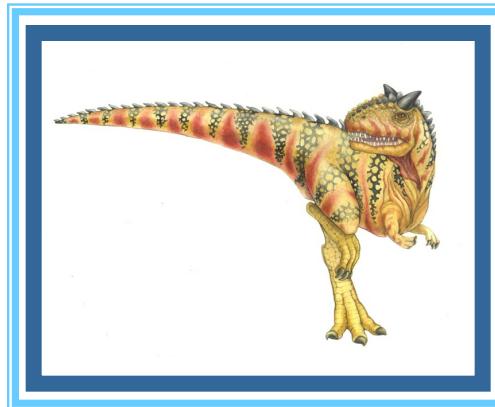


# Chapter 2: Operating-System Structures





# Chapter 2: Operating-System Structures

---

- Operating System Services
- User and Operating System Interfaces
- System Calls
- Operating system examples
- System Programs
- Linkers and Loaders
- Operating System Design and Implementation
- Operating System Structure





# Objectives

---

- Identify services provided by an operating system
- Illustrate how system calls are used to provide operating system services
- Compare and contrast **monolithic, layered, microkernel, modular, and hybrid strategies** for designing operating systems





# Operating System Services

---

- OS provides an environment for
  - Execution of programs
  - Services to programs and users
- One set of OS services - OS functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (**UI**)
    - Command-Line (CLI), Graphics User Interface (GUI), touch-screen,
  - **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** - The file system is of particular importance. Programs need to read and write files and directories, search, create and delete them, list file information, permission management.
- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via **shared memory** or through **message passing** (packets moved by the OS)
- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in CPU and memory, in I/O devices, in user programs
  - For each type of error, OS must take appropriate actions to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system





# Operating System Services (Cont.)

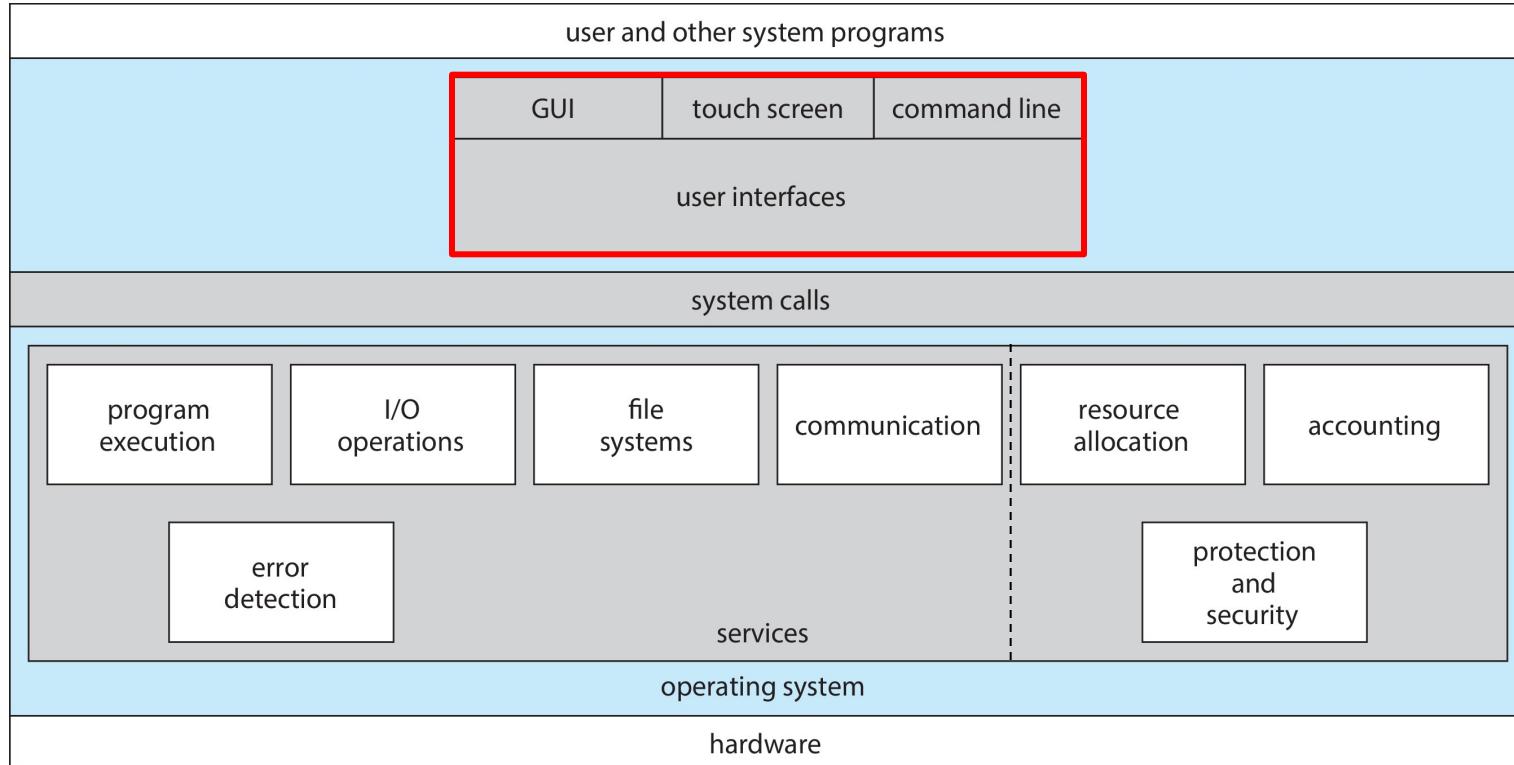
---

- Another set of OS services – OS functions for ensuring the *efficient* operation of the system itself via resource sharing
- **Resource allocation** - When multiple users or multiple jobs run concurrently, resources must be allocated to each of them
  - Many types of resources - CPU cycles, memory, file storage, I/O devices.
- **Logging** - To keep track of which users use how much and what kinds of computer resources
- **Protection** and **security** - The owners of information stored in a multiuser or networked system may want to 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, extends to defending external I/O devices from invalid access attempts





# A View of Operating System Services





# User Operating System Interface - CLI

- There are generally three approaches. One provides a [command-line interface \(CLI\)](#), or [command interpreter](#), that allows users to directly enter commands to be performed by the operating system. The other two allow users to interface with the OS via a [graphical user interface](#), or [GUI](#).

[CLI](#) or [command interpreter](#) allows direct command entry

- Sometimes implemented in kernel, sometimes by system programs
- Sometimes multiple flavors implemented – **shells** in Unix or Linux
- The primary functionality is to (1) fetch a command from user, (2) interprets it and (3) executes it
- UNIX and Linux systems provide different version of shells, such as C shell, Bourne-Again shell, Korn shell, and others





# Bourne Shell Command Interpreter

```
1. root@r6181-d5-us01:~ (ssh)
X root@r6181-d5-u... ● ⌘1 X ssh ⌘2 X root@r6181-d5-us01... ⌘3
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
06:57:48 up 16 days, 10:52, 3 users, load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem           Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                      50G   19G   28G  41% /
tmpfs                 127G  520K  127G   1% /dev/shm
/dev/sda1              477M   71M   381M  16% /boot
/dev/dssd0000          1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                      12T  5.7T  6.4T  47% /mnt/orangefs
/dev/gpfs-test         23T  1.1T   22T   5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root      97653 11.2  6.6 42665344 17520636 ?  S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root      69849  6.6  0.0     0     0 ?        S    Jul12 181:54 [vpthread-1-1]
root      69850  6.4  0.0     0     0 ?        S    Jul12 177:42 [vpthread-1-2]
root      3829   3.0  0.0     0     0 ?        S    Jun27 730:04 [rp_thread 7:0]
root      3826   3.0  0.0     0     0 ?        S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x----- 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```





# User Operating System Interface - GUI

---

- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, directories, and system functions
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC earlier 1970s, first widely use in Apple Macintosh (the first Mac released in 1984 replacing Apple II in 70s)
- Many systems now provide both CLI and GUI interfaces
  - Microsoft Windows uses GUI with CLI “command” shell
  - Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (KDE, GNOME)





# The Mac OS X GUI

The screenshot displays a Mac OS X desktop environment with several windows open:

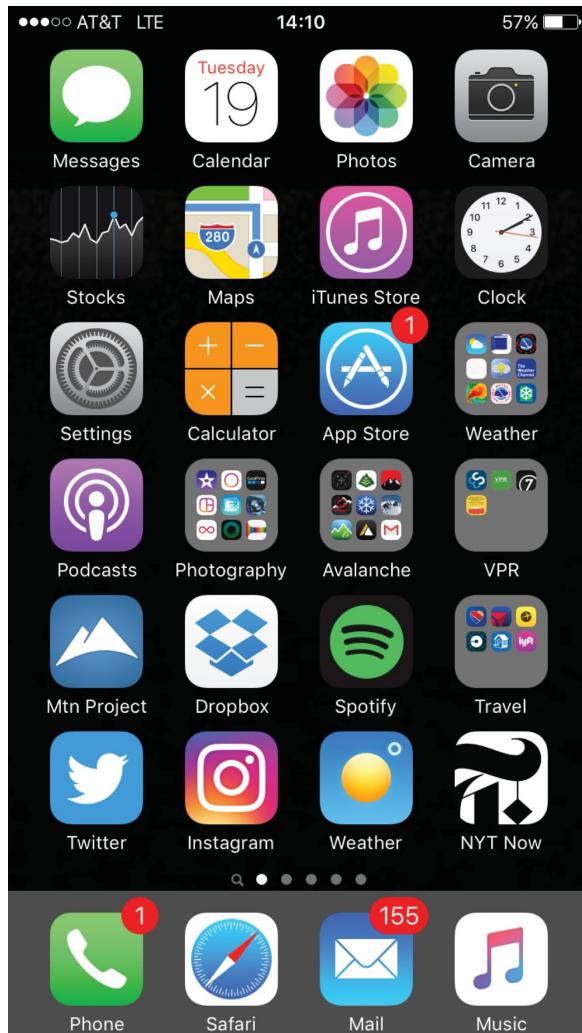
- PowerPoint Window:** Shows a presentation slide titled "PFS summary v3" with a subtitle "DSSD HIGH PERFORMANCE PARALLEL FILE SYSTEMS". The slide includes a large image of a server rack and some text.
- File Explorer Window:** Shows a file tree for "os10c-dir > txt-dir > Chapters > ch2".
- Dictionary Application:** Shows the "Shorter Oxford English Dictionary" and "The Sleeping Dictionary".
- OS X Finder Window:** Shows a list of files and folders in a folder named "os10c-dir".
- TextEdit Window:** Shows a document titled "operating system".
- Calculator Application:** Shows a numeric keypad interface.
- Address Book Window:** Shows a list of contacts.
- Mail Window:** Shows a list of messages.
- System Preferences Window:** Shows the "Sharing" pane.
- Dashboard Window:** Shows various widgets including a clock, weather, and calendar.
- Background:** A desktop background image of a Tyrannosaurus Rex.





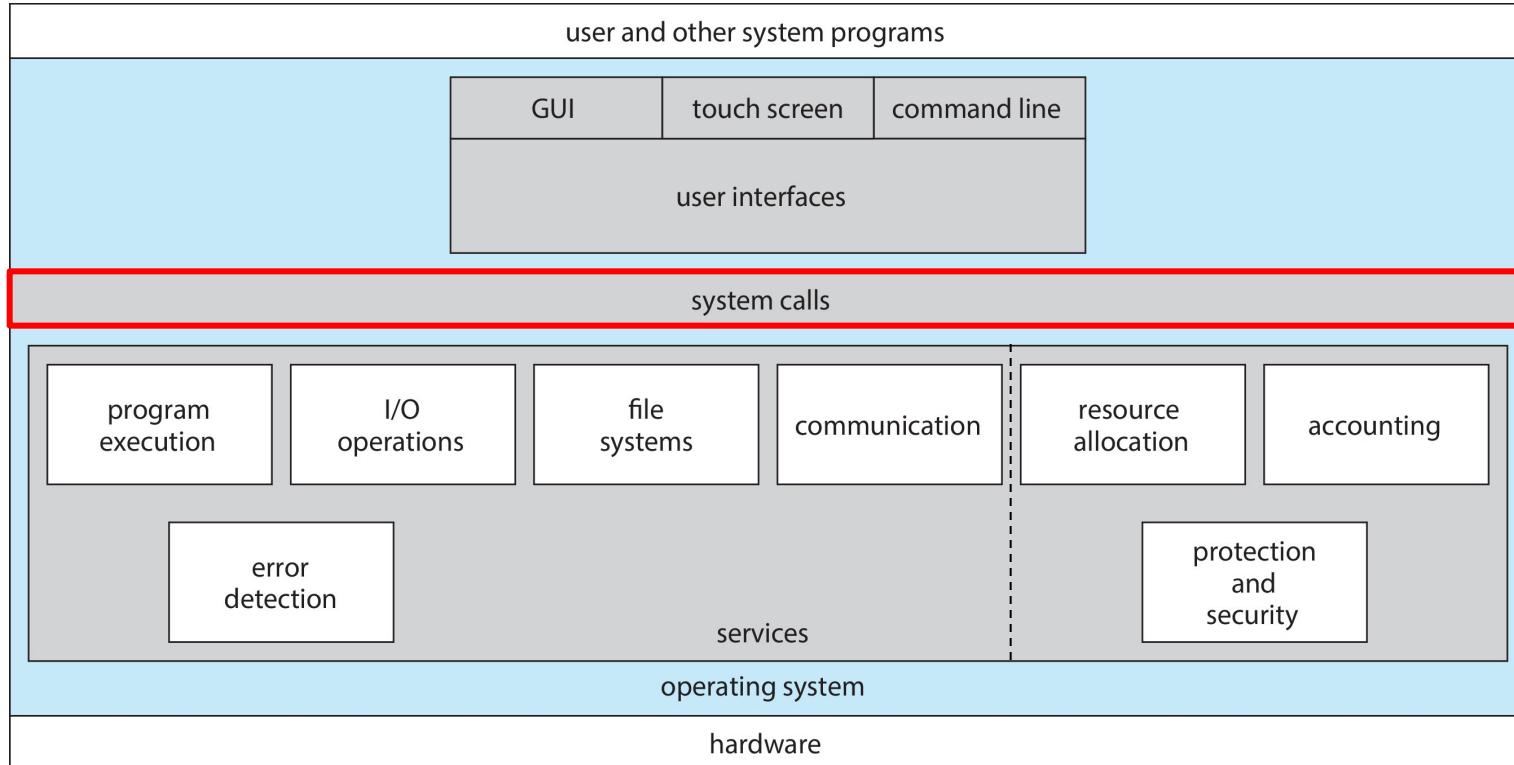
# Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not feasible or not desired
  - Actions and selection based on **gestures**
  - Virtual keyboard for text
- Voice command
  - iPhone **Siri**





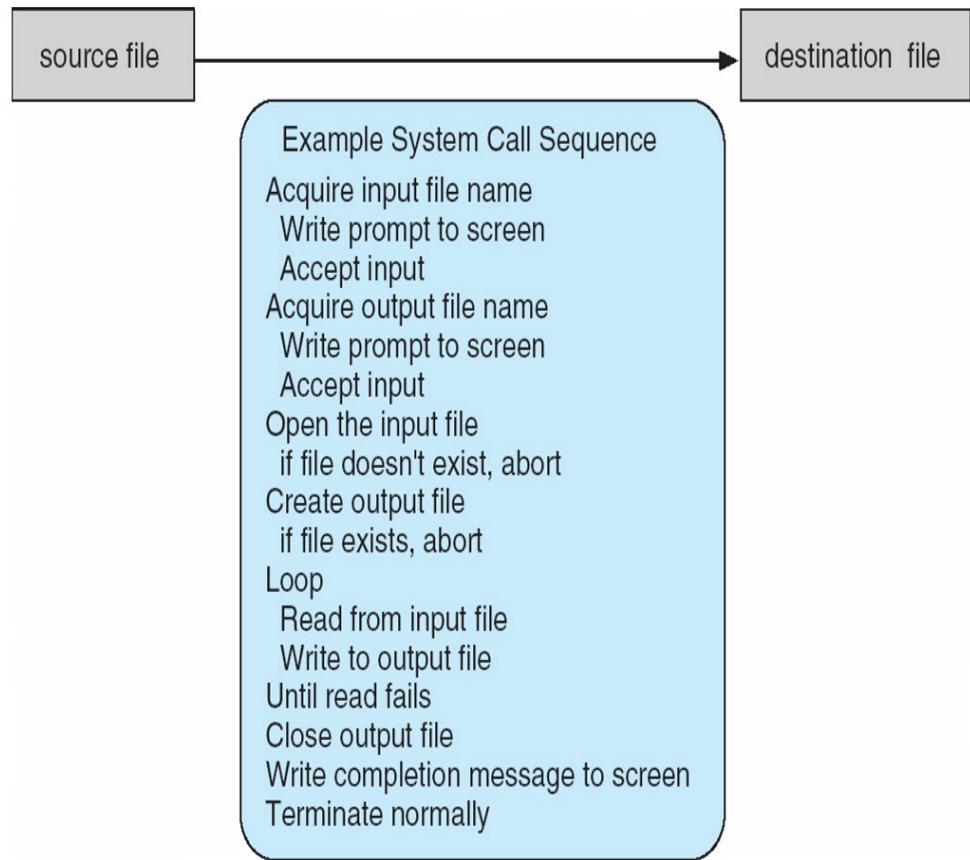
# System Calls





# System Calls (cont.)

- Programming interface to the services provided by the OS
- The system calls are generally available as functions written in a high-level language (C/C++), certain low-level tasks are written in assembly languages (accessing hardware)
- An example: cp in.txt out.txt – involves a sequence of system calls



Copying one file to another





# System Calls - API

- Application Program Interface (API) specifies a set of functions that are available for application programmers to use, including the parameters passed to the function and return values it may expect
- The functions that make up an API typically invoke the actual system calls on behalf of the application programmer
  - For example, the Windows function CreateProcess() invokes the NTCREATEPROCESS() system call in the Windows kernel
- A programmer access APIs via a library provided by the OS
  - In the case of UNIX and Linux for programs written in the C language, the library is called **libc**
- Three most common APIs for programming
  - Win32 API for Windows systems
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)





# Example of Standard API

Library header file provided by Linux

```
#include <unistd.h> ←  
  
ssize_t      read(int fd, void *buf, size_t count)  
  
return       function  
value        name                parameters
```

## Parameters

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

## Return value

- 0 indicates end of file
- -1 indicates an error has occurred





# System Calls – API (cont.)

**Advantages** - why use APIs rather than directly invoking system calls?

- **Program portability** – a programmer using an API expects the program to compile and run on any system that supports the same API – the implementation of system calls vary from machines to machines
- **Hide the complex details** of the system call from users
  - The actual system calls can often be more detailed and difficult to work with than the API available for an application programmer
  - A caller of an API (e.g., the program) need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey the API (format) and understand what the operating system will do as a result of the execution of that system call





# System Call Implementation

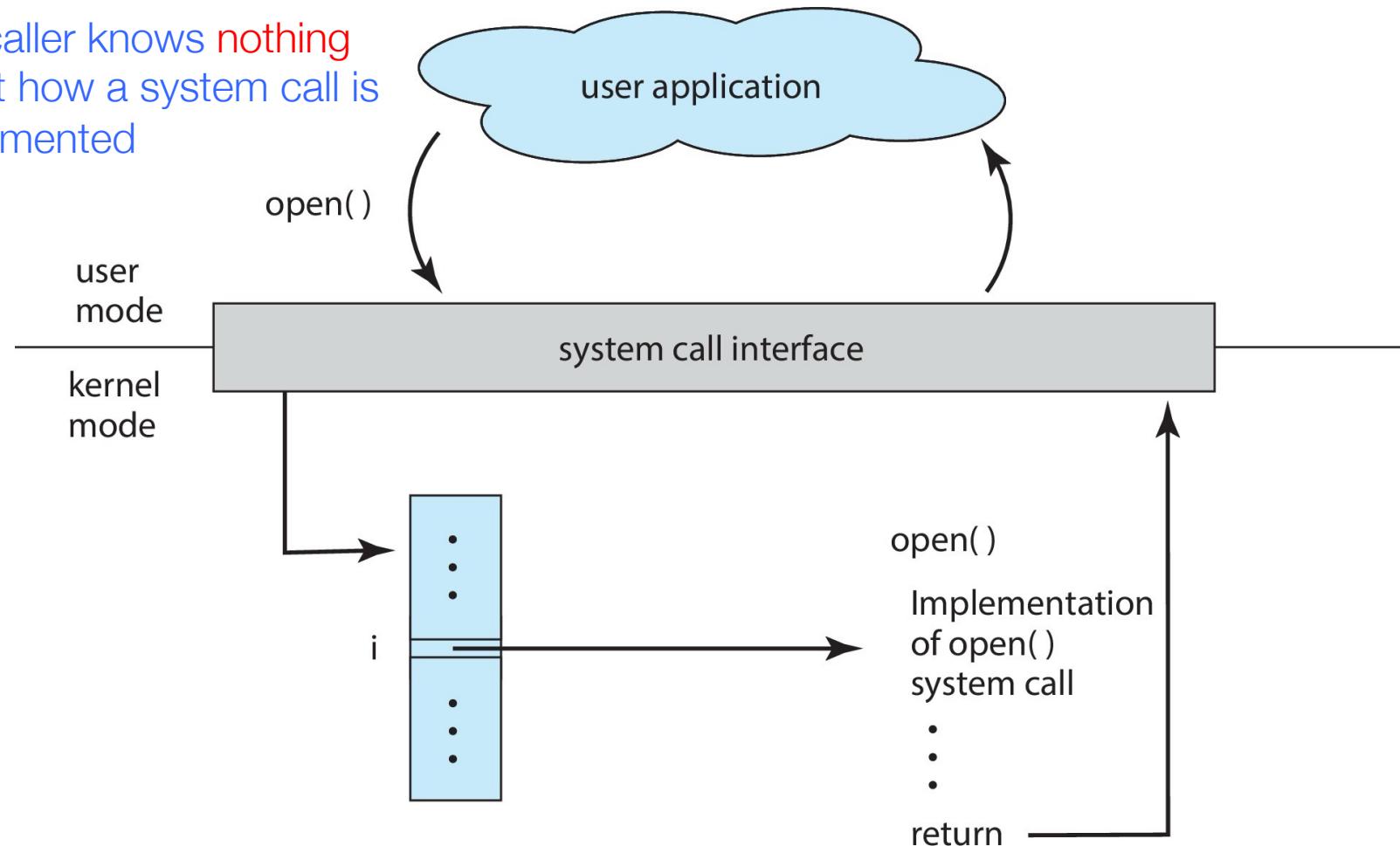
- For most programming languages, the run-time environment (RTE) (a set of functions built into libraries) provide a system call interface that serves as the link to system calls made available by the operating system
  - An identity number is associated with each system call
  - The system-call interface maintains a table indexed according to these numbers
- The system call interface - functions
  - intercept function calls in the API
  - invokes the necessary system call within the OS, and
  - returns status of the system call and return value(s), if any





# API – System Call – OS Relationship

The caller knows **nothing** about how a system call is implemented





# System Call Parameter Passing

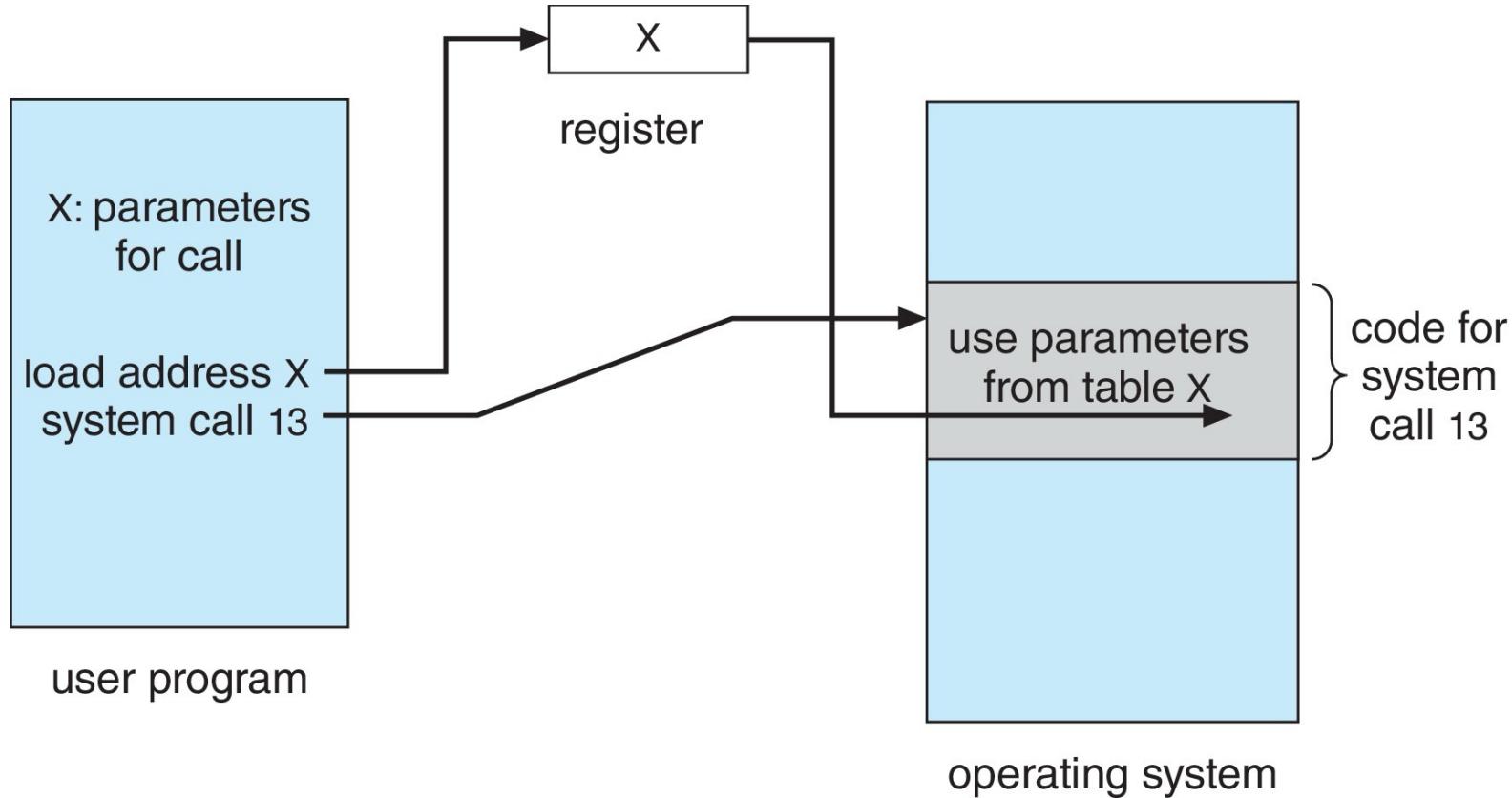
- Often, more information is required than the identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods are used to pass parameters from user programs to the OS
  - Simplest – pass the parameters in **registers**
    - ▶ In some cases, may be more parameters than registers
  - **Block method**– parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
  - **Stack method**– parameters placed, or pushed, onto the stack by the program and popped off the stack by the OS

Block and stack methods do not limit the number or length of parameters being passed





# Parameter Passing via Table





# 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
- [Shared-memory model](#) create and gain access to memory regions
- transfer status information
- attach and detach remote devices

## ■ 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.

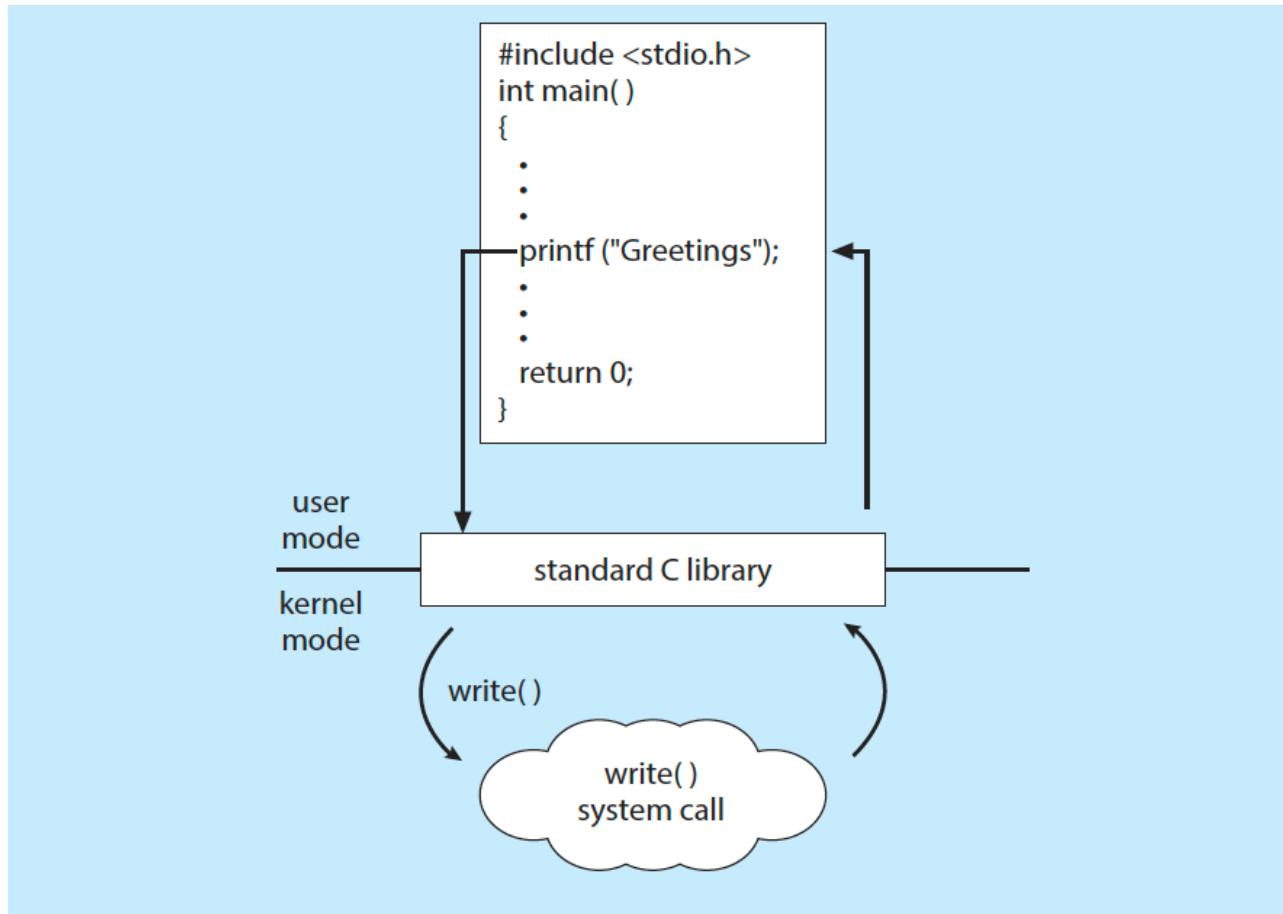
	Windows	Unix
<b>Process control</b>	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
<b>File management</b>	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
<b>Device management</b>	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
<b>Information maintenance</b>	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
<b>Communications</b>	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
<b>Protection</b>	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()





# Standard C Library Example

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





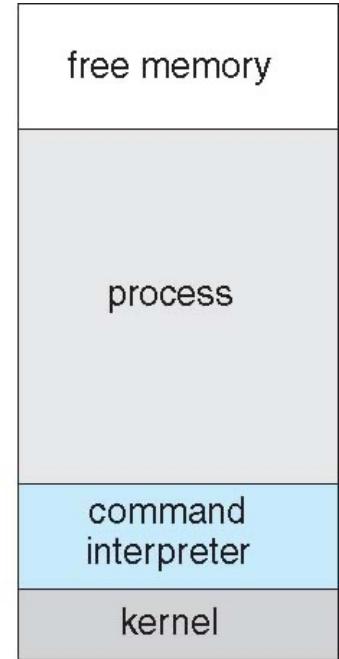
# Example: MS-DOS

- Single-tasking
- Shell invoked when system booted
- Simple to run a program
  - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded



(a)

At system startup



(b)

running a program





# System Programs

- Another aspect of a modern system is its collection of system services
- **System services**, or **system programs**, or **system utilities**, provide a convenient environment for program development and execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex
- The view of the operating system seen by most users is defined by the application and system programs, not the actual system calls
  - Considering a GUI featuring a mouse-and-windows interface, and a command line UNIX shell. Both essentially use the same set of system calls, but system calls appear very differently and act in different ways
    - user view





# System Programs (Cont.)

## ■ File management

- Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

## ■ Status information

- Some ask the system for the date, time, amount of available memory, disk space, number of users
- Others provide detailed performance, logging, and debugging information

## ■ File modification

- Text editors to create and modify files
- Special commands to search file contents or perform transformations of the text





# System Programs (Cont.)

## ■ Programming-language support

- Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and Python) are often provided

## ■ Program loading and execution

- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language

## ■ Communications - provide the mechanism for creating virtual connections among processes, users, and computer systems

- Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another





# System Programs (Cont.)

---

## ■ Background services

- Launch at boot time
  - ▶ Some of these processes terminate after completing their tasks
  - ▶ Some continue to run until the system is halted, often known as **services**, **subsystems**, or **daemons**
- Provide facilities like disk checking, process scheduling, error logging

## ■ Application programs

- Not part of the operating system
- Launched by command line, mouse click, finger poke
- Examples include web browsers, word processors, text formatters, spreadsheets, database systems, and games.





# Linkers and Loaders

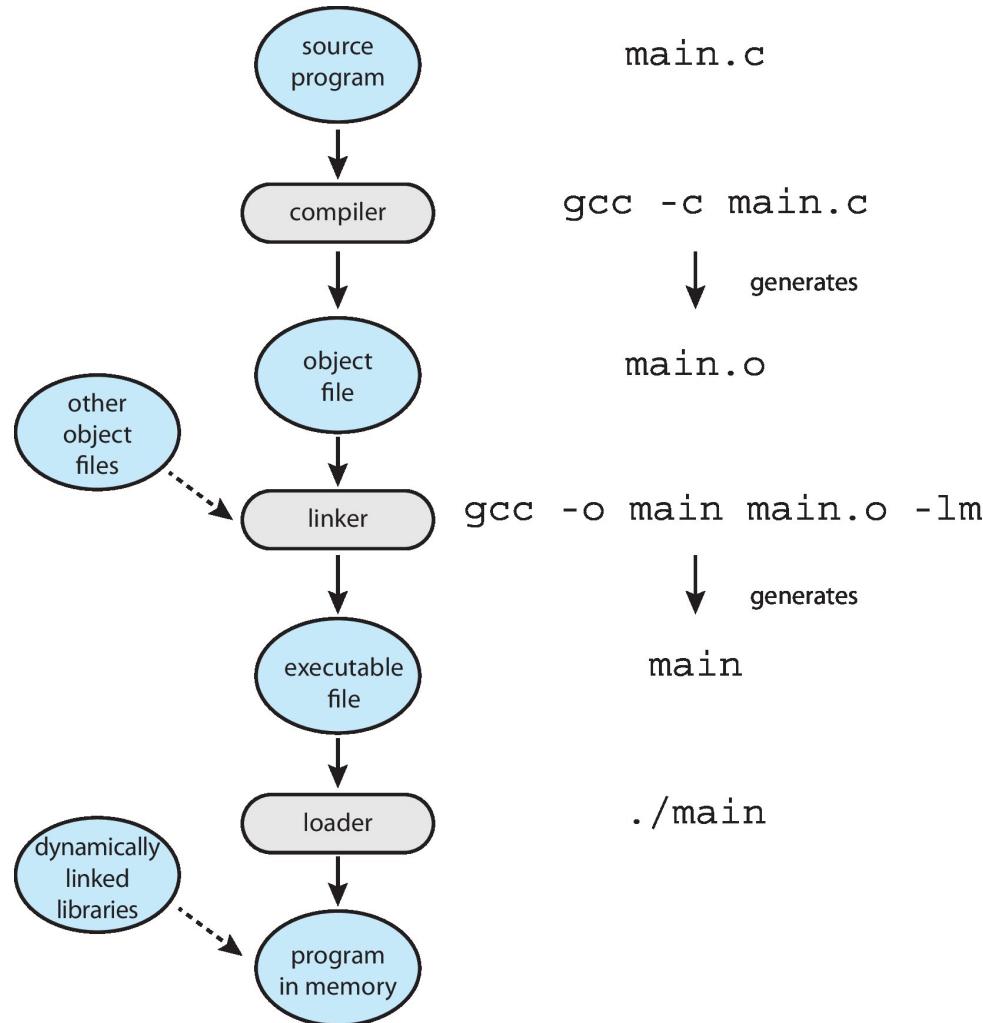
---

- Source codes are **compiled** into object files designed to be loaded into any physical memory location – **relocatable object files**
- **Linker** combines these object files into a single binary **executable file**, along with **libraries** if needed
- Programs reside on the secondary storage as **binary executables**, which must be brought into memory by **loader** to be executed
  - **Relocation** assigns final addresses to program parts and adjusts code and data in program to match those addresses (to be discussed later)
- Modern general-purpose operating systems don't link libraries into executables statically
  - Rather, **dynamically linked libraries** (in Windows, **DLLs**) are loaded when needed, which are shared by all programs that use the same version of that same library (loaded only once)
- Object files, executable files have standard formats (machine codes and symbolic tables), so operating system knows how to load and start them





# The Role of the Linker and Loader





# Protection

---

- **Protection** is an internal problem, **security**, in contrast, must consider both the system and the environment— within which the system is used.
- OS must protect itself from user programs
  - **Reliability**: compromising the OS generally causes it to crash
  - **Security**: limit the scope of what processes can do
  - **Privacy**: limit each process to the data it is permitted to access
  - **Fairness**: each should be limited to its appropriate share of system resources
- System protection features are guided by the principle of need-to-know and implement mechanisms to enforce the **principle of least privilege**.
- Computer systems contain objects that must be protected from misuse. Objects may be hardware (such as memory, CPU time, and I/O devices) or software (such as files, programs, and semaphores).





# Operating System Design and Implementation

---

Design and Implementation of OS not “solvable”, but some approaches have proven to be successful

- Internal structure of operating systems varies widely
- At the high-level, the design is affected by choice of hardware, type of the system (e.g., desktop/laptop, mobile, distributed system, or real-time)
- Start the design by defining goals and specifications
  - **User goals**: operating system should be convenient to use, easy to learn and to use, reliable, safe, and fast
  - **System goals**: operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Specifying and designing an OS is a highly creative task of **software engineering**





# Operating System Design and Implementation (Cont.)

---

- Important principle to separate
  - Policy: **What** will be done?
  - Mechanism: **How** to do it?
- Mechanisms specify how to do things; policies decide what will be done
  - A 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 from mechanism is a very important principle, it allows **flexibility** if policy decisions are changed, and they do
  - If properly separated, it can be used either to support a policy decision that I/O-intensive programs have priority over CPU-intensive ones or to support the opposite policy.
- In a nutshell, an operating system is designed with specific goals in mind. These goals ultimately determine the operating system's policies. An operating system implements these policies through specific mechanisms.





# Implementation

---

- Early OSes written in assembly languages
- Now, most are written in higher-level languages such as C or C++
  - The lowest levels of the kernel might still be in assembly languages.
  - More than one higher level language is often used. Most Android system libraries are written in C or C++, and its application frameworks are written mostly in Java
- The advantage of using high-level languages are **easier to port** to other hardware
  - The code can be written faster, is more compact, and is easier to understand and debug
- The only possible disadvantages are reduced speed and increased storage requirements – not a major issue in today's computer systems



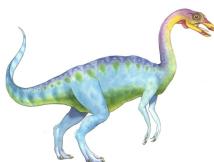


# Operating System Structure

---

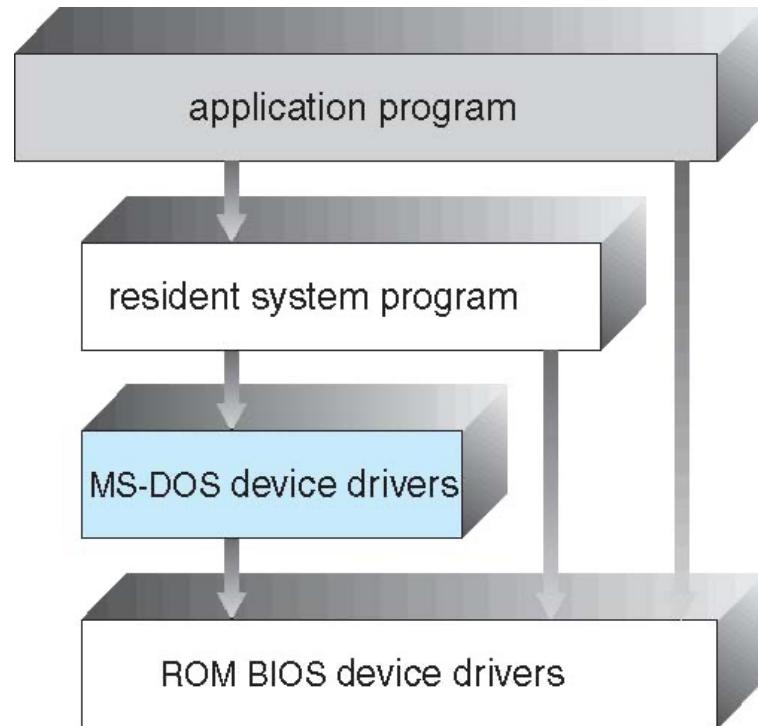
- General-purpose OS is a very large program
- Various ways to structure ones
  - **Monolithic** structure
  - **Layered** – a specific type of modular approach
  - **Microkernel** – Mach OS
  - **Loadable kernel modules** or **LKMs** – modular approach





# Simple Structure

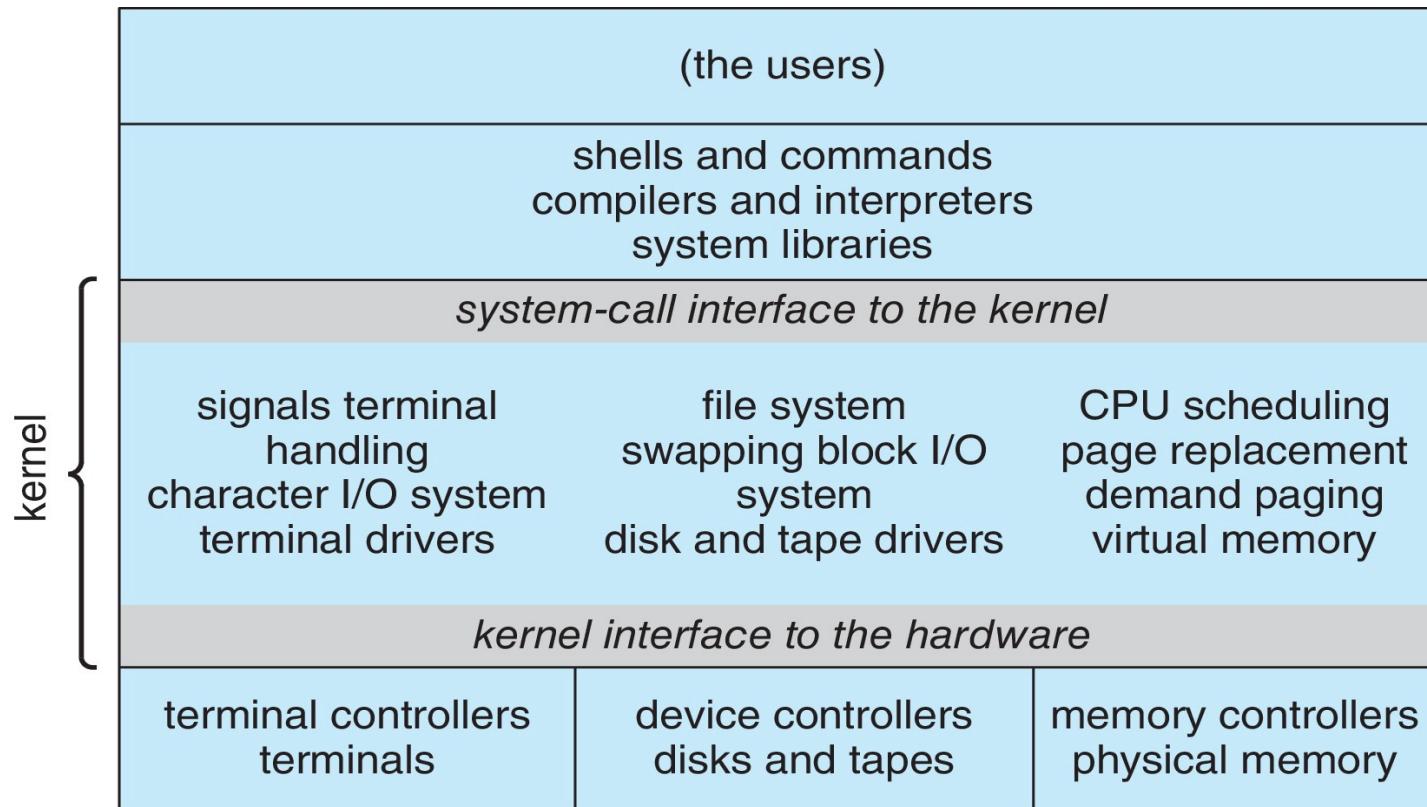
- Such OSes do not have well-defined structures, usually started as a small, simple and limited system
- MS-DOS – written to provide the most functionality in the least space
  - Not carefully divided into modules
  - Although it has some structure, interfaces and levels of functionality are not well separated – i.e., app programs can access I/O directly
  - Written for Intel 8088 with no dual mode (kernel vs. user modes) and no hardware protection





# Monolithic Structure – Original UNIX

- In traditional UNIX, the **kernel** consists of everything below the system-call interface and above the physical hardware





# Monolithic Structure – Original UNIX

---

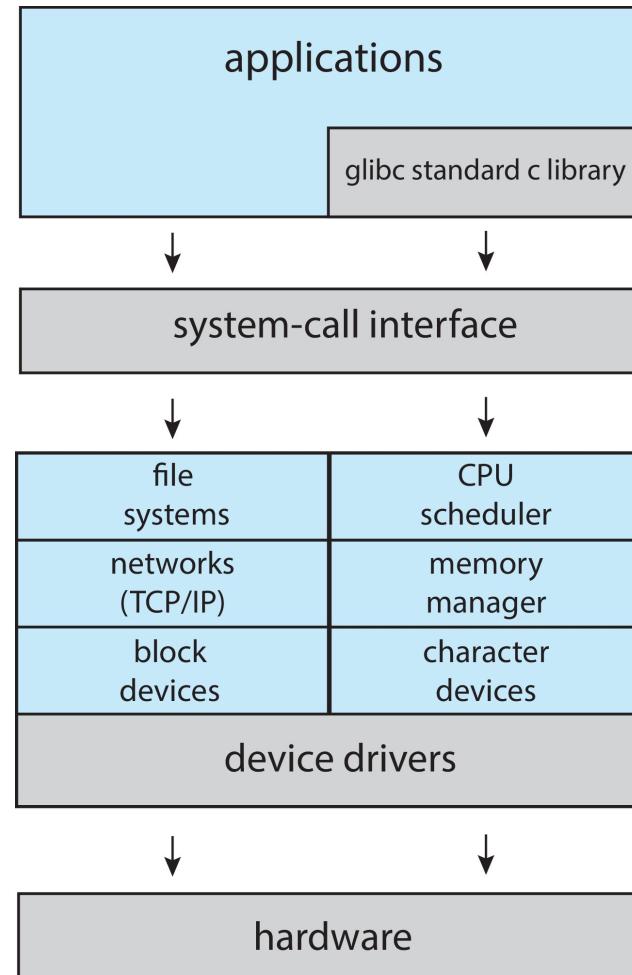
- UNIX – initially limited by hardware functionality, the original UNIX operating system had limited structuring
- Place all functionality of the kernel into a single, static binary file that runs in a single address space - known as a **monolithic structure**
- The UNIX OS consists of two separable parts: **the kernel** and **system programs**
  - The kernel is further separated into a series of interfaces and device drivers, which have been expanded considerably over the years as UNIX evolves
- The drawback is that enormous amount of functionalities are combined to one level, making it difficult to implement, debug and maintain
- **Monolithic kernels** have a distinct performance **advantage**, as there is very little overhead in the system-call interface, and communication within the kernel is fast
  - The speed and efficiency – still used in UNIX, Linux, and Windows





# Linux System Structure

- The Linux operating system is based on UNIX and is structured similarly
- The Linux kernel is **monolithic** in that it also runs entirely in kernel mode in a single address space
- Applications use the **glibc** standard C library when communicating with the system call interface to the kernel
- It has a **modular** design that allows the kernel to be modified during run time (**LKM** to be discussed)





# Modular Design

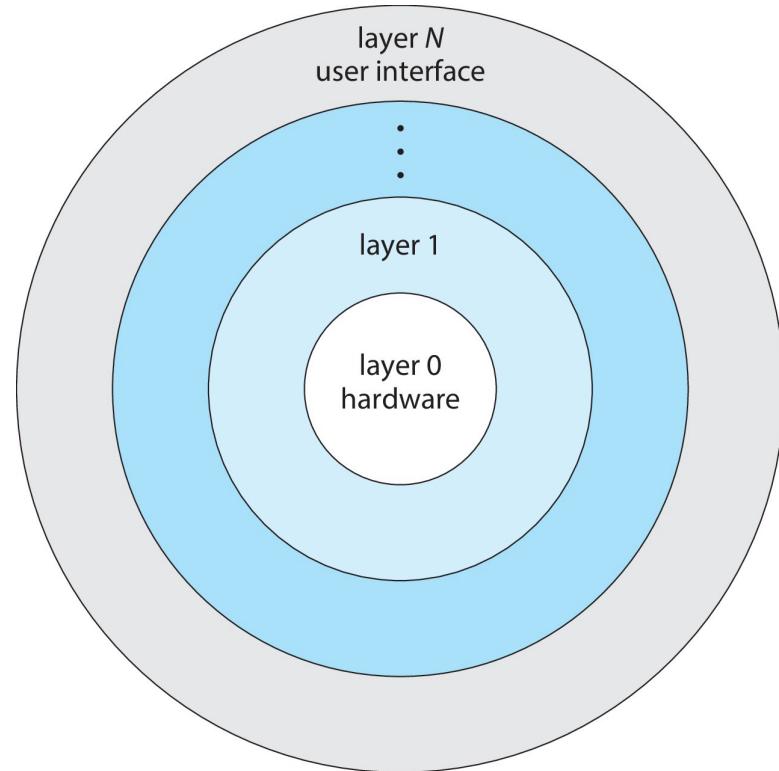
- The **monolithic approach** often known as a tightly coupled system as changes to one part of the system affect other parts – essentially one big program running in **one address space** (to be discussed in Chapters 3-4)
- Design a loosely coupled system, which is divided into separate, smaller components, each having specific and limited functionalities
  - The advantage of the modular approach is that changes in one component affect only that component, and no others, allowing more freedom and flexibility in creating and modifying each component
- A system can be made modular in many ways. One method is the **layered approach**





# Layered Approach

- The OS can be divided into a number of layers, each built on top of lower layers. The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.
- Each layer consists of data structures and a set of functions that can be invoked by higher-level layers, in turn, can invoke operations on lower-level layers
- In a nutshell, each layer utilizes the services from a lower layer (if any), provides a set of functions, and offers certain services for a higher layer (if any)





# Layered Approach (Cont.)

- **Information hiding** : a layer does not need to know how the lower-layer operations are implemented, only what these operations do - **interface**
  - Each layer hides the existence of its own data structures, operations, and hardware from higher-level layers
- The **main advantage** of a layered approach is the simplification of the construction and debugging – starting from the lowest layer
  - Layered systems have been successfully used in many other software systems such as computer networks (e.g., TCP/IP/link-physical) and web applications
- However, few operating systems use a pure layered approach because
  - There are significant challenges in appropriately defining different layers and their respective functionalities
  - The overall performance can be affected due to the overhead of requiring a program to traverse through multiple layers to obtain a service
- The trend is to have fewer layers with more functionality and modularized code, while avoiding the problems of complex layer definition and interaction

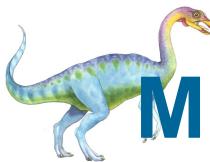




# Microkernel System Structure

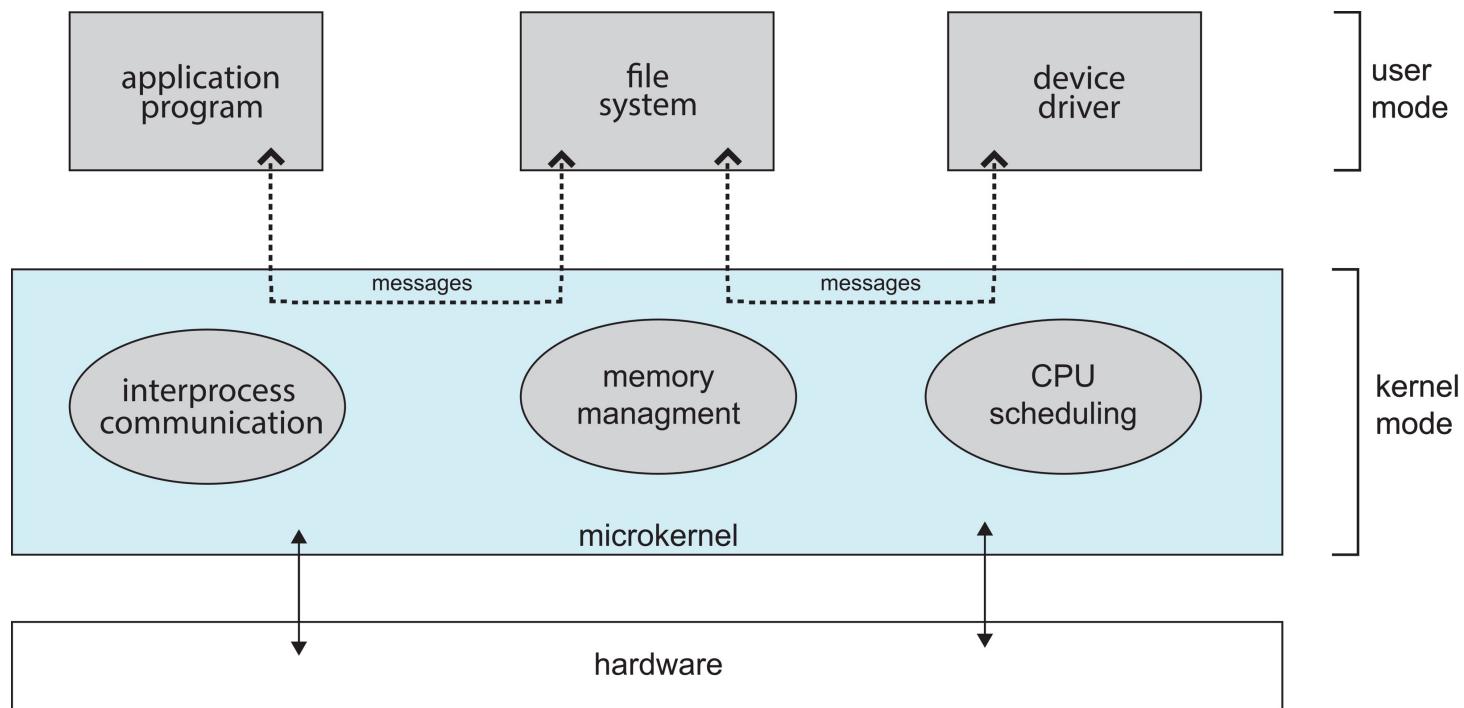
- The kernel became large and difficult to manage
- Removing all **nonessential** components from the kernel and implementing them as system or user-level programs in **separate address space**
- This results in a much smaller kernel
- **Mach** developed at CMU in mid-1980s, modularized the kernel using the **microkernel approach**
  - The best-known microkernel operating system is **Darwin**, used in Mac OS X and iOS. It consists of two kernels, one of which is the Mach microkernel
- There is little consensus regarding which services should remain in the microkernel and which should be implemented in user space, but typically, microkernels provide minimal **process** and **memory management**, in addition to a **communication facility**





# Microkernel System Structure (Cont.)

- One main function of microkernel is to provide communications between programs and various services running in **user address space** through **message passing**
- For example, if an application program wishes to access a file, it must interact with the file server. The program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.





# Microkernel System Structure (Cont.)

## ■ Advantages:

- Easier to extend a microkernel operating system, as all new services are usually added to user space without modification on the kernel
- When the kernel must be modified, the changes are fewer, as the kernel is much smaller
- Easier to port the operating system to new architectures (hardware)
- More secured and reliable (less code is running in kernel mode), since most services are running as user processes – not kernel processes. If a service fails, the rest of the OS remains untouched

## ■ Drawbacks:

- The performance of microkernels can [greatly suffer](#) due to increased system-function overhead, user space to kernel space communication
  - When two user-level services must communicate, messages must be copied between the services, which reside in separate address space
- Window NT had a layered microkernel, performance worse than Window 95, more monolithic (moving function to the kernel) with Window XP





# Modular Approach

---

- The best current methodology in OS design involves using **loadable kernel modules** or 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.
- The main idea is for the kernel to provide core services, while other services are implemented and added dynamically, when kernel is running
- Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the entire kernel every time a change was made
  - For example, the kernel has CPU scheduling and memory management algorithms into the kernel and then add support for different file systems by way of loadable modules





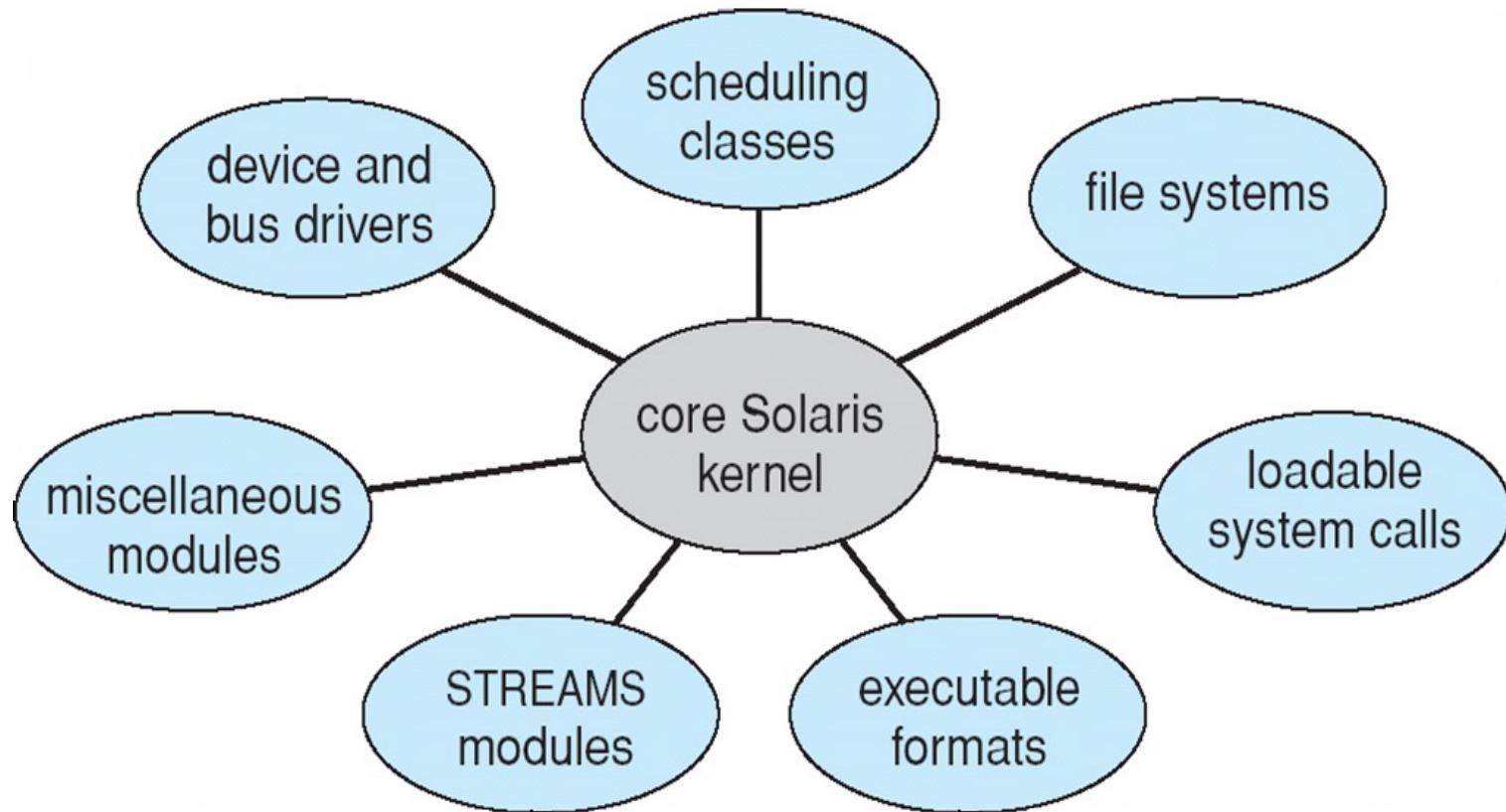
# Modular Approach (Cont.)

- This resembles a layered design in that each kernel section has a well-defined, protected interface, but it is more flexible as any module can call any other module
- It also resembles a microkernel in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but this is more efficient than a microkernel design because modules do not need to invoke message passing in order to communicate
- Linux uses loadable kernel modules, primarily for supporting device drivers and file systems.
- **LKMs** can be “inserted” into the kernel when system is **booted** or during run time, can also be removed from the kernel during run time as well
  - For example, a USB device is plugged into a running machine. If the Linux kernel does not have the necessary driver, it can be dynamically loaded.
- This allows a dynamic and modular kernel, while maintaining the performance benefits of a **monolithic** system for efficiency





# Solaris Modular Approach





# Hybrid Systems

- In practice, very few OSes adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems
- Hybrid systems combine multiple approaches to address performance, security, and usability needs
- Linux is *monolithic*, because OS in a single address space provides efficient performance. It is also *modular*, so that new functionality can be dynamically added to the kernel
- Windows is largely *monolithic*, but it retains some behaviour typical of *microkernel* systems by providing support for separate subsystems (known as *personalities*) that run as user-mode processes. Windows systems also provide support for dynamically loadable kernel modules





## Summary in Operating System Design

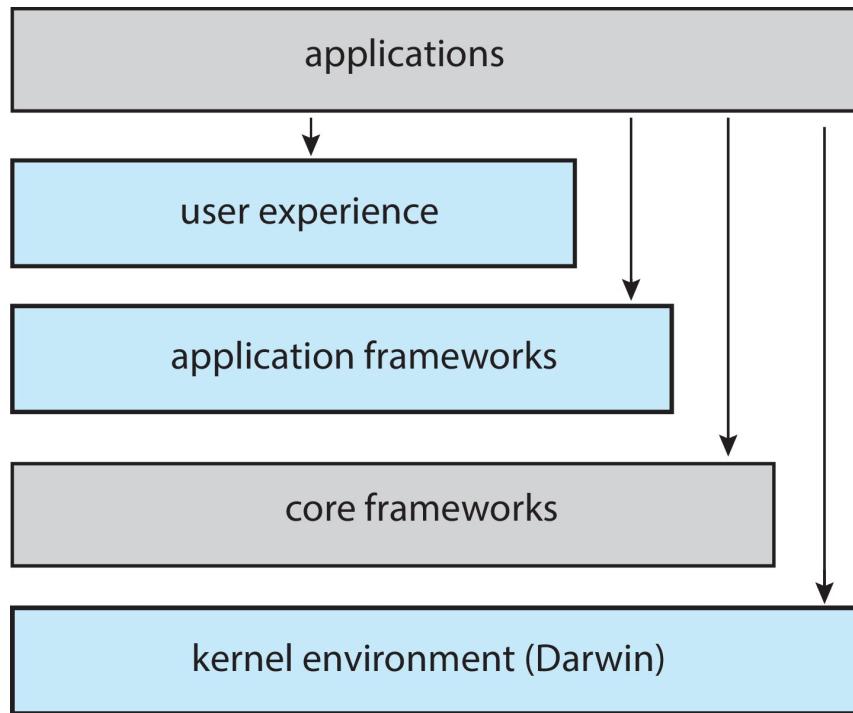
- A **monolithic** operating system has no structure; all functionality provided in a single, static binary file that runs in a **single address space**. Although such systems are difficult to modify, their primary benefit is efficiency.
- A **layered** operating system is divided into a number of discrete layers, where the bottom layer is the hardware interface, and the highest layer is the user interface. Although layered software systems have had some success, it is generally not ideal for designing operating systems due to performance problems and difficulty in defining appropriate layers.
- The **microkernel** approach for designing operating systems uses a **minimal kernel**; most services run as user-level applications, in which communication takes place via **message passing**.
- The **loadable kernel module** approach for designing operating systems provides operating-system services through **modules** that can be loaded and removed during runtime.
- Many contemporary operating systems are constructed as **hybrid systems** using a combination of a monolithic kernel and modules.





# macOS and iOS Structure

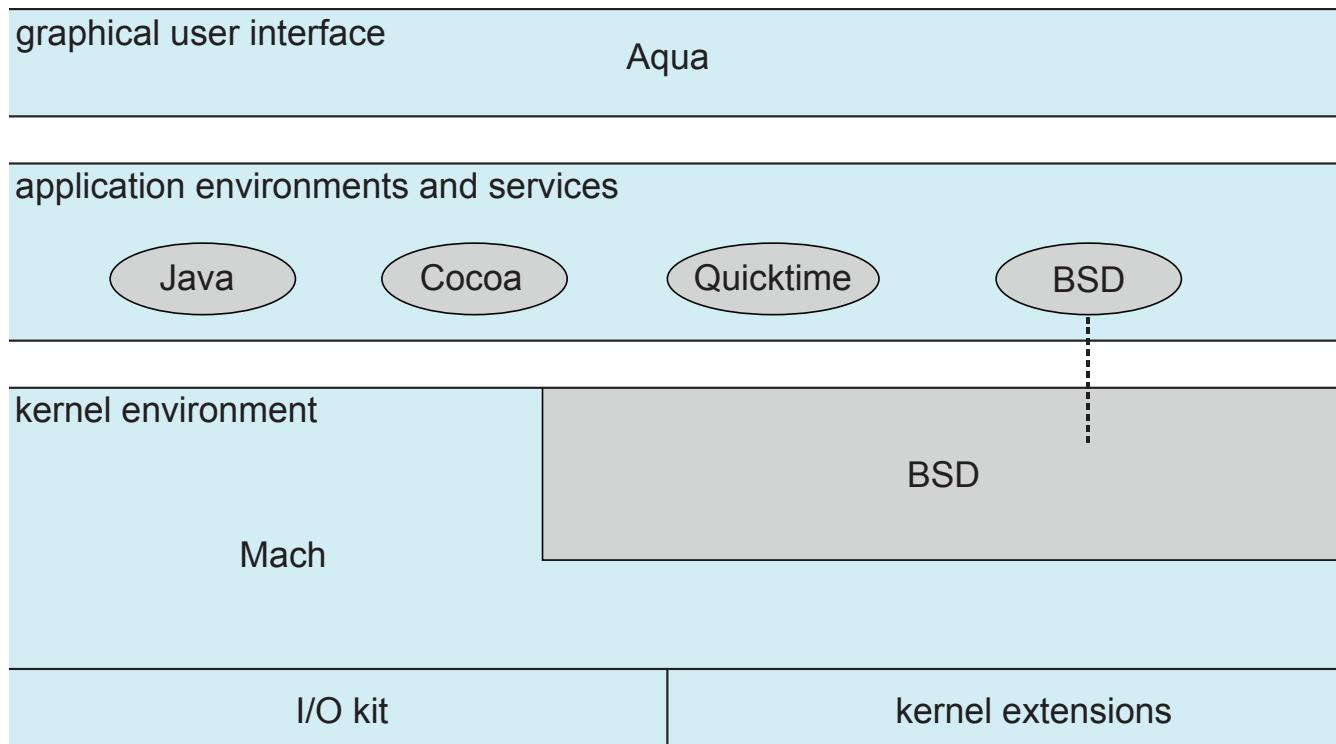
- Architecturally, macOS (for desktop and laptops) and iOS (iPhone and iPad) have much in common – user interface, programming (language) support, graphics and media, and kernel environment - Darwin includes the Mach microkernel and the BSD UNIX kernel





# Mac OS X Structure

- Apple Mac OS X is hybrid, layered, [Aqua](#) UI (for a mouse or trackpad), plus [Cocoa](#) programming environment providing an API for the Objective-C
- Core frameworks support graphics and media including, Quicktime and OpenGL
- The kernel environment, also known as [Darwin](#) includes Mach microkernel and BSD Unix





# iOS

Apple mobile OS for iPhone, iPad

- Structured on Mac OS X, added functionality
- Does not run Mac OS X applications natively
  - Also runs on different CPU architecture (ARM vs. Intel)
- **Springboard** user interface, designed for touch devices.
- **Cocoa Touch** Objective-C API for developing apps on mobile devices (touch screen)
- **Media services** layer for graphics, audio, video – Quicktime, OpenGL
- **Core services** provides cloud computing, databases
- Core operating system, based on Mac OS X kernel

Cocoa Touch

Media Services

Core Services

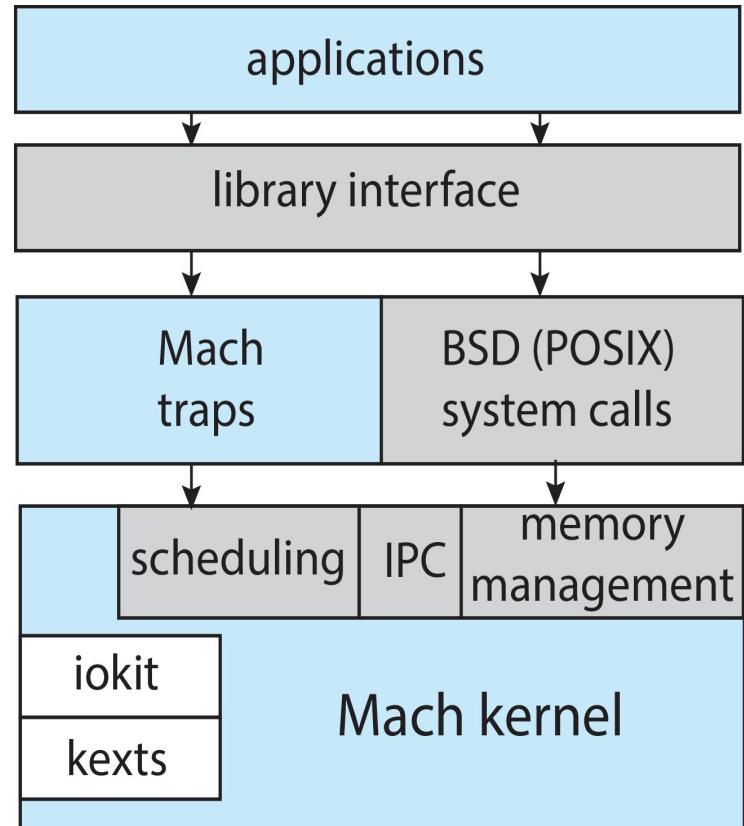
Core OS





# Darwin

- A layered system that consists of the Mach microkernel and the BSD UNIX kernel – a hybrid system
- Two system-call interfaces - **Mach** system calls (known as **traps**) and **BSD** system calls (which provide POSIX functionality)
- The interface is a rich set of libraries - the standard C library, and libraries supporting networking, security, and programming language
- Mach provides fundamental OS services, including memory management, CPU scheduling, and IPC facilities
- The kernel environment provides an I/O kit for device drivers and dynamically loadable modules refers to as **kernel extensions**, or **kexts**





# Android

---

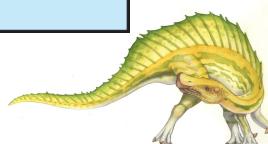
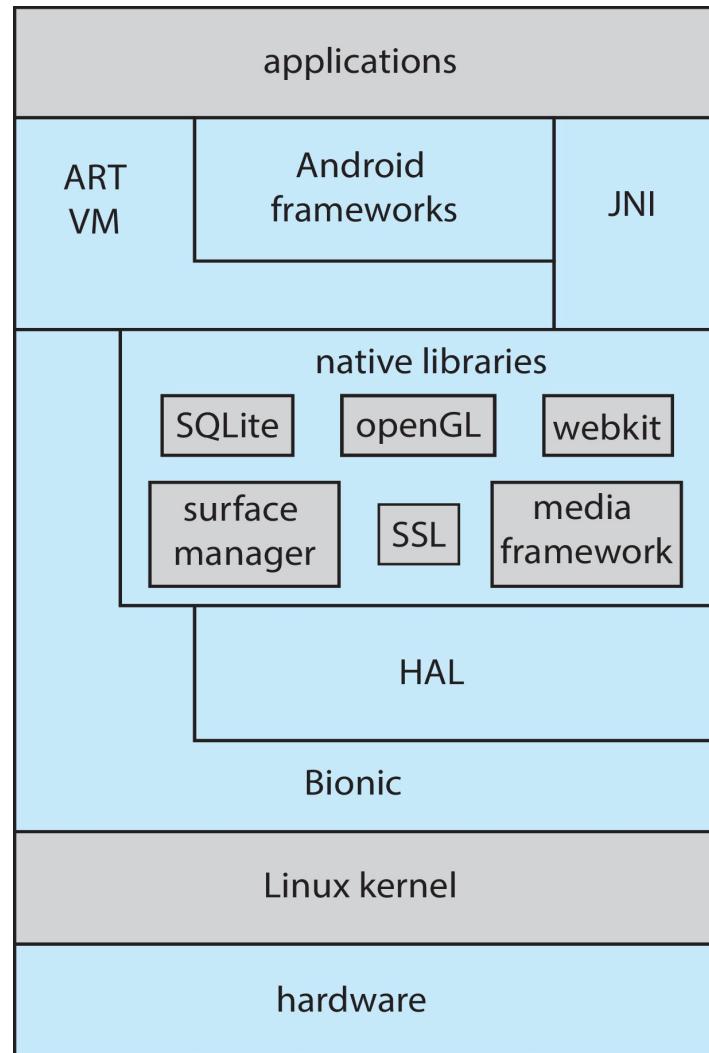
- Developed by Open Handset Alliance (led primarily by Google)
- Android runs on a variety of mobile platforms and is [open-sourced](#), in contrast iOS runs on Apple mobile devices and is [closed-sourced](#)
- Android is similar to iOS - a layered system that provides a rich set of frameworks supporting graphics, audio, and hardware features.
- Instead of using standard Java API, Google designed a separate Android API for Java development. Java applications execute on the Android RunTime or ART, a virtual machine optimized for mobile devices with limited memory and CPU processing capabilities
- Java native interface or [JNI](#), which allows developers to bypass the virtual machine and write Java programs that can access specific hardware features.
  - Programs written using JNI are generally not portable from one hardware to another.





# Android Architecture

- The libraries include frameworks for developing web browsers (webkit), database support (SQLite), and network support, such as secure sockets (SSLs).
- For Android to run on any hardware devices, **hardware abstraction layer**, or **HAL** abstract all hardware, e.g., camera, GPS chip, and other sensors, and provides applications with a consistent view independent of specific hardware
- Google developed the **Bionic** standard C library for Android, instead using standard GNU C library (**glibc**) for Linux systems
- The modified Linux kernel for mobile systems, including power management.



# End of Chapter 2

