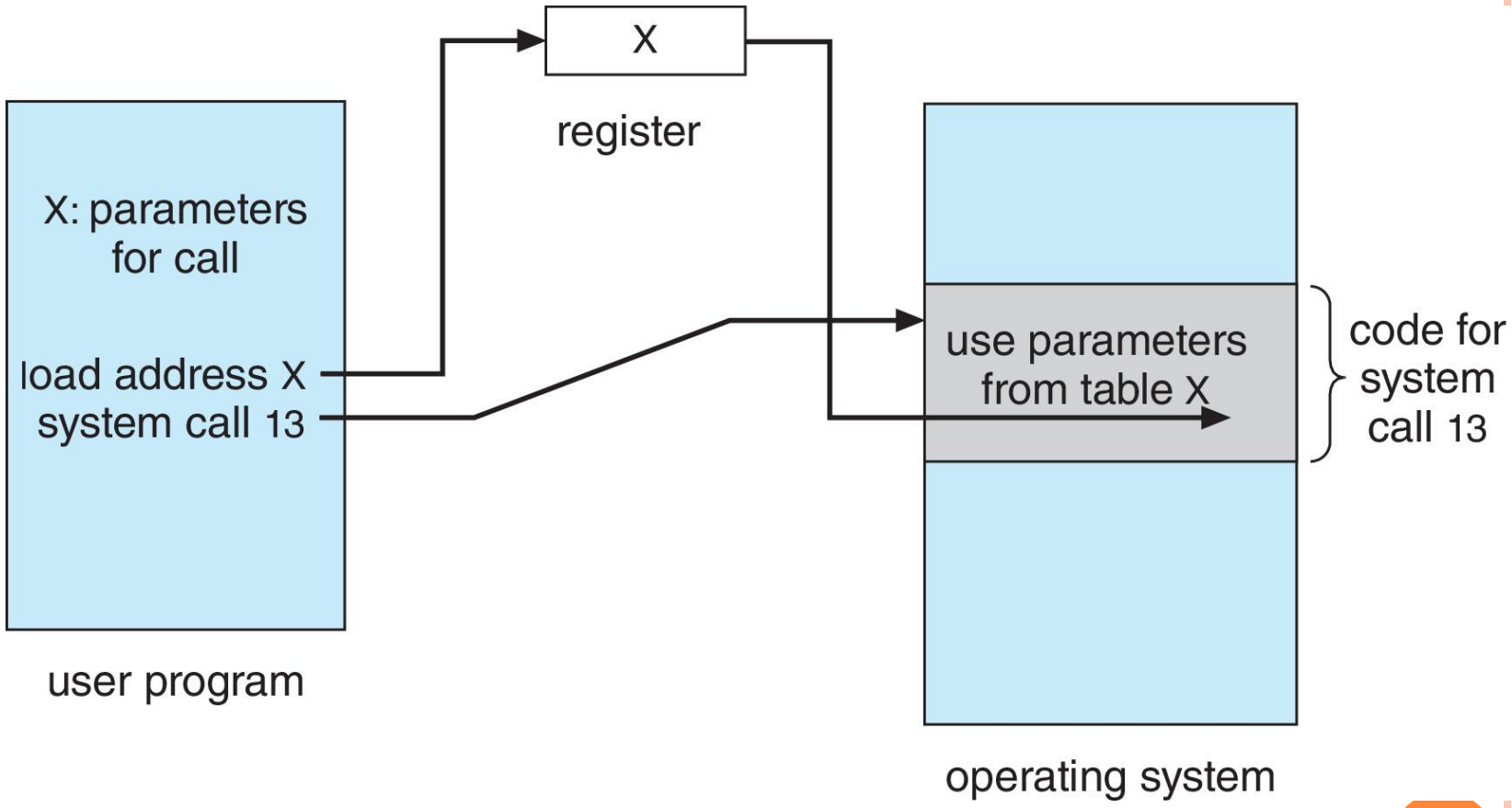
API – System Call – OS Relationship



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



Types of System Calls

- System calls can be grouped roughly into six major categories:
  1. **Process control**,
  2. **File management**,
  3. **Device management**,
  4. **Information maintenance**,
  5. **Communications**,
  6. **Protection**

Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes

Types of System Calls (Cont.)

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

Types of System Calls (Cont.)

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

Types of System Calls (Cont.)

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Examples of Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix

Process control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

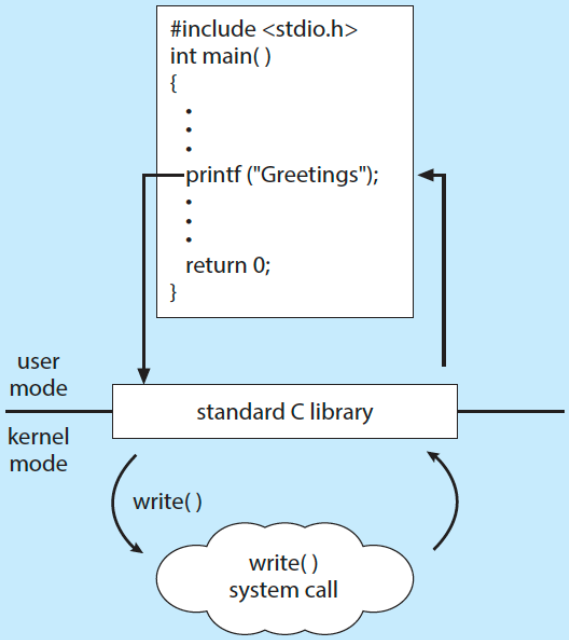
Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

Standard C Library Example

- C program invoking printf() library call, which calls write() system call

THE 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:

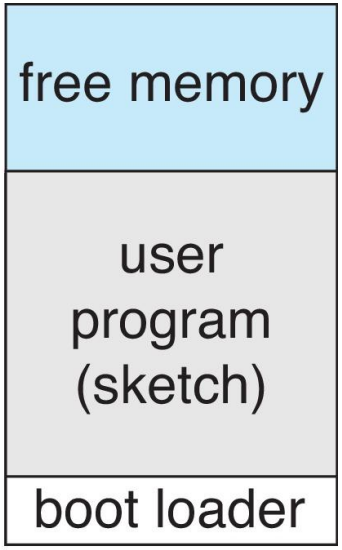


Example: Arduino

- Single-tasking
- No operating system
- Programs (sketch) loaded via USB into flash memory
- Single memory space
- Boot loader loads program
- Program exit -> shell reloaded

(a)

At system startup

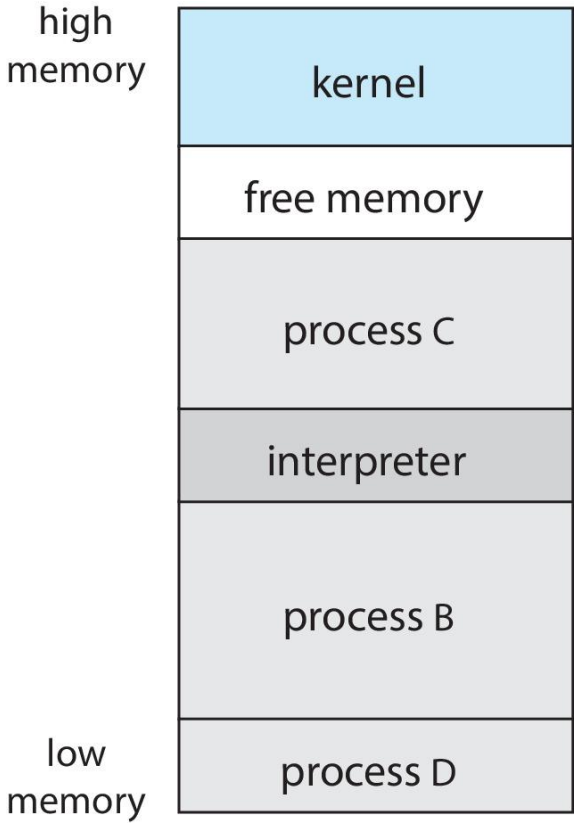


(b)

running a program

EXAMPLE: FREEBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
  - Executes exec() to load program into memory
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0 – no error
  - code > 0 – error code



Interrupts

COMPUTER-SYSTEM OPERATION

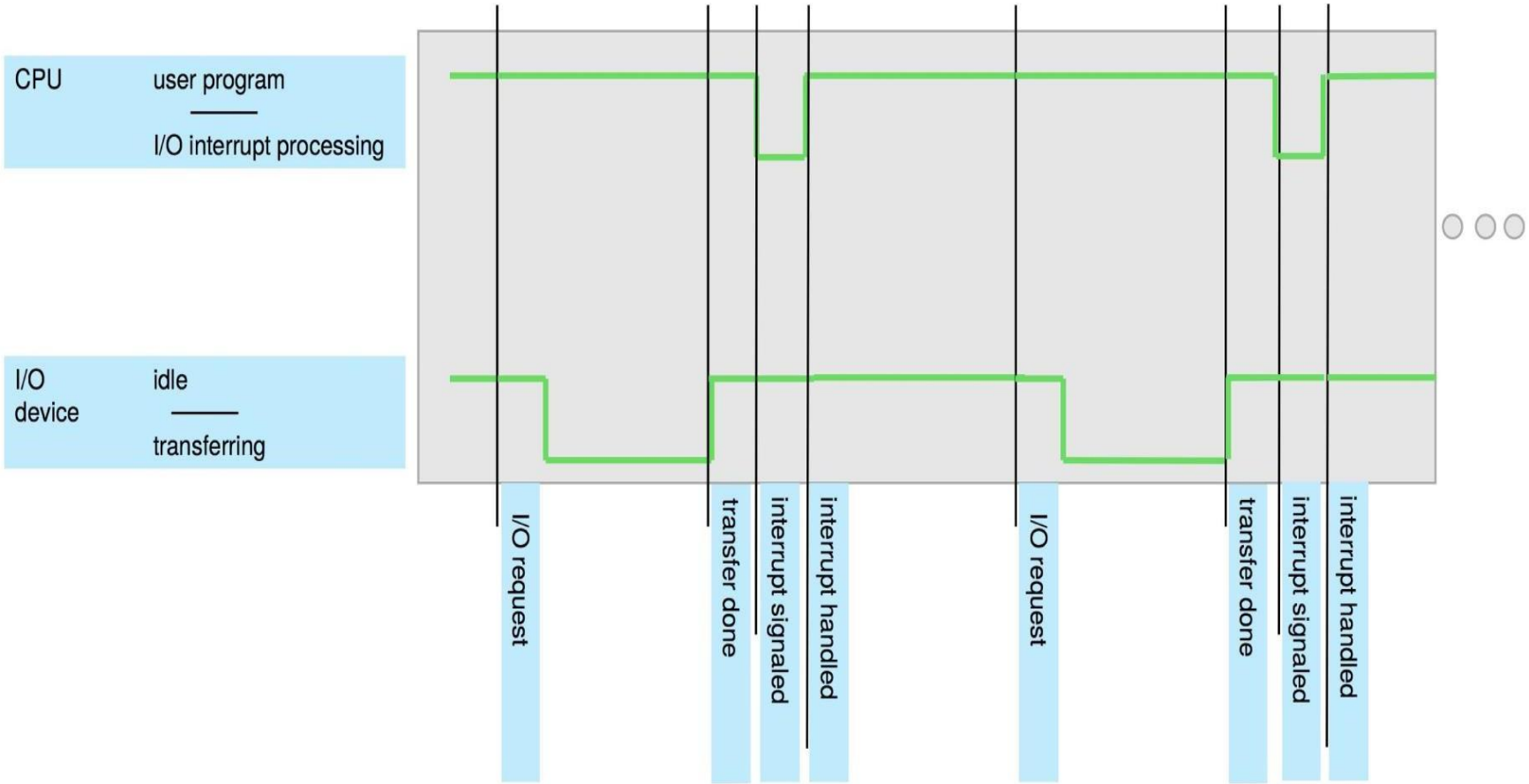
- I/O devices and the CPU can execute concurrently
- Each **device controller** is in charge of a particular device type for example, a disk drive, audio device, or graphics display)
- Each device controller has a local buffer storage and a set of special-purpose registers.
- The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.
- Each device controller type has an operating system **device driver** to manage it
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an **interrupt**

INTERRUPTS

- Interrupt is a signal sent to the CPU can be triggered by the hardware or software indicating that an event needs immediate attention.
- It temporarily halts the current program execution, saves the state, and transfers control to an interrupt service routine (ISR).
- That is, When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location.
- Device controllers and hardware faults raise interrupts.
- The fixed location usually contains the starting address where the service routine for the interrupt is located.
- The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.

INTERRUPTS

Interrupt timeline for a single program doing output



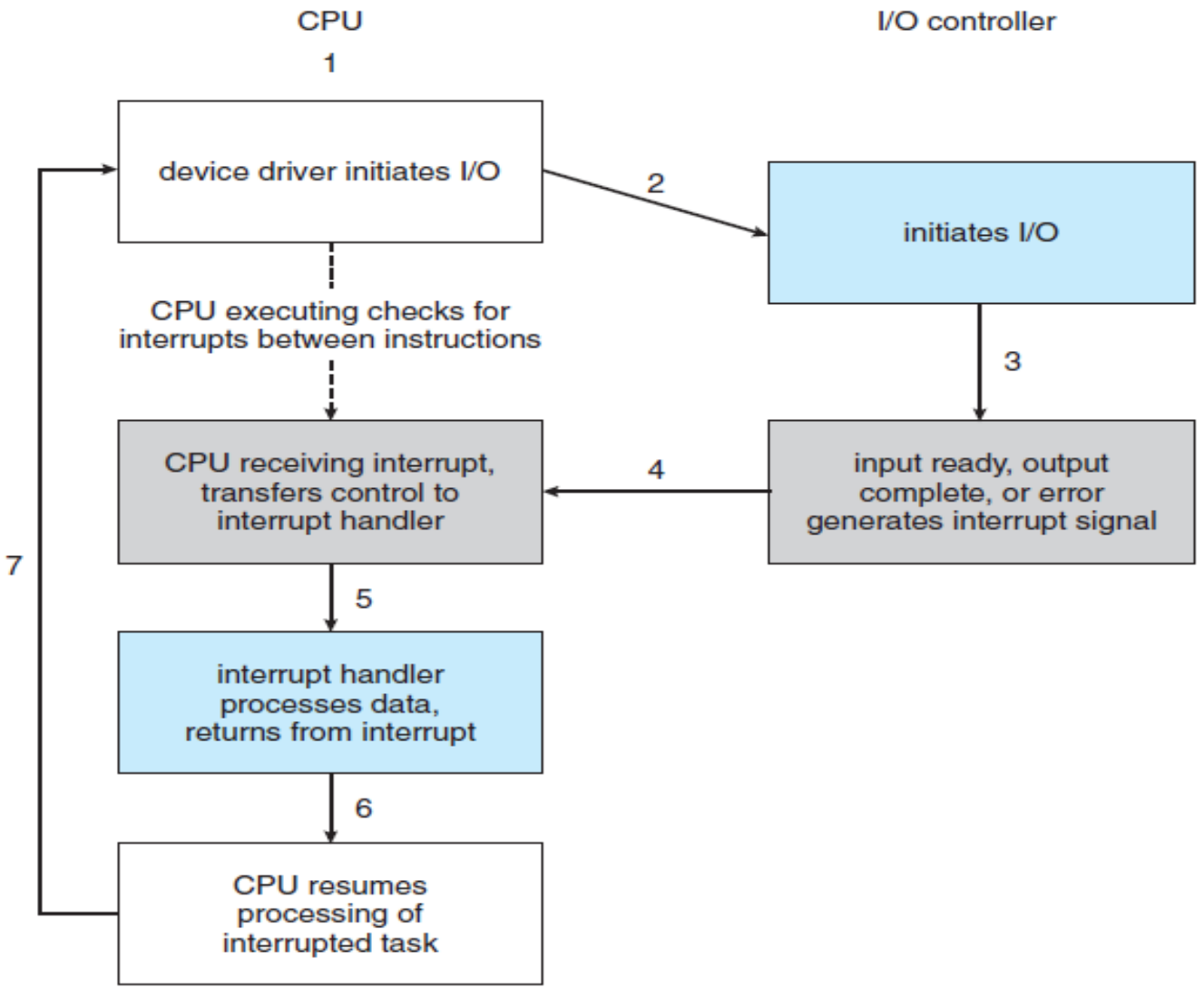
INTERRUPTS

- The interrupt must transfer control to the appropriate interrupt service routine through the **interrupt vector**, which contains the addresses of all the service routines.
- Interrupts must be handled quickly.
- Interrupt architecture must save the address of the interrupted instruction.
- After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.
- A **trap** or **exception** is a software-generated interrupt caused
  either by an error or a user request

Interrupt Implementation

- The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction.
- When the CPU detects that a controller has asserted a signal on the interrupt-request line, it reads the interrupt number and jumps to the **interrupt-handler routine** by using that interrupt number as an index into the interrupt vector.
- It then starts execution at the address associated with that index.
- The interrupt handler saves the following states:
  - determines the cause of the interrupt,
  - performs the necessary processing,
  - performs a state restore,
  - and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt.
- The device controller ***raises*** an interrupt by asserting a signal on the interrupt request line
- The CPU ***catches*** the interrupt and ***dispatches*** it to the interrupt handler, and the handler ***clears*** the interrupt by servicing the device.

Interrupt-driven I/O cycle



Types of Interrupt

- Maskable Interrupts
  - CPU can disable or "mask" the interrupts using specific instructions.
- Non Maskable Interrupts
  - These interrupts cannot be disabled or ignored by the CPU.
- Vectored Interrupts
  - These interrupts to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service.
  - This can be handled by Interrupt Chaining. Interrupt chaining is a technique where multiple devices share a single interrupt line, and the CPU identifies the source of the interrupt by passing control through a chain of devices in priority order.

Types of Interrupt

- Priority Interrupts
  - These interrupts to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service.
  - This can be handled by Interrupt Chaining. Interrupt chaining is a technique where multiple devices share a single interrupt line, and the CPU identifies the source of the interrupt by passing control through a chain of devices in priority order.
  - These levels enable the CPU to defer the handling of low-priority interrupts without masking all interrupts and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

OS Design Issues

OS Design Issues

- Operating system design issues involve the following challenges:
  - Balancing performance,
  - Reliability,
  - Scalability,
  - Security,
  - Usability while ensuring smooth interaction between hardware and software.
- The first problem in designing a system is to define goals and specifications.
- The OS must adapt to different hardware platform and user requirements.
- Processes and threads must coordinate correctly to avoid race conditions.
- OS should protect against unauthorized access and should maintain data integrity.
- Resources must be uniquely identifiable.
- OS should handle the growth in users, hardware and workload without degradation.
- Efficient resource management is essential
- OS should ensure that the system is running in all conditions.

OS Design Issues

- At the highest level, the design of the system will be affected by the choice of hardware and the type of system.
- Beyond this highest design level, the requirements may be much harder to specify.
- The requirements can be divided into two basic groups:
  - **user goals : Convenience, Simplicity and Responsiveness**
  - **system goals : Maintainability, Modularity and Portability**
- The system should be convenient to use, easy to learn and to use, reliable, safe, and fast.
- A similar set of requirements can be defined by the developers who must design, create, maintain, and operate the system.
- The system should be easy to design, implement, and maintain;
- It should be flexible, reliable, error free, and efficient.

Building and Booting an OS

Building and Booting an Operating System

- Most commonly, a computer system, when purchased, has an operating system already installed
- If you purchase a computer without an operating system, then follow the below given steps:
  - If generating an operating system from scratch
    - Write the operating system source code
    - Configure the operating system for the system on which it will run
    - Compile the operating system
    - Install the operating system
    - Boot the computer and its new operating system

Building and Booting Linux

- 1. Download the Linux source code from http://www.kernel.org.
- 2. Configure the kernel using the “make menuconfig” command. This step generates the .config configuration file.
- 3. Compile the main kernel using the “make” command. The make command compiles the kernel based on the configuration parameters identified in the .config file, producing the file vmlinuz, which is the kernel image.
- 4. Compile the kernel modules using the “make modules” command. Just as with compiling the kernel, module compilation depends on the configuration parameters specified in the .config file.
- 5. Use the command “make modules install” to install the kernel modules
- into vmlinuz.
- 6. Install the new kernel on the system by entering the “make install”
- command.
- When the system reboots, it will begin running this new operating system.

System Boot

- After an operating system 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 memory and started.
- **3.** The kernel initializes hardware.
- **4.** The root file system is mounted.

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 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.
- 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 CPUand 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.
- **GRUB** is an open-source bootstrap program for Linux and UNIX systems.

Reference

Abraham Silberschatz, Peter B. Galvin, Greg Gagne, "Operating System Concepts", Wiley, 10th Edition, 2018