

Chapter 2: Operating-System Structures





Chapter 2: Operating-System Structures

In this chapter, we explore all three aspects of operating systems,

- 1) Services that the system provides
- 2) An interface that makes it available to users and programmers
- 3) Its components and their interconnections

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Structure
- System Boot





Objectives

- To describe the services an operating system provides to users, processes, and other systems
- To discuss the various ways of structuring an operating system





Operating System Services

There are 9 Operating System Services

- 1. User Interface**
- 2. Program execution**
- 3. I/O Operations**
- 4. File System Manipulation**
- 5. Communications**
- 6. Error Detection**
- 7. Resources Allocation**
- 8. Accounting**
- 9. Protection and Security**





Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
 - **User interface** - Almost all operating systems have a user interface (**UI**).
 - ▶ Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **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.)

- One set of operating-system services provides functions that are helpful to the user (Cont.):
 - **File-system manipulation** - The file system is of particular interest. Programs need to read and write **files** and **directories**, create and delete them, search 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 the CPU and memory hardware, in I/O devices, in 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 efficiently use the system





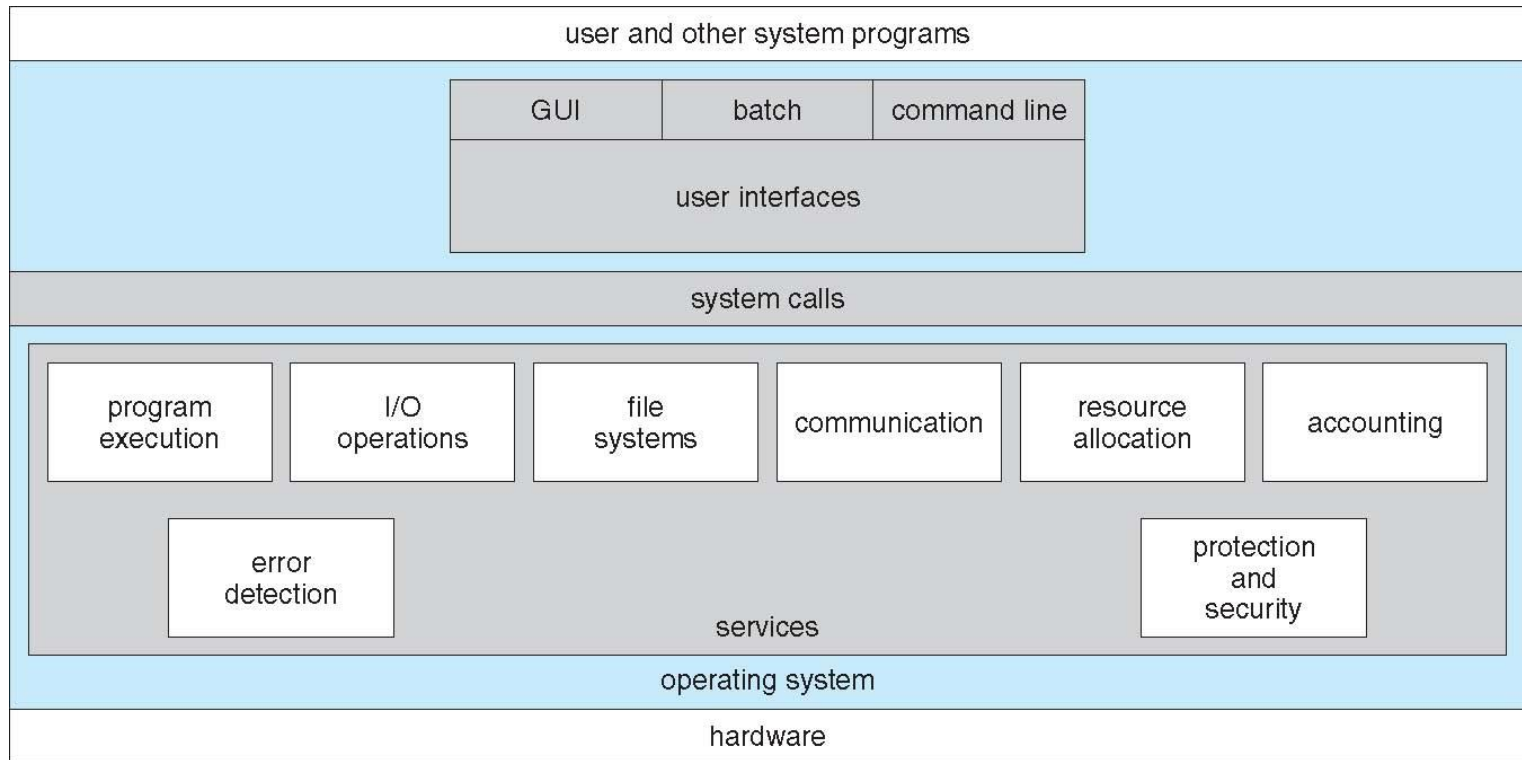
Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
 - **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.**
 - **Accounting** - 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 computer system may want to control 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

CLI or **command interpreter** allows direct command entry

- OS provides **command-line interface**, or command interpreter that allows users to directly enter commands to be performed by the operating system.
- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
- Sometimes commands are built-in, sometimes just names of programs(implements most commands through system programs)
 - ▶ If the latter, adding new features doesn't require shell modification





User Operating System Interface - GUI

- User-friendly **desktop** interface
 - Usually mouse, keyboard, and monitor
 - **Icons** represent files, programs, actions, etc
 - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**))
- Many systems now include both CLI and GUI interfaces
 - Microsoft Windows is 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





Touchscreen Interfaces

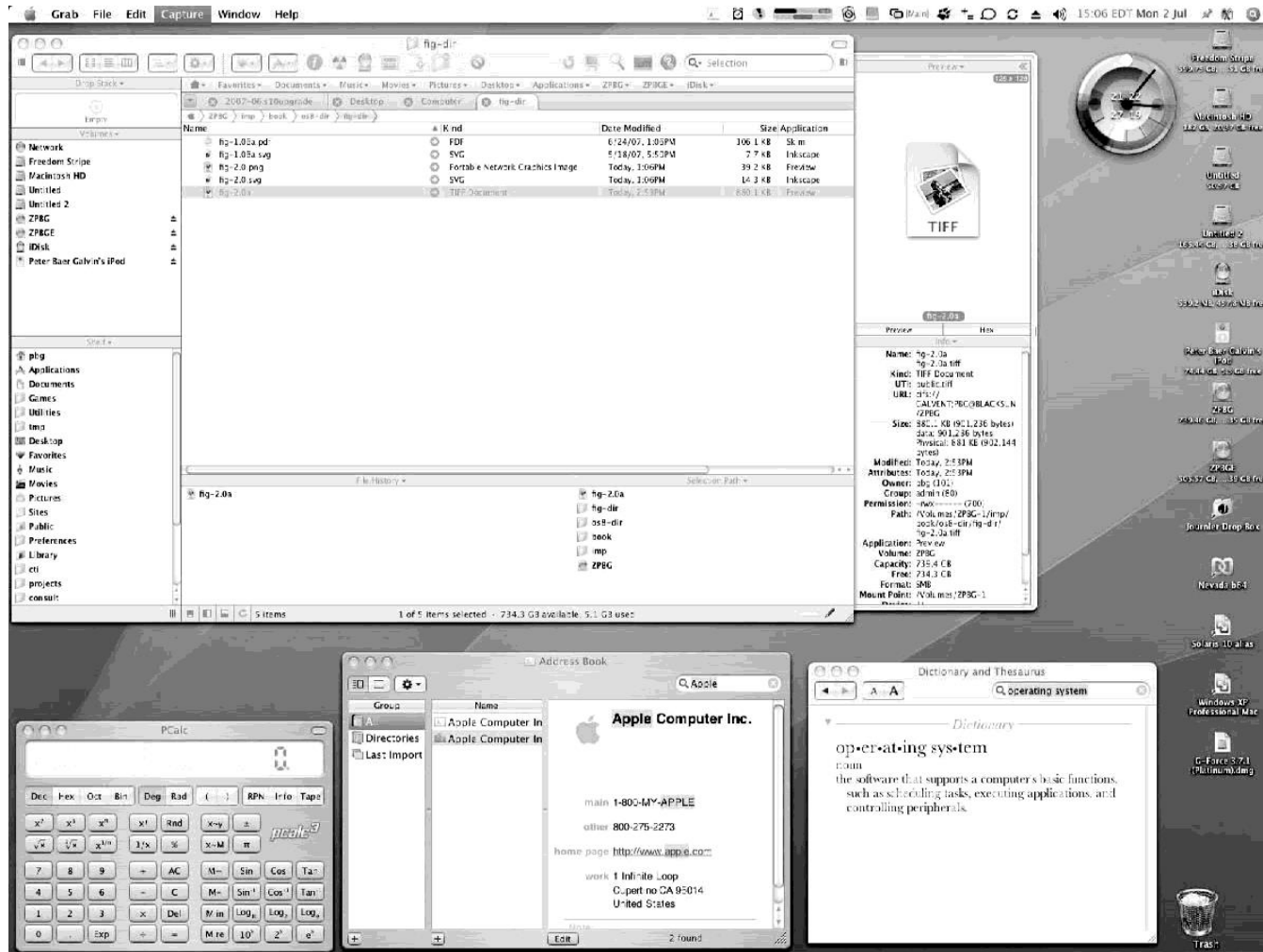
Touchscreen devices require new interfaces

- Mouse not possible or not desired
- Actions and selection based on gestures
- Virtual keyboard for text entry
- Voice commands.





The Mac OS X GUI





System Calls

- Programming interface to the services provided by the OS
- 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)

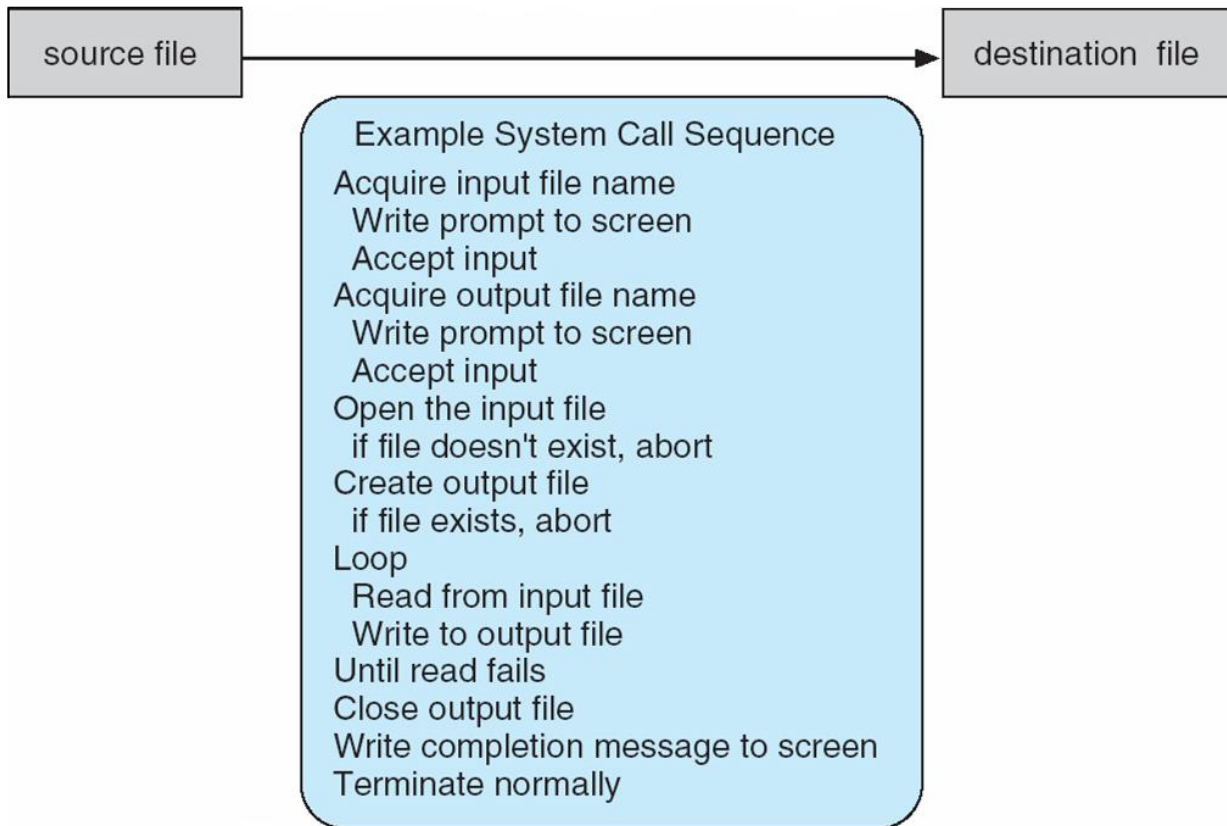
Note that the system-call names used throughout this text are generic





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)
```

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 where the data will be read into
- `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 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 need 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





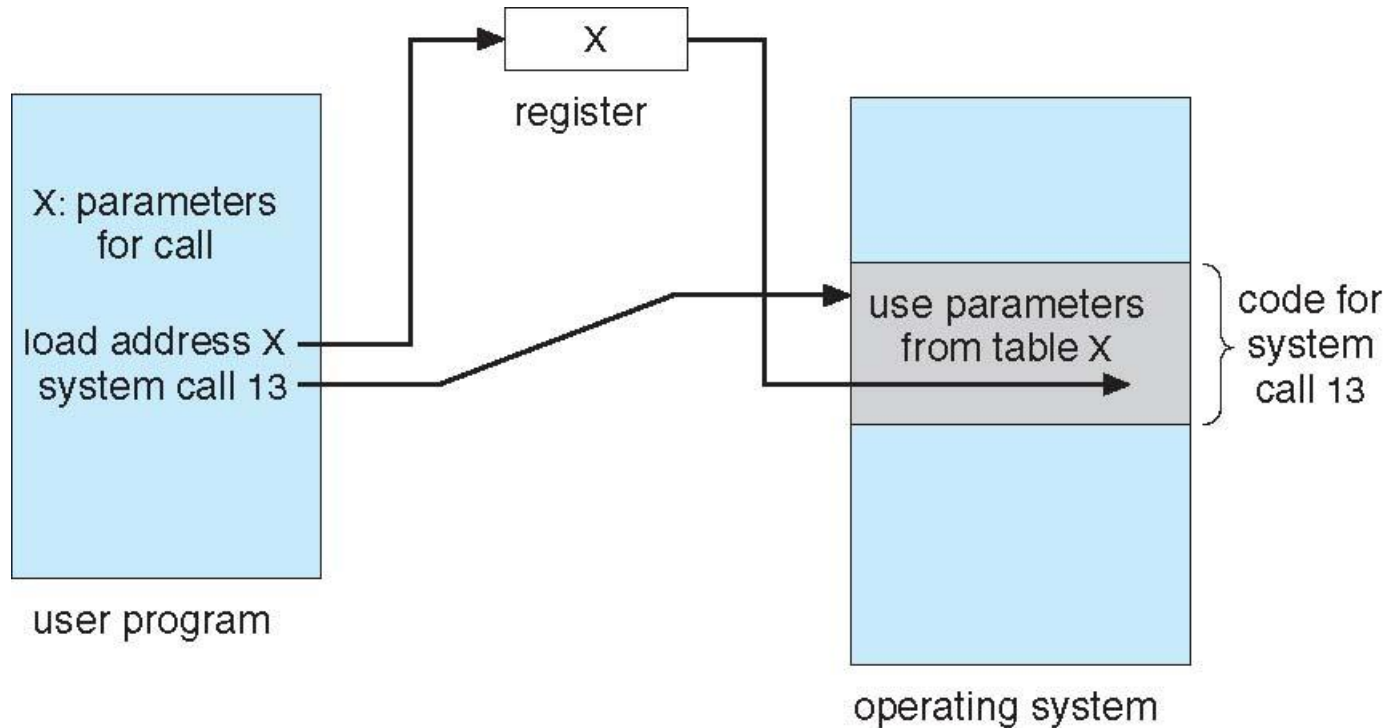
System Call Parameter Passing

- 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



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**
- ❖ **Communications**
- ❖ **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 -corrupted system files, damaged HDD,
Corrupted RAM, Compatibility of h/w and s/w
- **Debugger** for determining **bugs, single step** execution
- **Locks** for managing access to shared data between processes
- Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command.





Types of System Calls

- 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: **message passing model** , **Shared-memory model**
 - create, delete communication connection
 - send, receive messages if **message passing model** to **host name** or **process name**
 - ▶ useful for exchanging **smaller amounts** of data
 - **Shared-memory model** create and gain access to memory regions
 - allows **maximum** speed
 - Must ensure processes are not writing to the same location simultaneously
 - attach and detach remote devices





Types of System Calls (Cont.)

- Protection
 - Control access to resources
 - Get and set permissions
 - Allow and deny user access
- Typically, system calls providing protection include **set permission()**, **get permission()**,





Examples of Windows and Unix System Calls

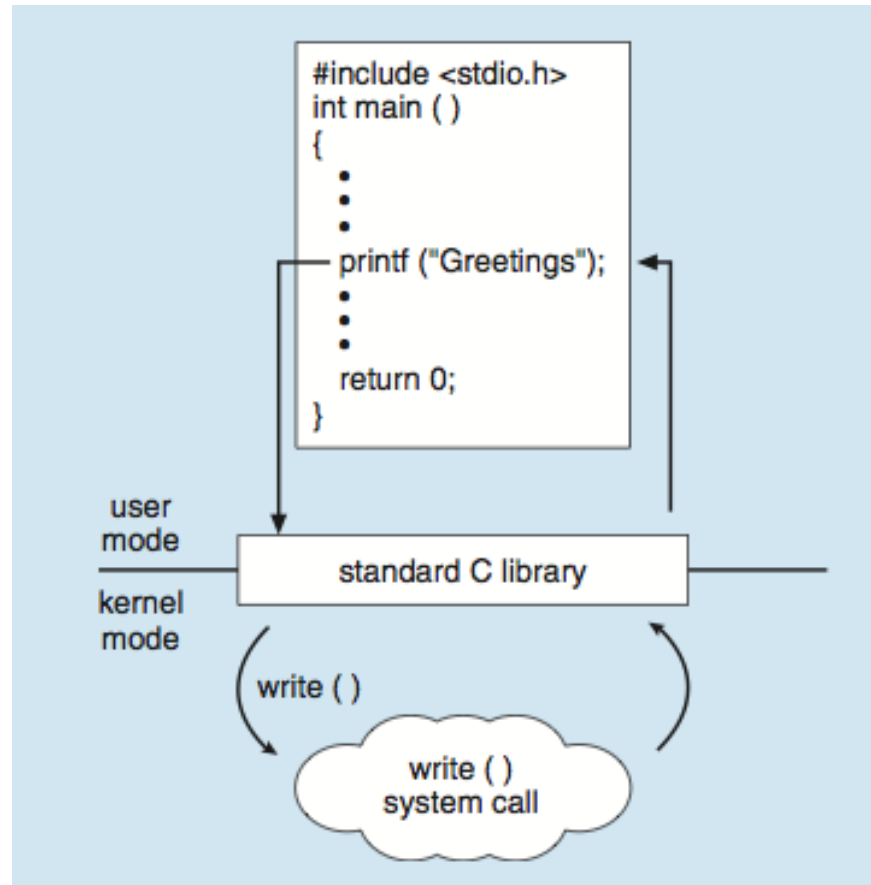
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information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





Standard C Library Example

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

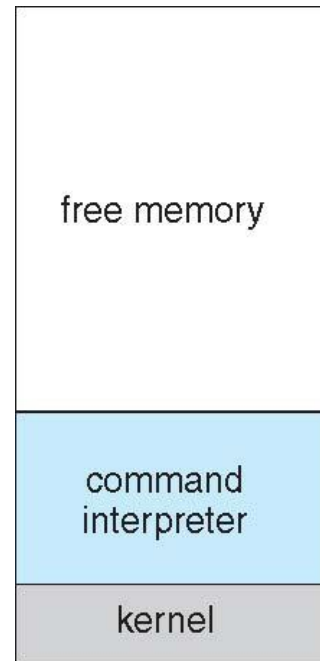




Variations in process control

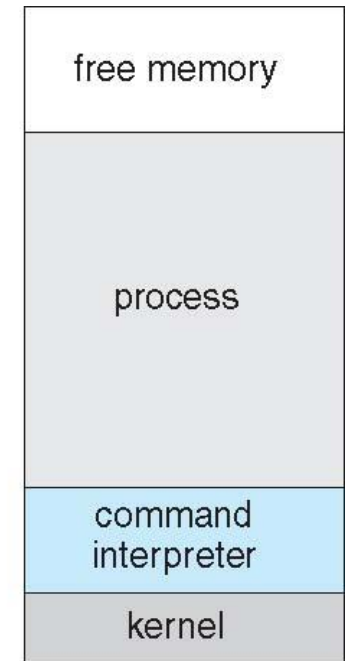
Example: MS-DOS

- **Single-tasking** System
- Shell invoked when system booted
- Simple method to run program
 - Running a program **replaces the currently running program.**
 - The currently running program is overwritten,
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded



(a)

At system startup



(b)

running a program

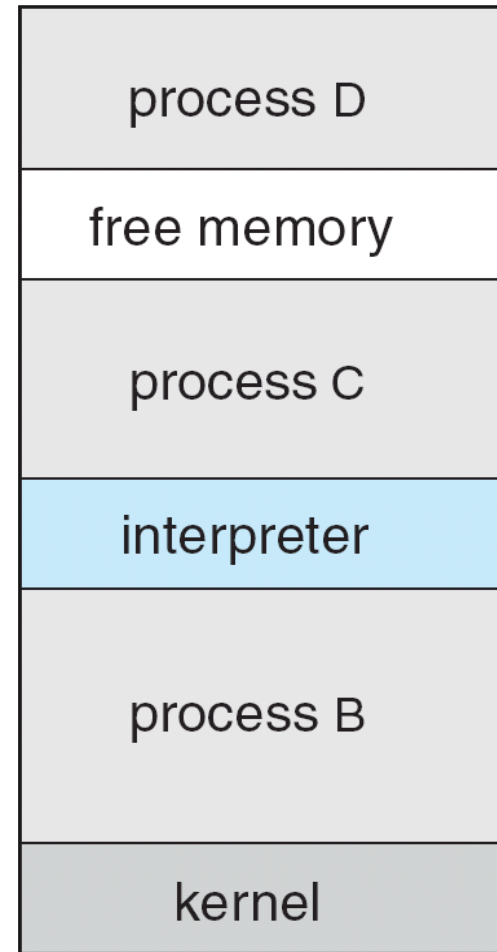




Example: FreeBSD

FreeBSD (**Free Berkeley Software Distribution**), FreeBSD is a **free and open-source Unix-like operating system**

- 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 process
 - Shell waits for process to terminate or continues with user commands
- Process exits with:
 - `code = 0` – no error
 - `code > 0` – error code





System Programs

- System programs provide a convenient environment for program development and execution.

They can be divided into:

- File manipulation
 - Status information
 - ▶ File modification
 - 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 Programs

- Provide a convenient environment for program development and execution
 - Some of them are simply user interfaces to system calls; others are considerably more complex
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
 - Some ask the system for info - date, time, amount of available memory, disk space, number of users
 - Others provide detailed performance, logging, and debugging information
 - Typically, these programs format and print the output to the terminal or other output devices
 - Some systems implement a **registry** - used to store and retrieve configuration information





System Programs (Cont.)

- **File modification**
 - Text editors to create and modify files
 - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes 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 for system startup, then terminate
 - ▶ Some from system boot to shutdown
- Provide facilities like disk checking, process scheduling, error logging, printing
- Run in user context not kernel context
- Known as **services**, **subsystems**, **daemons**

■ Application programs

- Run by users
- Not typically considered part of OS
- Launched by command line, mouse click, finger poke





Operating System Structure

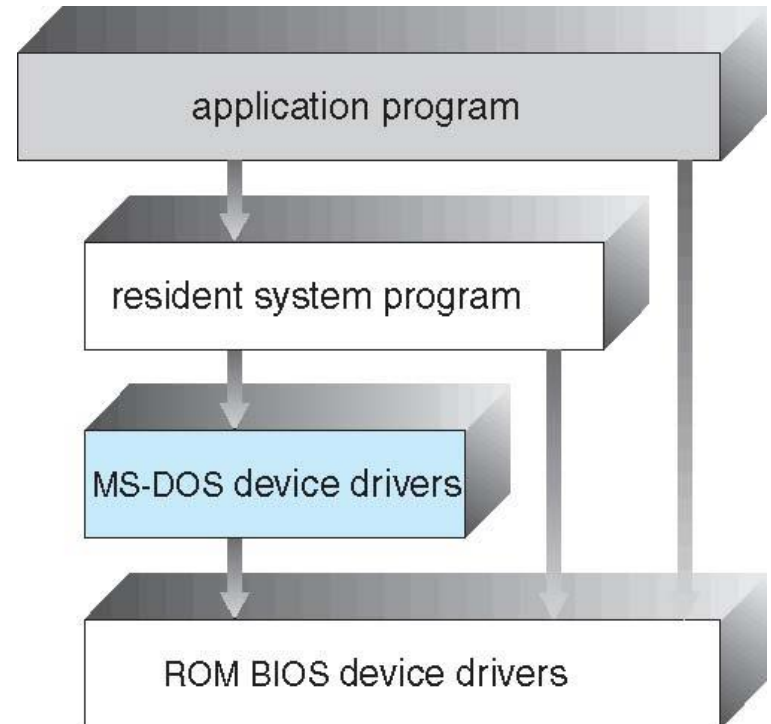
- General-purpose OS is very large program
- Various ways to structure ones
 - Simple structure – MS-DOS
 - More complex -- UNIX
 - Layered – an abstraction
 - Microkernel -Mach





Operating System Structure

- MS-DOS – written to provide the most functionality in the least space.
 - Not divided into modules
 - Although MS-DOS has some structure, its interfaces and levels of functionality are **not well separated**.





Non Simple Structure -- UNIX

UNIX – limited by hardware functionality, the original UNIX operating system had **limited structuring**. The UNIX OS consists of two separable parts

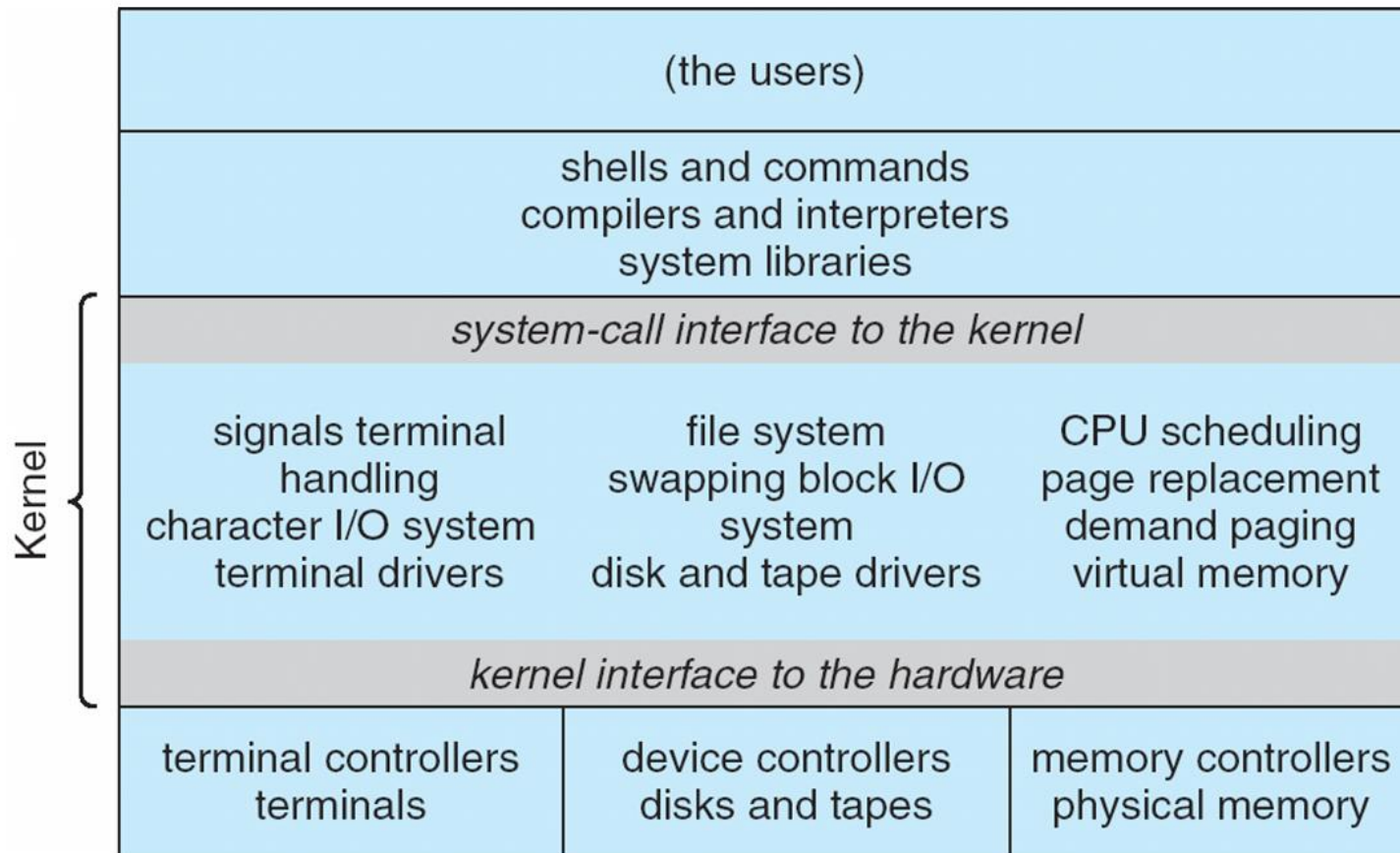
- Systems programs
- The kernel
 - ▶ Consists of everything below the system-call interface and above the physical hardware
 - ▶ Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level





Traditional UNIX System Structure

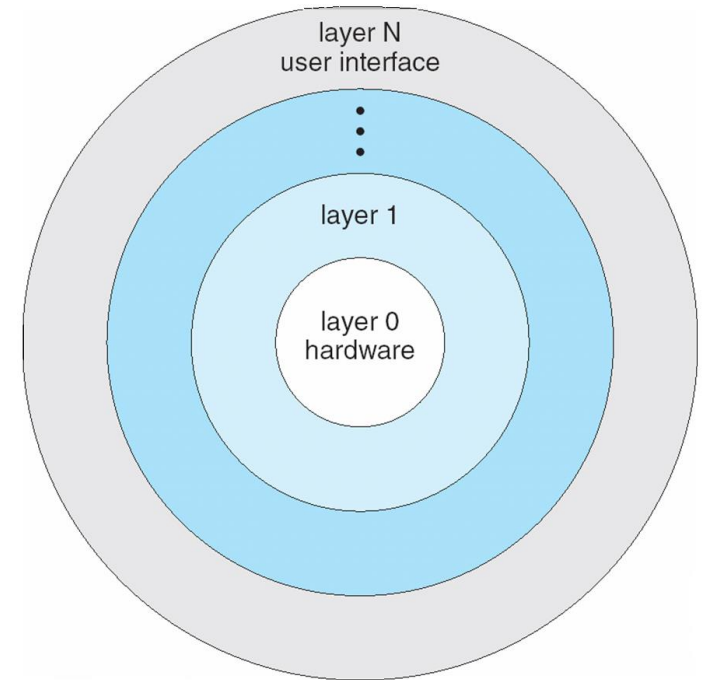
Beyond simple but not fully layered: **monolithic structure, difficult** to implement and maintain





Layered Approach

- The operating system is divided into several 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.
- With **modularity**, layers are selected such that each uses functions (operations) and services of only **lower-level layers**.
- Easy to implement and debug.
- The major difficulty with the layered approach involves appropriately defining the various layers.
- **Ex:** the **device driver** for the backing store (disk space used by virtual-memory algorithms) must be at a lower level than the **memory-management routines** because memory management requires the ability to use the backing store.





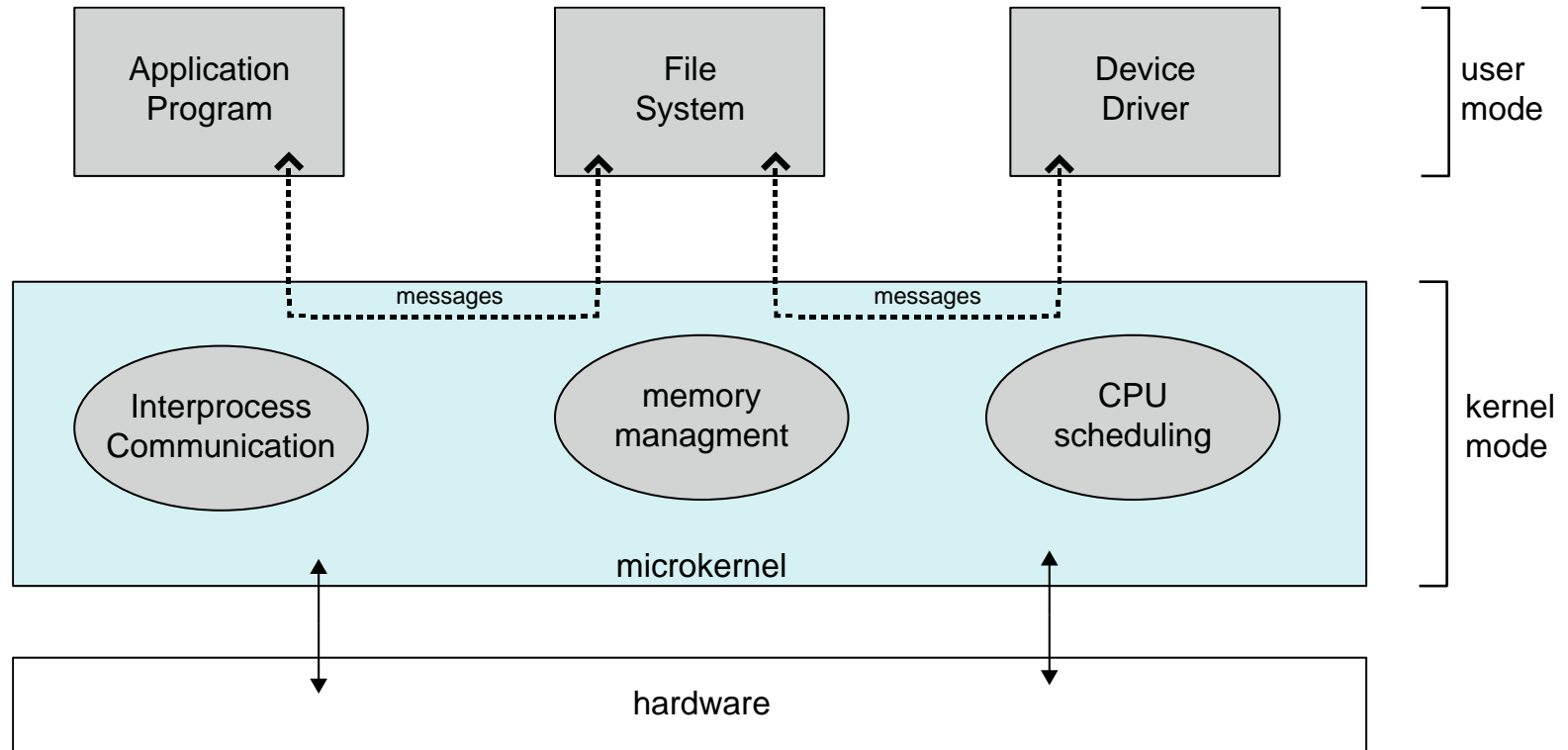
Microkernel System Structure

- Moves as much from the kernel into user space
- **Mach** example of **microkernel**
 - Mac OS X kernel partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
 - Easier to extend a microkernel
 - Easier to port the operating system to new architectures
 - More reliable (less code is running in kernel mode)
 - More secure
- Detriments:
 - Performance overhead of user space to kernel space communication



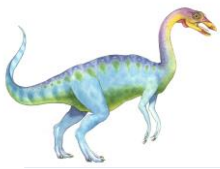


Microkernel System Structure



Architecture of a typical microkernel.





Monolithic vs Microkernel

MICROKERNEL KERNEL

A kernel type that provides mechanisms such as low-level address space management, thread management and interprocess communication to implement an operating system

OS services and kernel are separated

Slow

Failure in one component will not affect the other components

Easier to add new functionalities

Smaller in size

MONOLITHIC KERNEL

A type of kernel in operating systems where the entire operating system works in the kernel space

Kernel contains the OS services

Fast

Failure in one component will affect the entire system

Difficult to add new functionalities

Larger in size



System Boot

- When power is initialized on system, execution starts at a fixed memory location.
- The operating system must be made available to hardware so hardware can start it
- The procedure of starting a computer by loading the **kernel** is known as **booting** the system
- Small piece of code – **bootstrap loader**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options

Sometimes **it's a two-step** process where simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the **kernel**.

- Kernel loads and system is then **running**



End of Chapter 2

