# Chapter 2: Operating-System Structures

# Design Goals

An operating system provides the environment within which programs are executed.

It is important that the goals of the system be well defined before the design begins.

1. One view focuses on the services that the system provides;
2. Another, on the interface that it makes available to users and programmers;
3. A third, on its components and their interconnections.

# 2.1 Operating-System Services

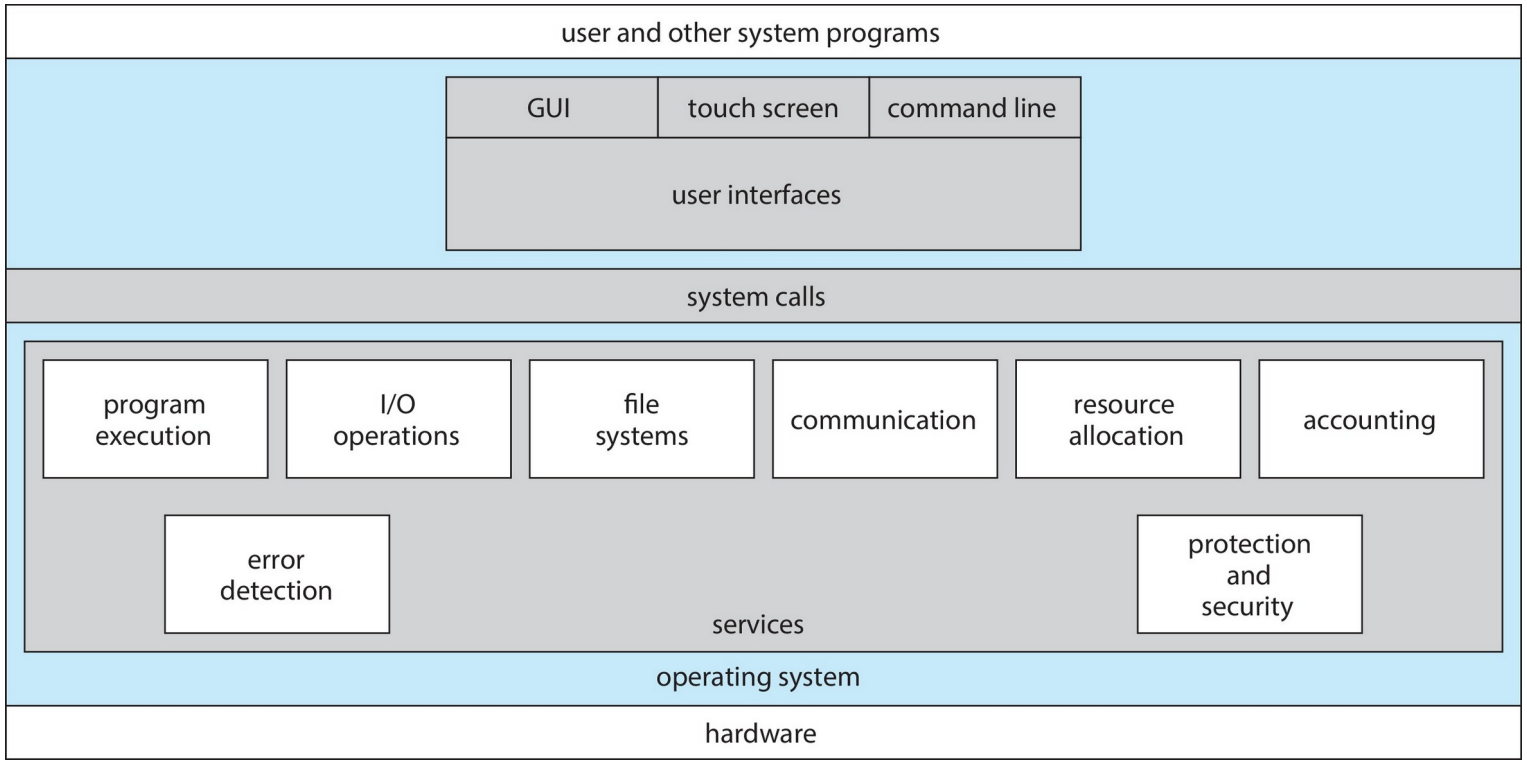
# Operating System Services

The specific services provided differ from one operating system to another, but we can identify common classes.

These services make the programming task easier for the programmer

1. One set of operating-system services provides functions that are helpful to the user
2. Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing

# A View of Operating System Services



# 1. Operating System Services- for user ease

**One set of operating-system services provides functions that are helpful to the user:**

1. **User interface** - Almost all operating systems have a user interface (**UI**).
   - Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **touch-screen, Batch**
2. **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)
3. **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.

# Operating System Services - for user (cont.)

4. **File-system manipulation** - Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management. Many operating systems provide a variety of file systems.
5. **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)
6. **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

# 2. Operating System Services – for system efficiency

Another set of OS functions exists for ensuring the **efficient operation** of the **system** itself via **resource sharing:**

1. **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.
2. **Logging** - To keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics.
3. **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

# 2.2 User and Operating-System Interface

# User Interfaces

There are several ways for users to interface with the operating system.

Three fundamental approaches:

One provides a command-line interface, 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 operating system via a graphical user interface, or GUI.

# Command-line interface - CLI

**CLI** or **command interpreter** allows users to directly enter commands to be performed by the operating system

- On systems with multiple command interpreters to choose from, the interpreters are known as **shells**.
  - UNIX and Linux systems provide several shells, including the C shell, Bourne-Again shell (bash), and Korn shell. Third-party shells and free user-written shells are also available.

The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on.

# Command interpreter implementation

These commands can be implemented in two general ways:

1. Command interpreter itself contains the code to execute the command.
   - Code sets up the parameters and makes the appropriate system call
   - The number of commands that can be given determines the size of the command interpreter since each command requires its own implementing code

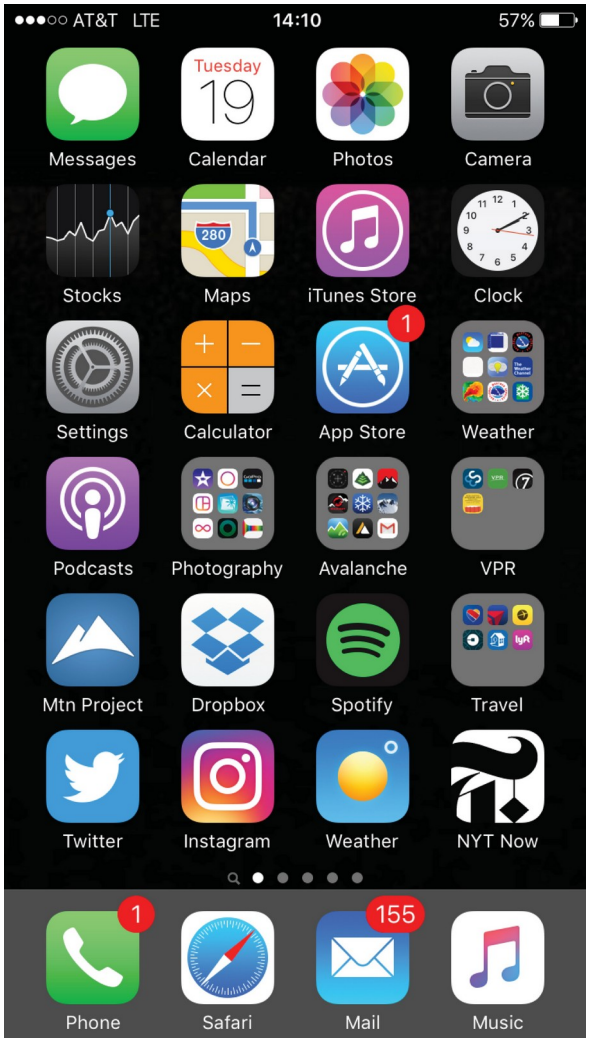
2. An alternative approach—used by UNIX, among other operating systems —implements most commands through system programs. Merely uses the command to identify a file to be loaded into memory and executed.
   - Example: UNIX command to delete a file `rm file.txt` would search for a file called `rm`, load the file into memory, and execute it with the parameter `file.txt`.
   - Programmers can add new commands to the system easily by creating new files with the proper program logic.

# Graphical User Interface- GUI

- User-friendly **desktop** metaphor **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**)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows - GUI with **CLI** "command" shell
  - Apple macOS - "***Aqua***" GUI interface with **UNIX kernel** underneath and **shells** available
  - Unix and Linux have **CLI** with optional GUI interfaces (**CDE, KDE** *K(ool) Desktop Environment*, ***GNOME*** desktop by the GNU project

# 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



# Choice of Interface

The choice of whether to use a command-line or GUI interface is mostly one of personal preference.

- **System administrators** who manage computers and **power users** who have deep knowledge of a system frequently use the command-line interface.
  - It is more efficient, giving faster access to the activities they need to perform.
  - On some systems, only a subset of system functions is available via the GUI, leaving the less common tasks to those who are command-line knowledgeable.
  - Further, command-line interfaces make repetitive tasks easier, in part because they have their own programmability.
    - For example, if a frequent task requires a set of steps, those steps can be recorded into a file, and that file can be run just like a program.
    - The program is not compiled into executable code but rather is interpreted by the command-line interface. These **shell scripts** are very common on systems that are command-line oriented, such as UNIX and Linux.

# 2.3 System Calls

# System Calls

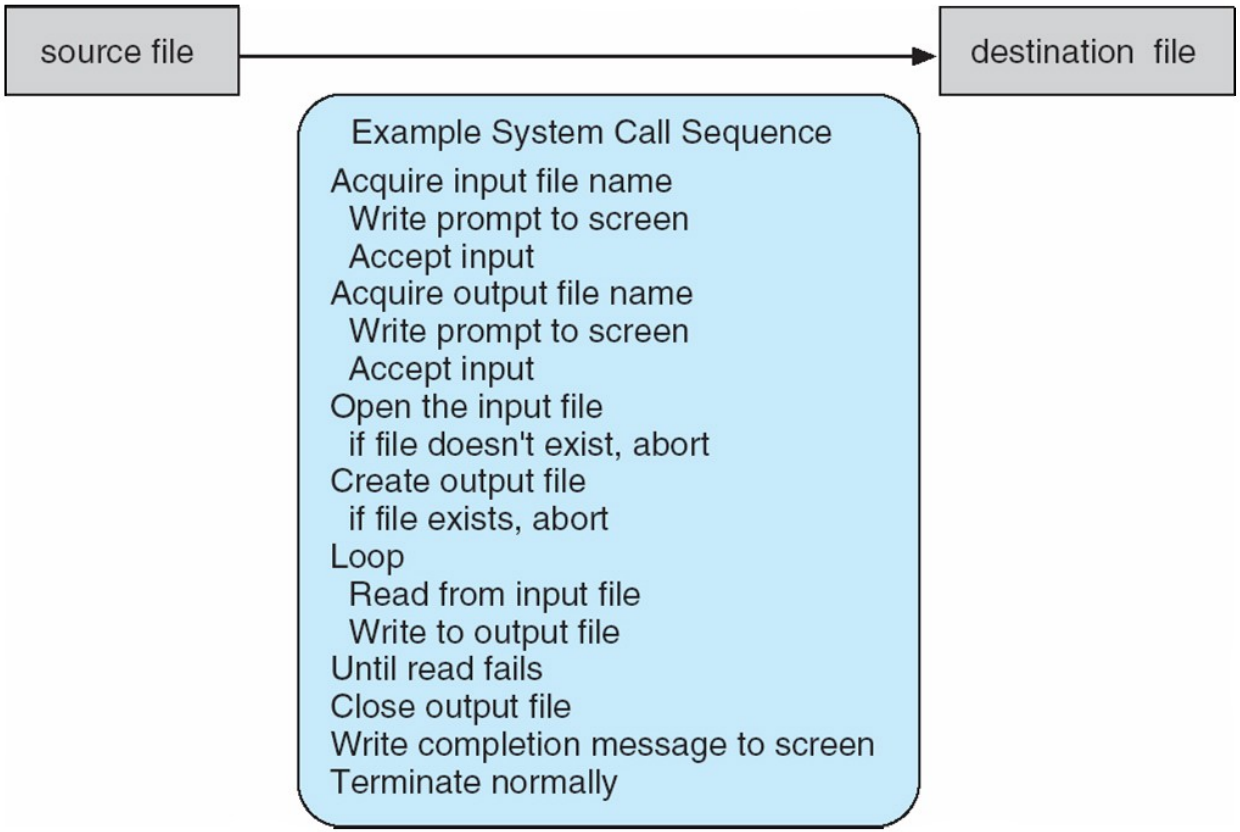
**System calls** provide an **interface** to the services provided by the **OS**

- Generally available as functions written in C and C++
- Certain low-level tasks may have to be written using assembly-language instructions.
  - Example: tasks where hardware must be accessed directly
    - Even simple programs may make heavy use of the operating system.
    - Frequently, systems execute thousands of system calls per second.

    Most programmers never see this level of detail.
    - Each operating system has its own name for each system call.

# Example of System Calls

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



# Application Programming Interface

System Calls are mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use.

- The **API specifies a set of functions** that are **available to an application programmer**, including the parameters that are passed to each function and the return values the programmer can expect.

- Functions that make up an API typically invoke the actual system calls on behalf of the application programmer.
- An API can specify the interface between an application and the operating system.

- A programmer accesses an API via a library of code provided by the operating system.
  - In UNIX and Linux for programs written in the C language, the library is called **libc**

# System Calls (cont.)

Two modes of OS:

1. **Kernel mode:** Privileged and powerful mode used by the operating system kernel

2. **User mode:** Where most user applications run

- **System calls** work silently in the background, **interfacing with the kernel** to get work done.

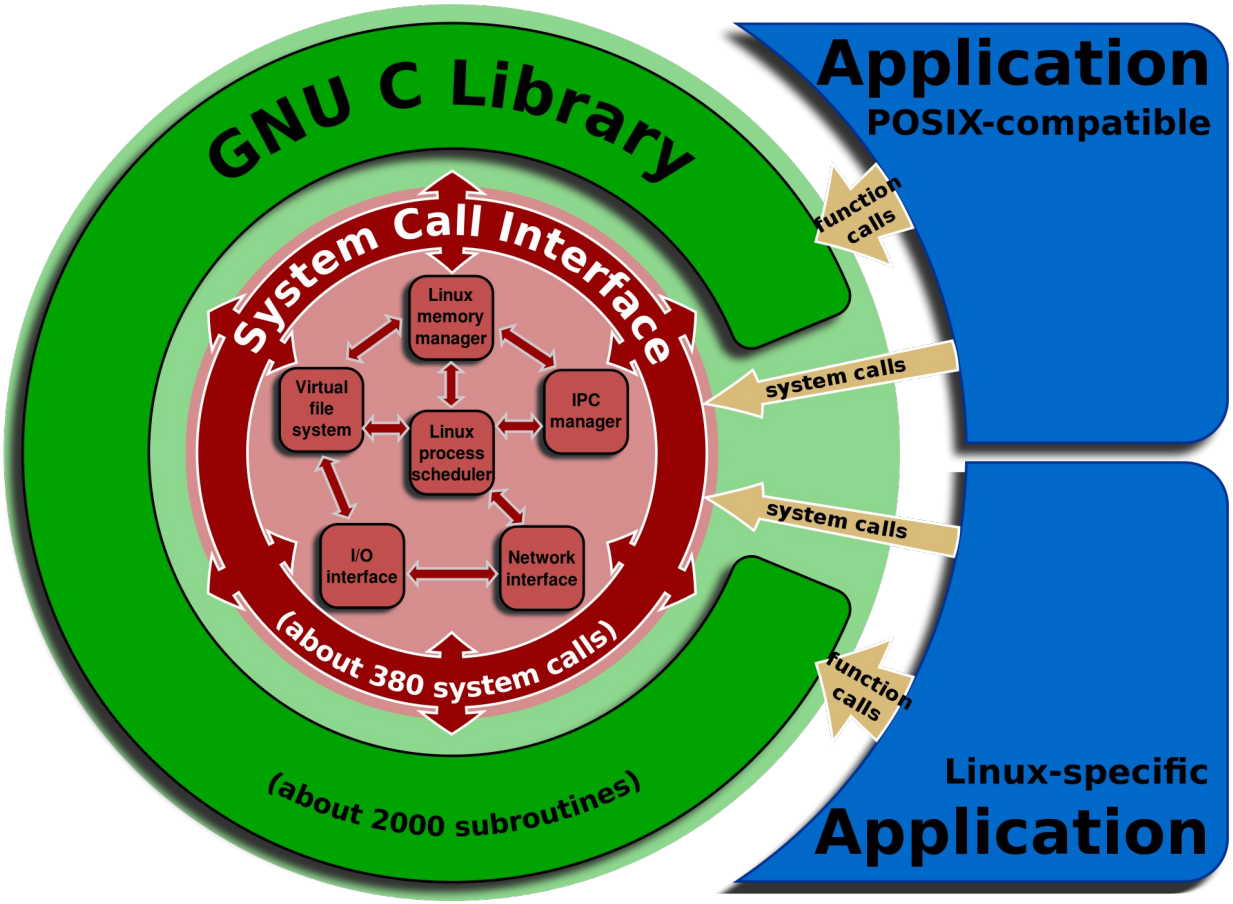
- System calls are very similar to function calls, which means they accept and work on arguments and return values.
  The only difference is that **system calls enter a kernel**, while function calls do not.

- Switching from user space to kernel space is done using trap (exception). Most of this is hidden away from the user by using system libraries (**glibc** on Linux systems).

- Even though system calls are generic in nature, the mechanics of issuing a system call are very much machine-dependent.

# Linux API

The Linux API is the kernel–user space API, which allows programs in user space to access **system resources** and **services of the Linux kernel**. It is composed out of the System Call Interface of the Linux kernel and the subroutines in the GNU C Library (glibc).



# Most Common APIs

**Three** of the **most common APIs** are:

1. **Win32 API** for **Windows,**

2. **POSIX API** for **POSIX-based systems (including virtually all versions of UNIX, Linux, and macOS**), and

   *Portable Operating System Interface is a family of standards for compatibility between operating systems*

3. **Java API** for the **Java virtual machine (JVM)**

# 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.

# Reasons for API

Why would an application programmer prefer programming according to an API rather than invoking actual system calls?

1. Program portability. An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API. (Architectural differences often make this more difficult
2. System calls can often be more detailed and difficult to work with than the API available to an application programmer.

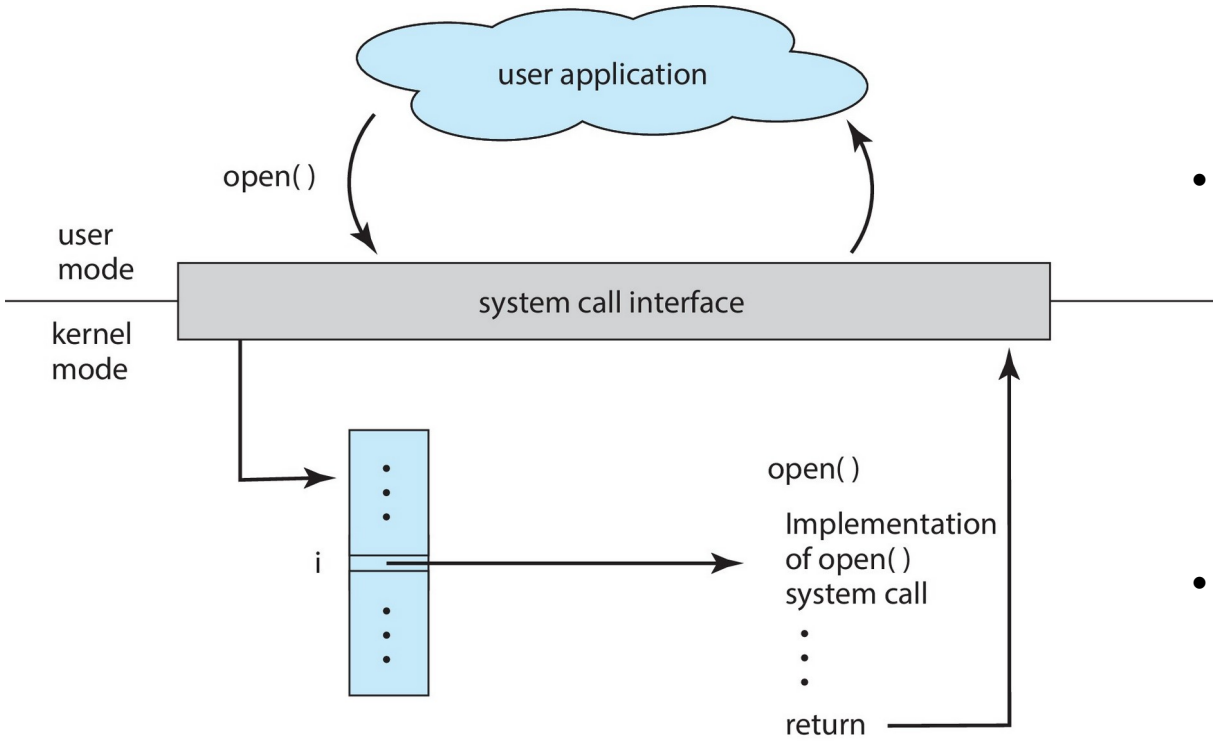
- The caller need know nothing about how the system call is implemented
- Most **details of OS interface hidden from programmer by API**

**Correlation** between a function in the API and its associated system call within the kernel:

Many of the POSIX and Windows APIs are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

# API – System Call – OS Relationship

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




- The system-call interface intercepts function calls in the API and invokes the necessary system calls within the OS
- Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers.
- The system-call interface then invokes the intended system call in the **operating-system kernel** and returns the status of the system call.

# 2.3.3 Types of System Calls

# 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.

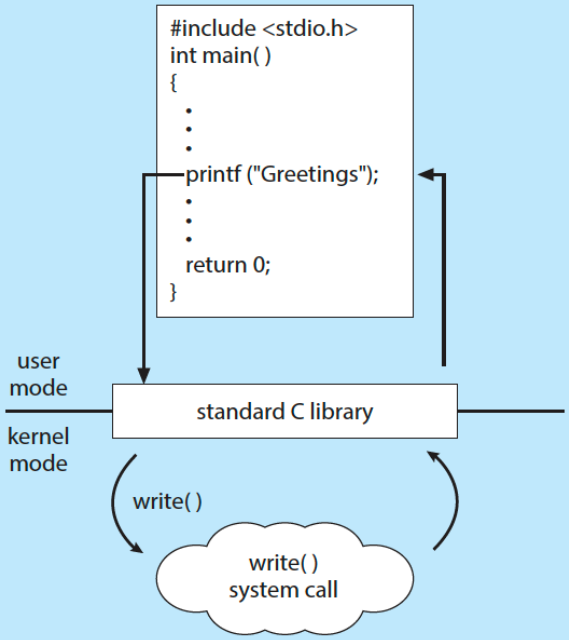
| | Windows | Unix |
|---|---|---|
| **Process control** | `CreateProcess()`<br>`ExitProcess()`<br>`WaitForSingleObject()` | `fork()`<br>`exit()`<br>`wait()` |
| **File management** | `CreateFile()`<br>`ReadFile()`<br>`WriteFile()`<br>`CloseHandle()` | `open()`<br>`read()`<br>`write()`<br>`close()` |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

# Standard C Library Example

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

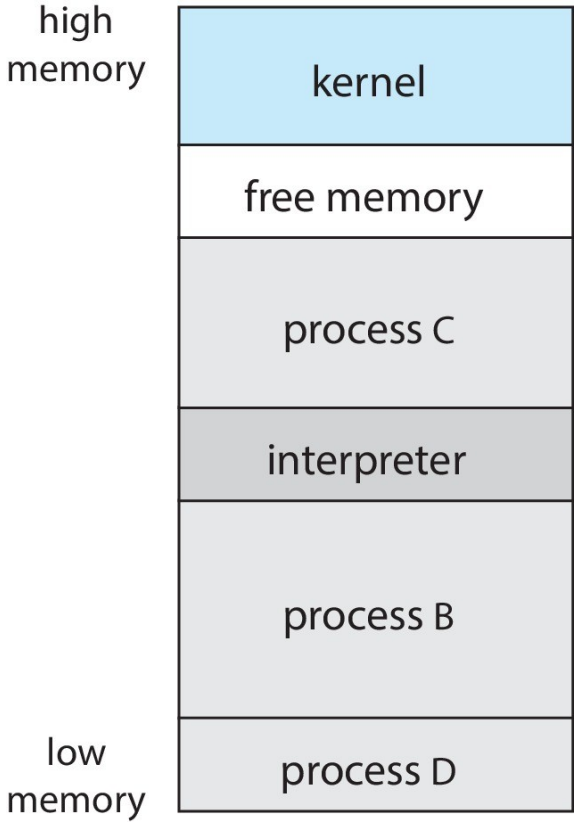
*THE STANDARD C LIBRARY*

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: FreeBSD - Multitasking

- Unix variant – from Berkeley UNIX
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes
  - fork() system call to create process
  - exec() to load program into process
  - Shell waits for process to terminate
  - or runs the process "in the background" and continues with other user commands. Here,
    - cannot receive input from keybord
    - I/O done through files or a GUI
- Process exits with exit() command:
  - code = 0 – no error
  - code > 0 – error code



# 2.4 System Services

# System Services

1. File management
2. Status information sometimes stored in a file
3. Programming language support
4. Program loading and execution
5. Communications
6. Background services
7. Application programs

# System Services (System utilities)

System services, also known as 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.

1. **File management** - **Create, delete, copy, rename, print, dump, list**, and generally **manipulate** files and directories

2. **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 (specific hardware and software details) information

# System Services (Cont.)

3. **File modification**
   - Text editors to create and modify files
   - Special commands to search contents of files or perform transformations of the text
4. **Programming-language support** - Compilers, assemblers, debuggers and interpreters for common programming languages (such as C, C++, Java, and Python)
5. **Program loading and execution-** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide Absolute loaders, Relocatable loaders, Linkage editors, Overlay-loaders, and debugging systems
6. **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 Services (Cont.)

6. **Background Services**
   - Launch at boot time
     - Some for system startup, then terminate
     - Others continue to run till shutdown
   - Constantly running system-program processes are known as **services**, **subsystems**, or daemons.
   - Provide facilities like disk checking, process scheduling, error logging, printing
   - OS that run important activities in user context rather than in kernel context may use daemons to run these activities.
7. **Application programs**
   - Don't pertain to system
   - Run by users
   - Not typically considered part of OS

# Application programs

Along with system programs, most operating systems are supplied with programs that are **useful in solving common problems** or performing **common operations**.

Application programs include

- web browsers,
- word processors and text formatters,
- spreadsheets,
- database systems,
- compilers,
- plotting and statistical-analysis packages,
- Games
- …

The view of the operating system seen by most users is defined by the application and system programs, rather than by the actual system calls.

# 2.5 Linkers and Loaders

# Program → Process

**Programs** reside on disk as a binary executable. (e.g. prog.exe or a.out). To run on CPU, must be brought into memory and placed in the context of a **process**

**Steps:**

1. **Compiler**

   Source files (.c, .cpp, etc.) **compiled** into **object files** designed to be loaded into any physical memory location, a format known as **relocatable object file (.obj)**

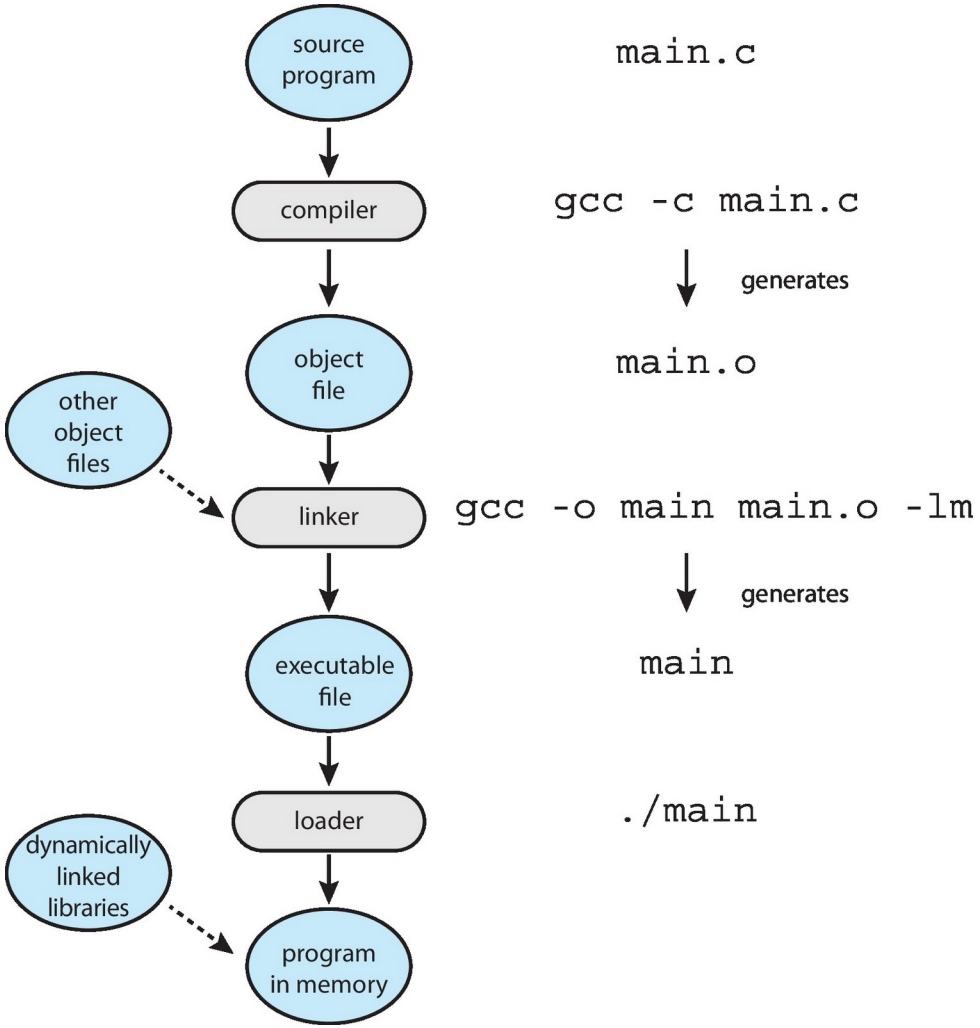
2. **Linker**

   1. **Linker** combines object (.obj) files into single binary **executable file (.exe)**
      - Linking also includes other object files or libraries

3. **Loader**

   - To run on a CPU core, the binary executable (.exe) file is brought into memory by **loader**, and be placed in the context of a **process**
   - **Relocation** assigns final addresses to program parts and adjusts code and data in program to match those addresses

# The Role of the Linker and Loader




gcc -c compiles source files without linking

Converts into binary file .o object code

gcc -l links with link shared libraries
-lm includes math library
gcc -o generates an executable output program

# Example Program Execution

1. Enter the program name on the command line. for example, ./main

2. The shell first **creates** a **new process** to run the program using the fork() system call.

3. The shell then invokes the loader with the exec() system call, passing exec() the name of the executable file.

4. The loader then **loads** the specified program **into memory** using the ***address space*** of the newly created process.

When a GUI interface is used, double-clicking on the icon associated with the executable file invokes the loader using a similar mechanism.

# DLL, ELF

Not all libraries are linked into the executable file and loaded into memory

- Rather, **dynamically linked libraries** (in **Windows**, **DLLs**) are loaded as needed, shared by all that use the same version of that same library (loaded once)
  - The benefit of this approach is that it avoids linking and loading libraries that may end up not being used into an executable file. Instead, the library is conditionally **linked** and is **loaded** if it is required during program **run time**.
- Object, executable files have standard formats, so operating system knows how to load and start them.
  - Format includes compiled machine code and a symbol table containing metadata about functions and variables that are referenced in the program
    - In UNIX and **Linux** the standard format is **ELF** (Executable and Linkable Format)

# Why Applications are Operating System Specific

Apps compiled on one system usually not executable on other operating systems

- Each operating system provides its own **unique system calls**
- Own file formats, etc.

Apps can run on **multiple operating system** in one of 3 ways:

1. Written in interpreted language like Python, Ruby, and **interpreter available** for multiple operating systems
2. App written in language that **includes a VM** containing the running app
   - like Java which includes loader, byte-code verifier, etc.
3. **Use standard language or API,** compile separately on each operating system to run on each
   - Like POSIX API on variants of UNIX like OS

- The application can be written in an interpreted language (such as Python

or Ruby) that has an interpreter available for multiple operating systems.

- The interpreter reads each line of the source program, executes equivalent instructions on the native instruction set, and calls native operating system calls. Performance suffers relative to that for native applications.

# Application Binary Interface (ABI)

APIs specify certain functions at the application level. **ABI** (Application Binary Interface) is the architecture equivalent of API

An ABI is used to define how different components of binary code can interface for a given operating system on a given architecture, CPU, etc.

ABI specifies low-level details, including address width, methods of passing parameters to system calls, the organization, and so on.

If a binary executable file has been compiled and linked according to a particular ABI, it should be able to run on different systems that support that ABI.

# Cross-platform compatibility

These differences mean that unless an interpreter, RTE, or binary executable file is written for and compiled on a specific operating system
on a specific CPU type (such as Intel x86 or ARMv8), the application will fail to run.

- **RTE (run-time environment)**—the full suite of software needed to execute applications written in a given programming language, including its compilers or interpreters as well as other software, such as libraries and loaders.

Imagine the amount of work that is required for a program such as the Firefox browser to run on Windows, macOS, various Linux releases, iOS, and Android, sometimes on various CPU architectures.

# 2.7 Operating-System Design and Implementation

# OS Design Goals

- Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- **User** goals and **System goals**
  - **User goals** – operating system should be convenient to use, easy to learn, 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

# Implementation

- Much variation
  - Early Operating Systems in assembly language
  - Now many are written in C, C++, with small amounts of the system written in assembly language.
- In fact, more than one higher level language is often used.
  - The lowest levels of the kernel might be written in assembly language and C.
  - Higher-level routines might be written in C and C++, and system libraries might be written in C++ or even higher-level languages.
- More high-level the language easier to **port** to other hardware
  - But slower
- **Emulation** can allow an OS to run on non-native hardware

# 2.8 Operating-System Structure

# Operating System Structure

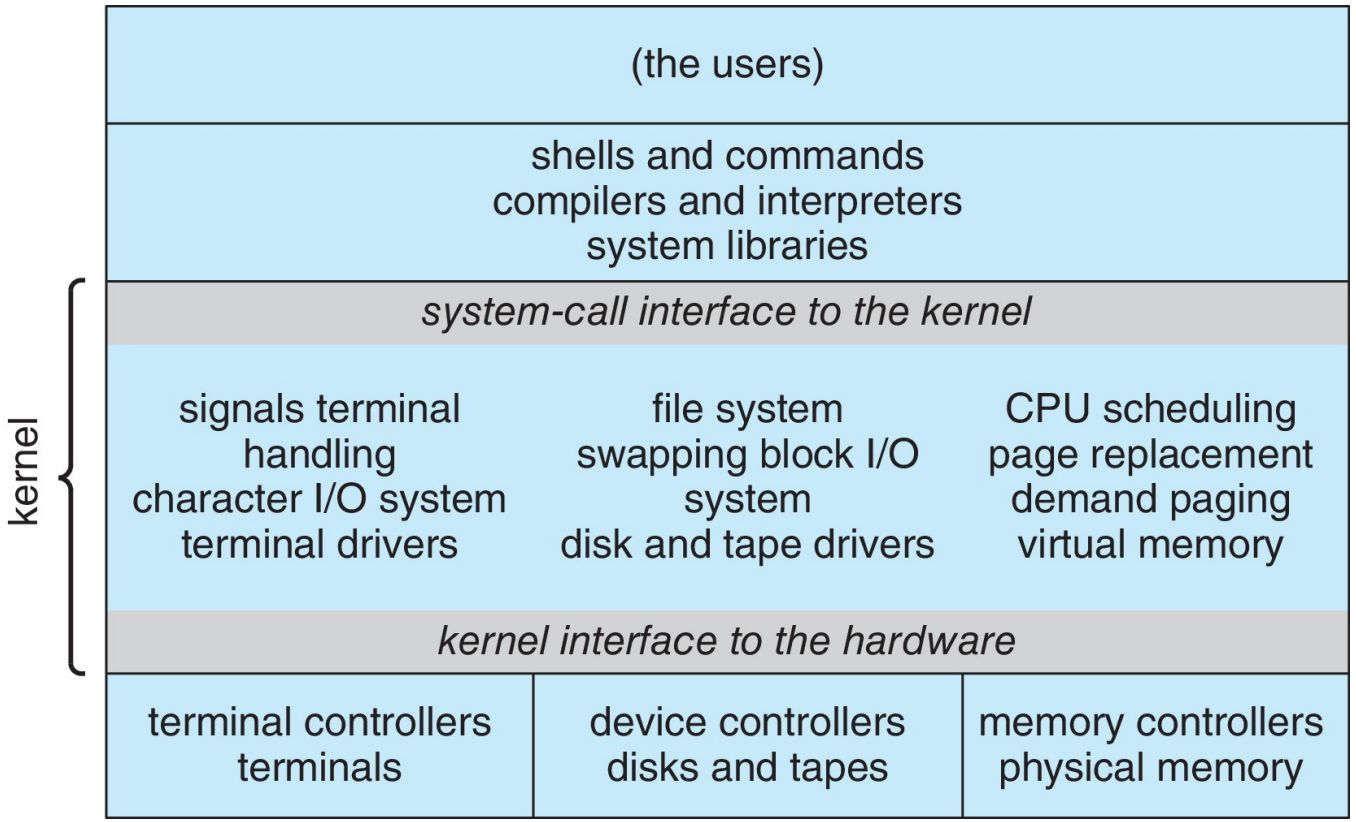
- A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily.
- A common approach is to partition the task into small components, or modules, rather than have one single system.
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex – UNIX
  - Layered – an abstraction
  - Microkernel – Mach

# Monolithic Structure – Original 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

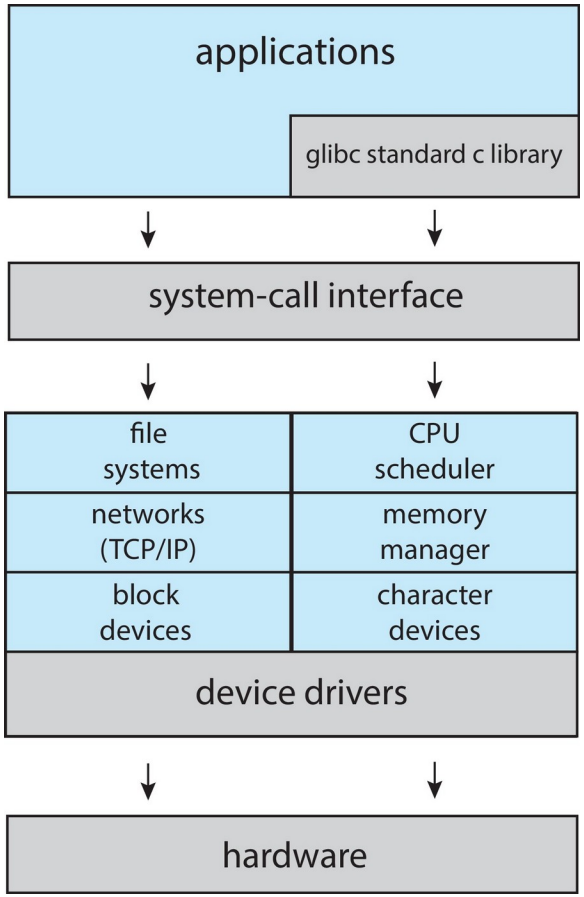
# Traditional UNIX System Structure

Beyond simple but not fully layered



# Linux System Structure

Monolithic plus modular design

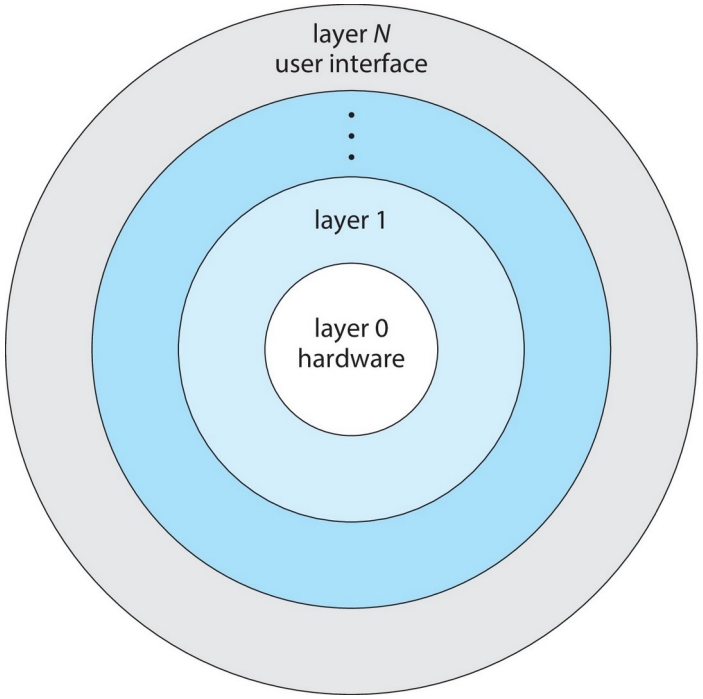



The Linux operating system is based on UNIX and is structured similarly.

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, but it does have a modular design that allows the kernel to be modified during run time.

# Layered Approach

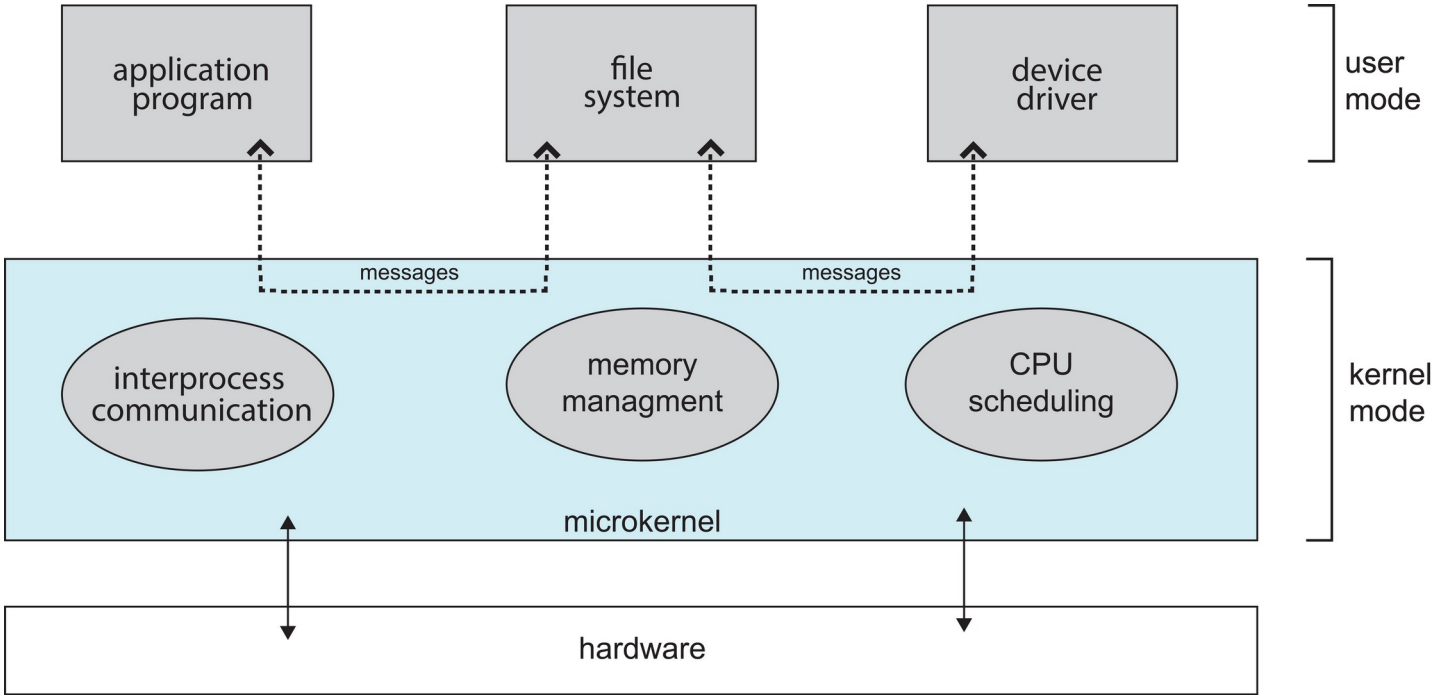
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



# Microkernels

- Moves as much from the kernel into user space
- **Mach** is an example of **microkernel**
  - Mac OS X kernel (**Darwin**) 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



# Hybrid Systems

- Most modern operating systems are not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem ***personalities***
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

# Android

- Developed by Open Handset Alliance (mostly Google)
  - Open Source
- Similar stack to IOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
    - Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc