# Operating System
# CS-2006
# Lecture 3


Mahzaib Younas

Lecturer Department of Computer Science

FAST NUCES CFD

# Outlines

- Operating System Services
- User and Operating System-Interface
- System Calls
- System Services
- Linkers and Loaders
- Operating System Structure

# Operating System Services

Operating systems provide an environment for execution of programs and services to programs and users.

**<u>Set of OS services provides functions that are helpful to the user</u>**

1. User interface
2. Program execution
3. I/O operations
4. File-system manipulation
5. Commination
6. Error Detection

**<u>set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing</u>**

1. Resource Allocation
2. Logging
3. Protection and security

# User Services

**User interface**

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

**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

**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)

**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.

# User Services

**I/O operations**

A running program may require I/O, which may involve a file or an I/O device

**Communication**

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)

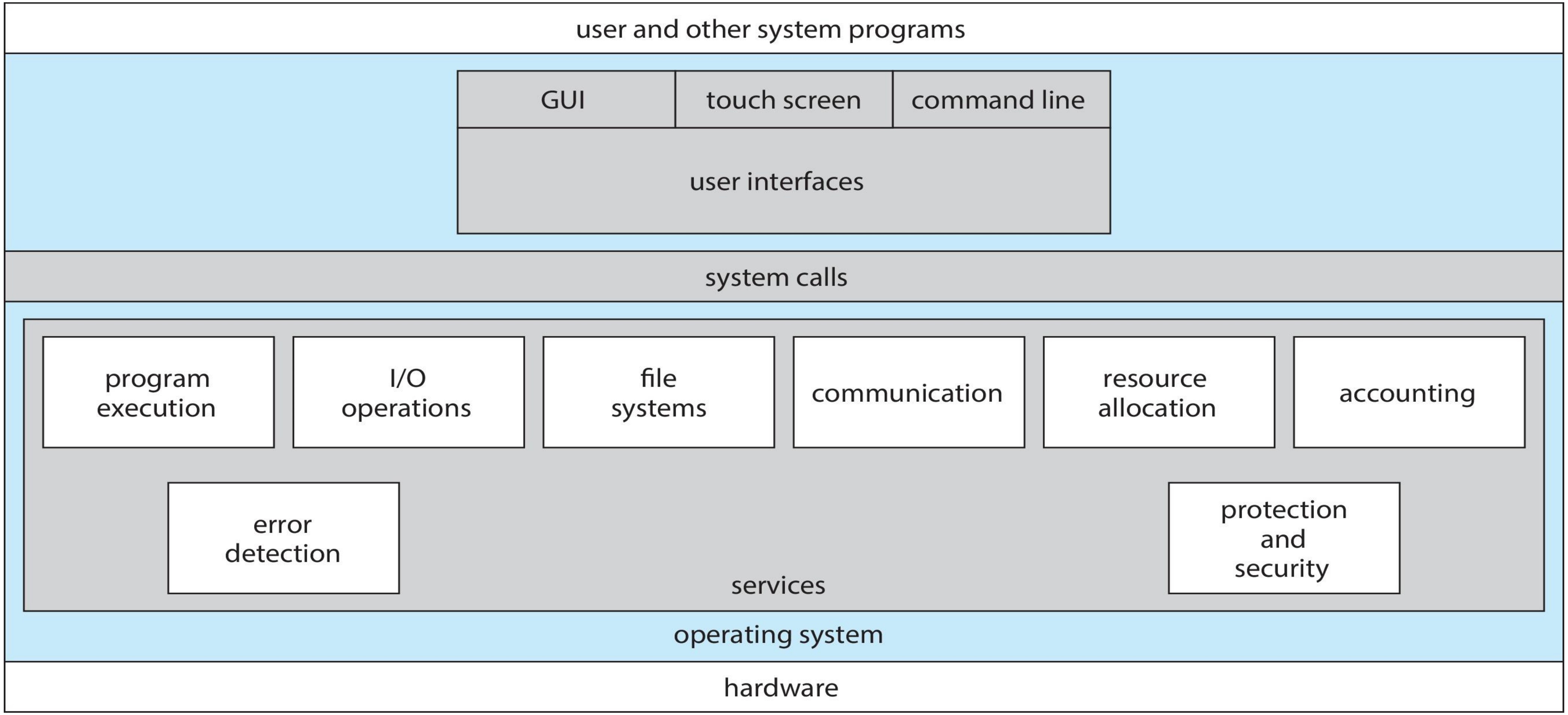
# Resource Sharing

**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**.

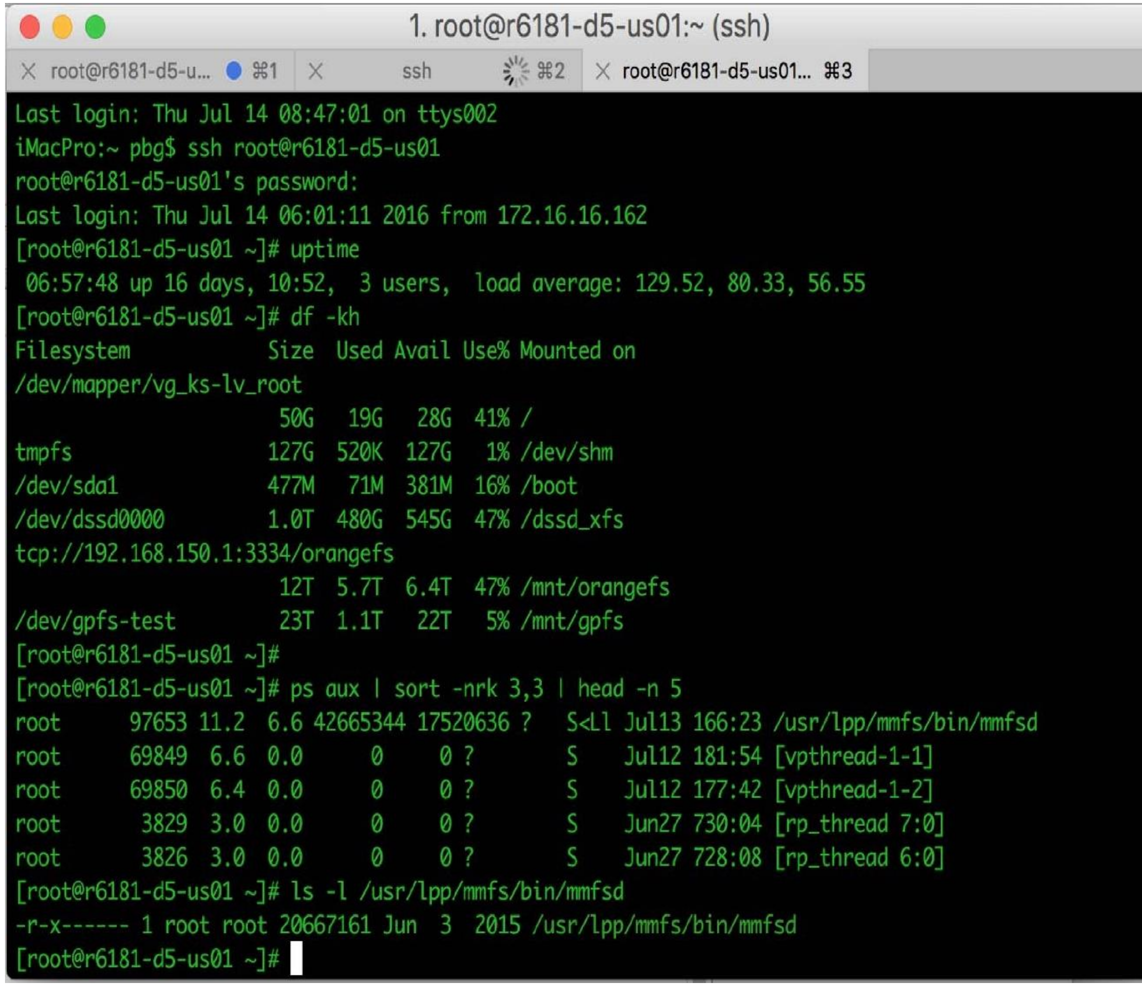
- **Logging -** To keep track of which users use how much and what kinds of computer resources
- **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each 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



# Command Line Interpreter

- CLI allows direct command entry
  - Sometimes implemented in **kernel**, sometimes by systems program
  - Sometimes multiple **flavors implemented – shells**
  - Primarily fetches a **command from user and executes it**
  - Sometimes **commands built-in, sometimes just names of programs**
  - If the latter, adding new features doesn't require shell modification



# Graphical User Interface

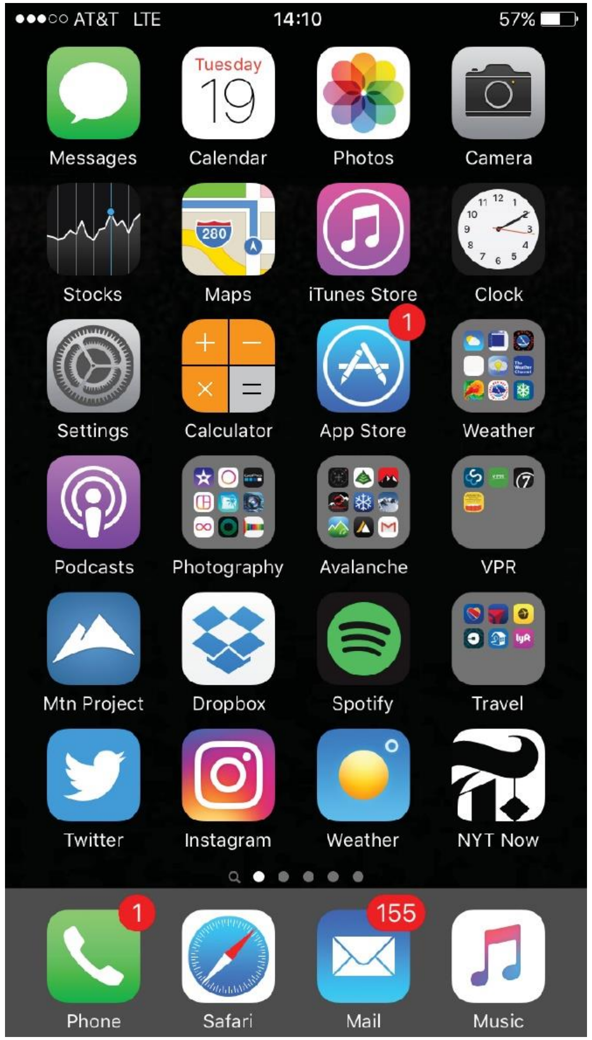
- User-friendly desktop metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
- Invented at Xerox PARC

# Both GUI and CLI

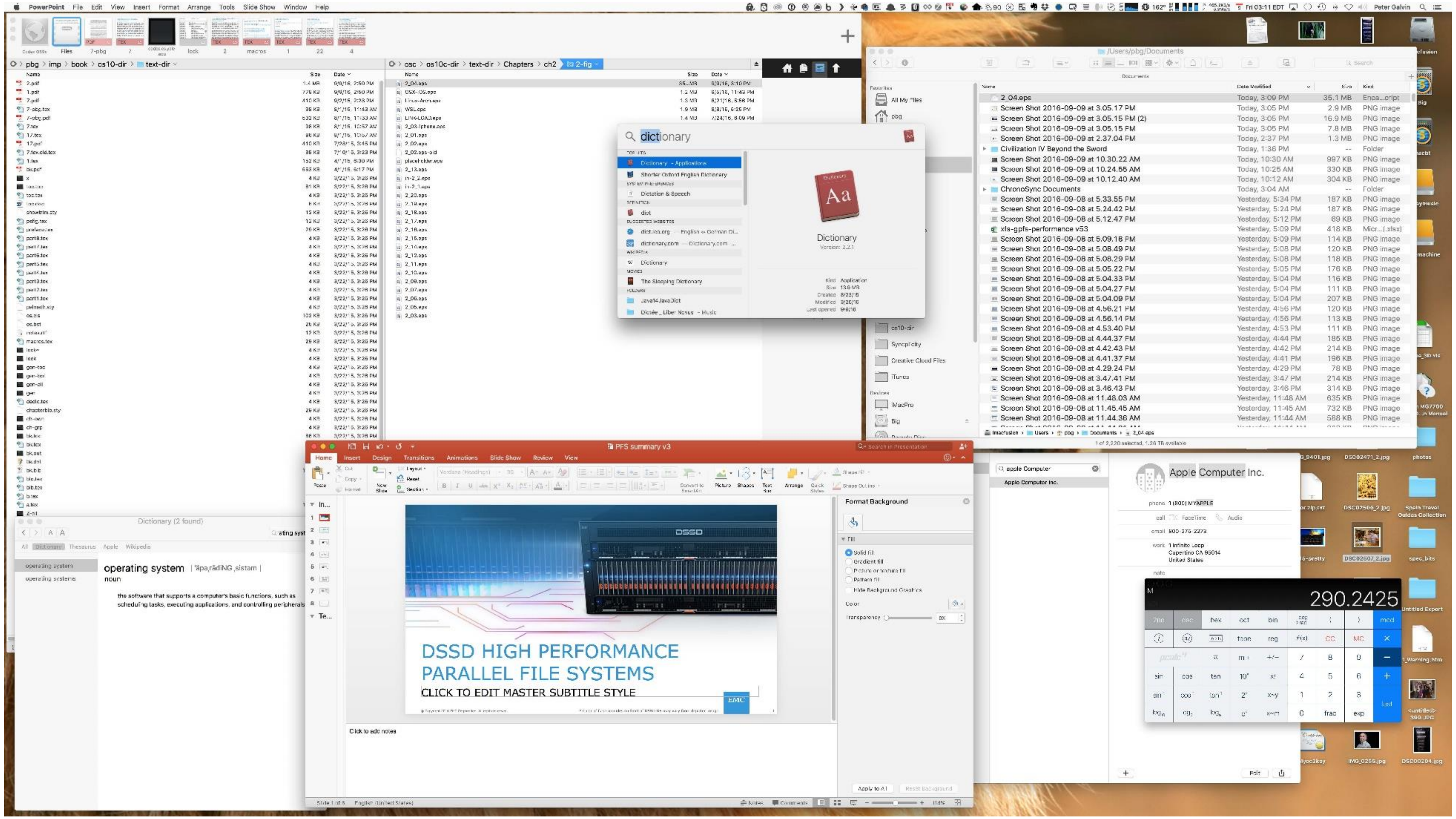
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

# Touch Screen Interface

- 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



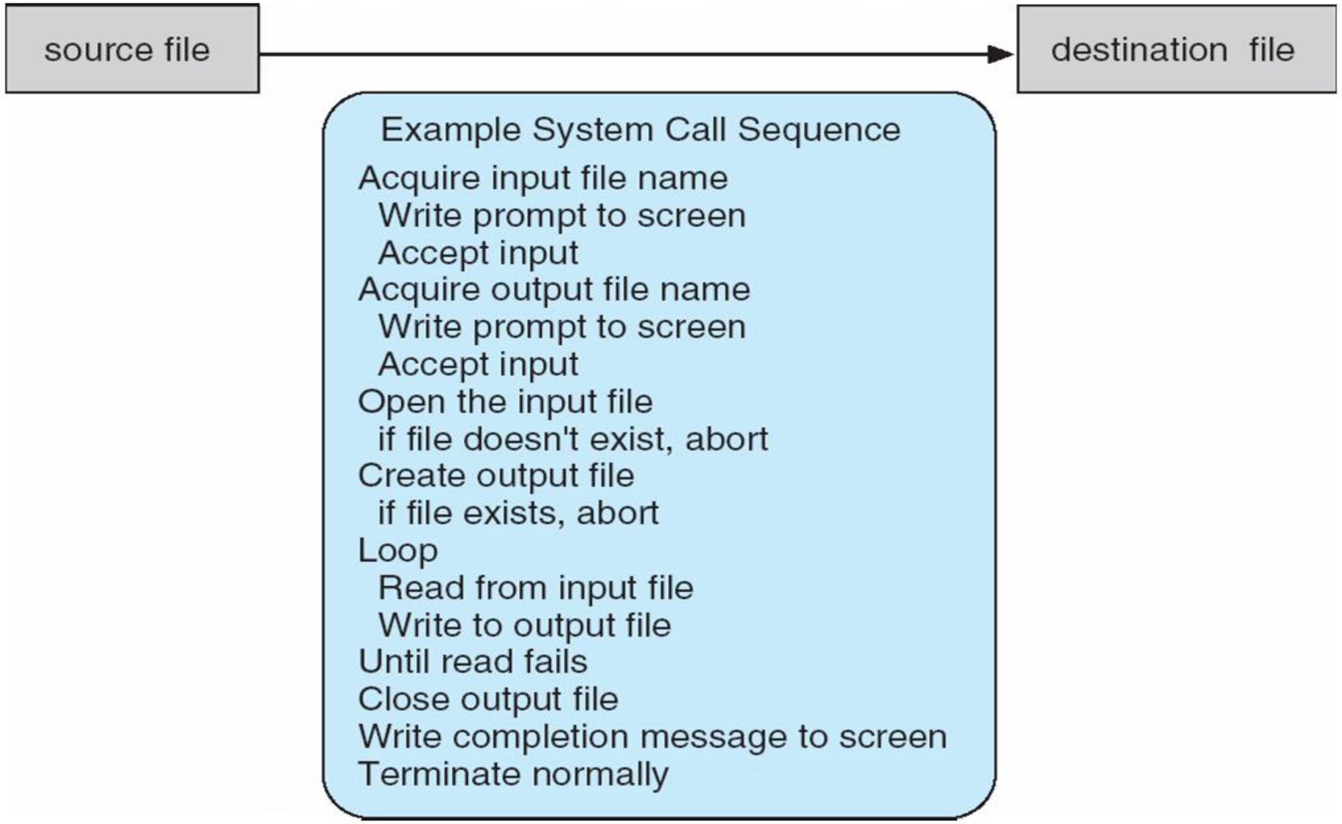
# System Calls

- 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

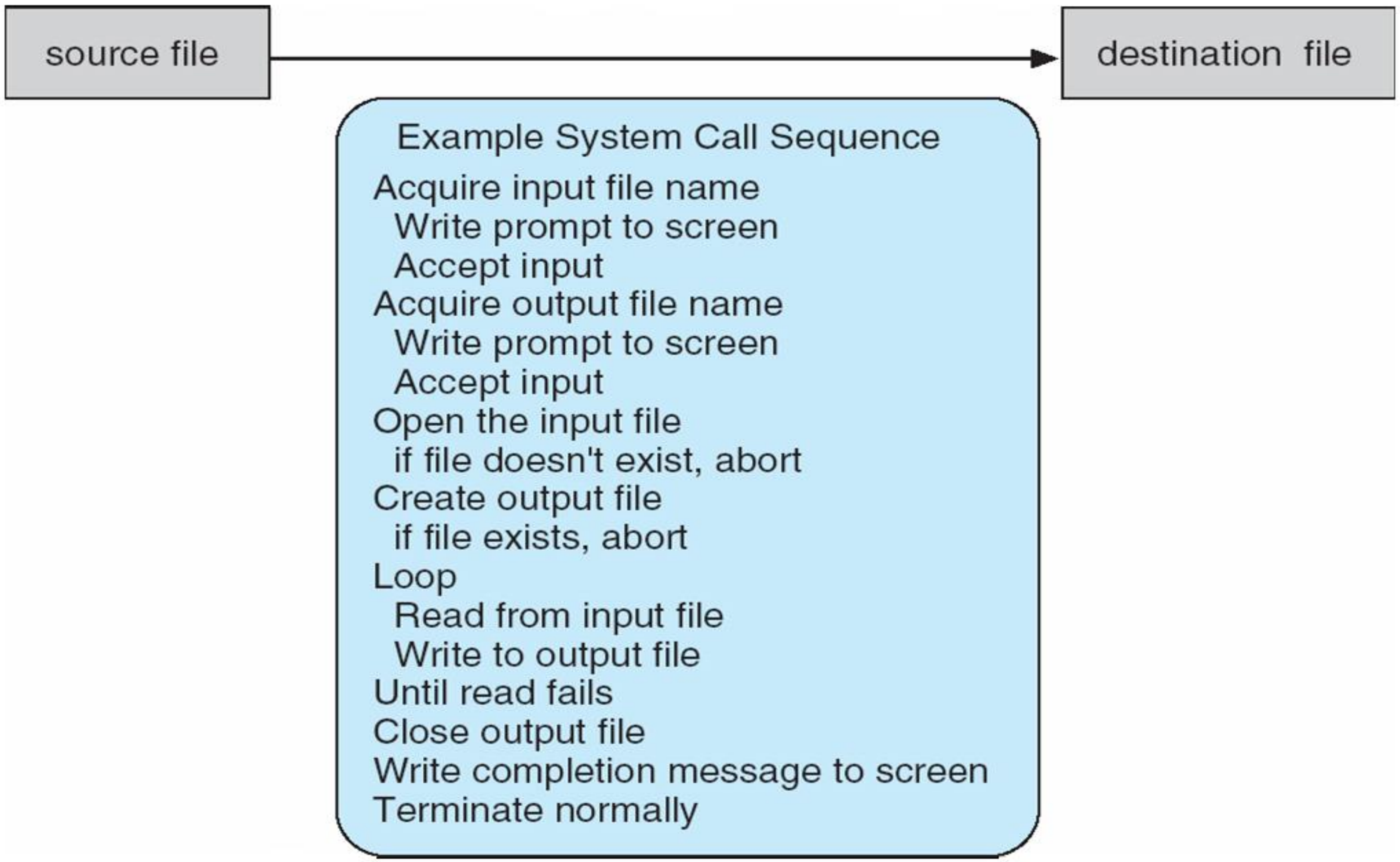
1. Win32 API for Windows
2. POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
3. Java API for the Java virtual machine (JVM)

# Example of system calls

- System call sequence to copy the content of one file to other



What system call sequence is used to copy the contents of the source file to the destination file? Note down the required system calls and provide an ordered sequence of system calls in your final answer.


source file
destination file
Example System Call Sequence
Acquire input file name
Write prompt to screen
Accept input
Acquire output file name
Write prompt to screen
Accept input
Open the input file
if file doesn't exist, abort
Create output file
if file exists, abort
Loop
Read from input file
Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally

# Example of standard APIs

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

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:

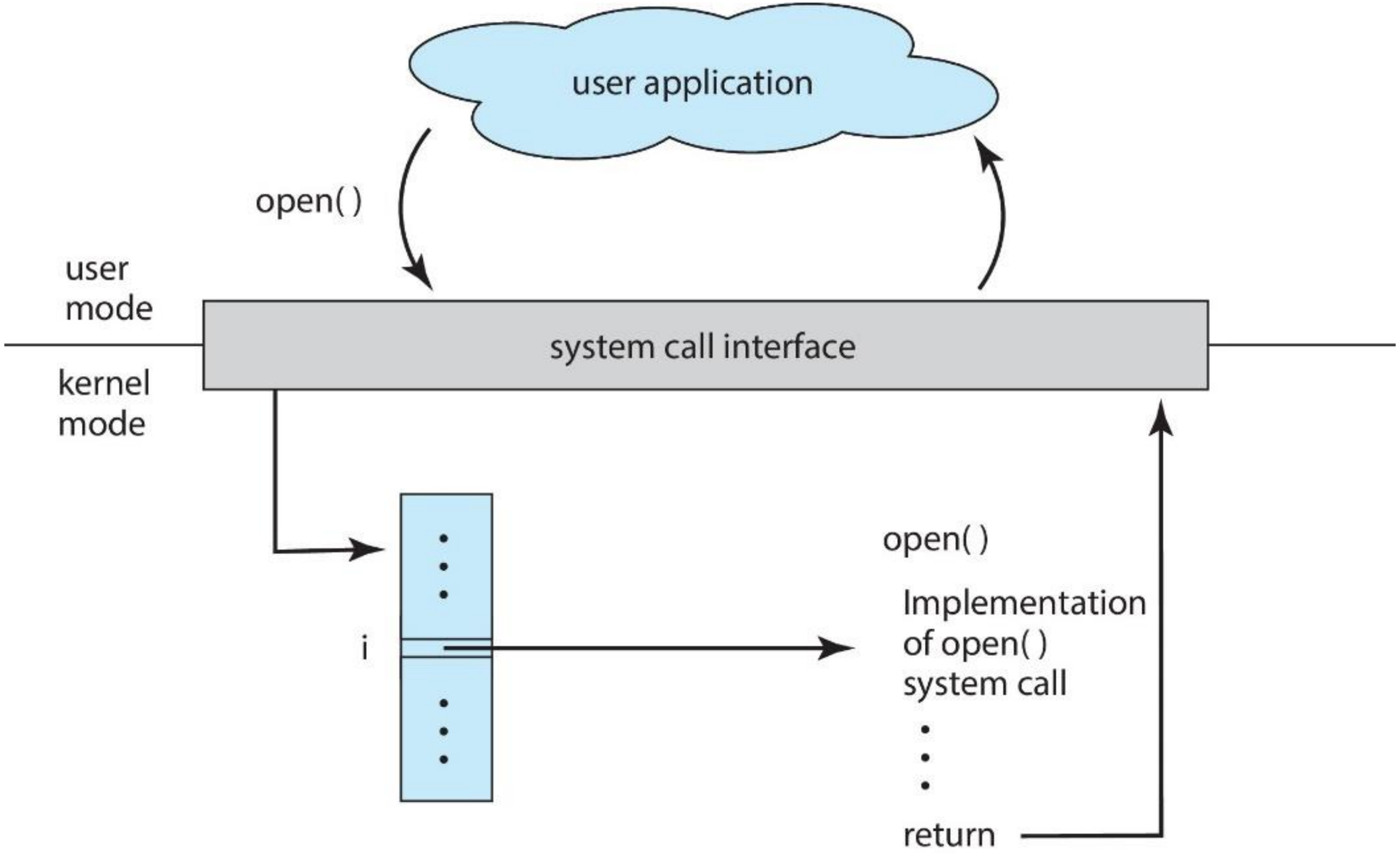
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

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.

# Systems Call Implementation

- Typically, a number is associated with each system call
  - **System-call interface maintains a table indexed according to these numbers**
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller 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)

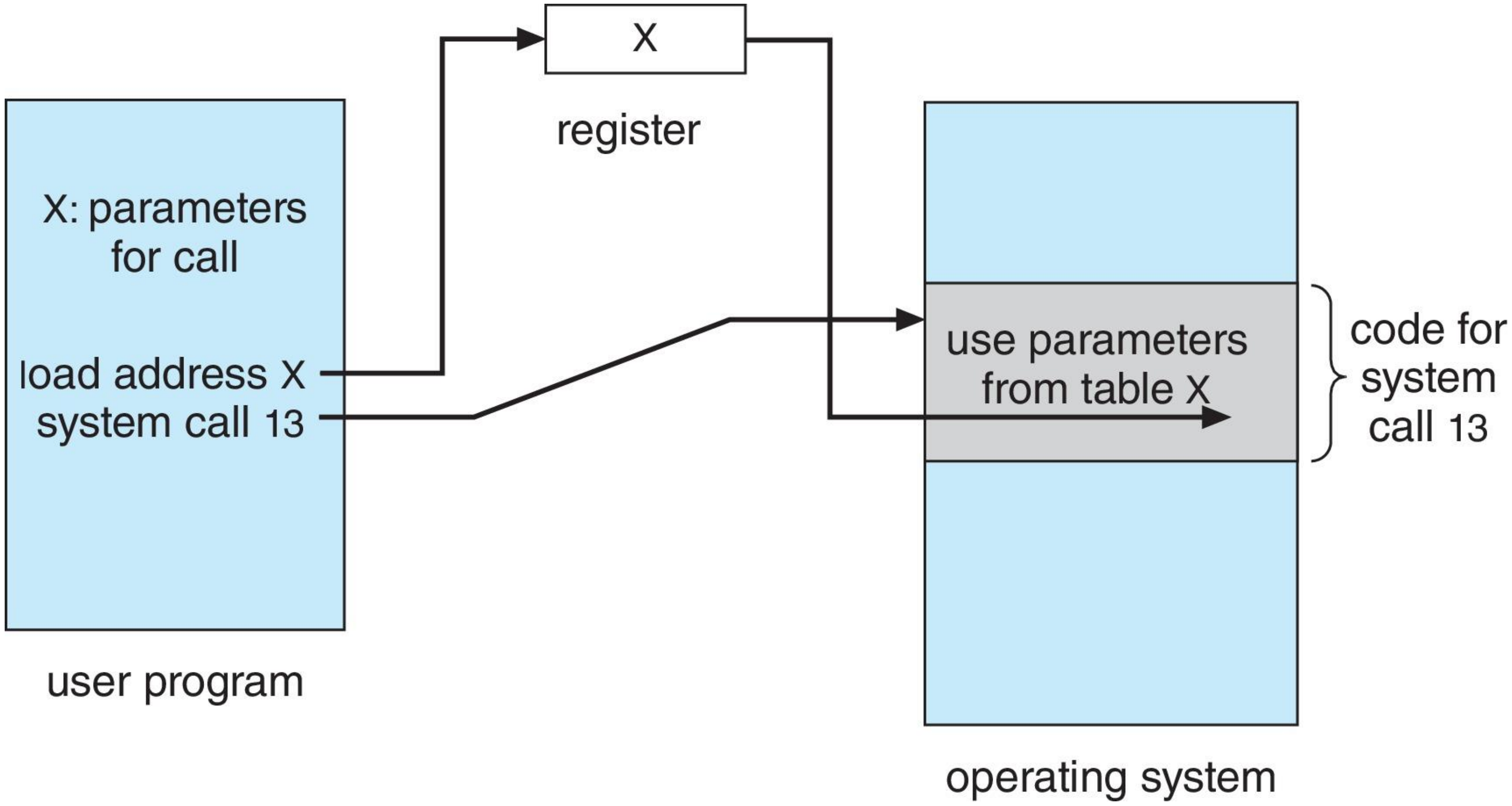
# API – system call OS relation



# System Call Parameter Passing

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

# Parameter passing via Table



# Types of System Calls

- Process control
- File Management
- Device Management
- Protection
- Information Maintenance
- Communication

# 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

- **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

**Protection**

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

# Types of System Calls

- **<u>Information maintenance</u>**
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- **<u>Communications</u>**
  - 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

# Examples of Windows and Unix System calls

***EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS***

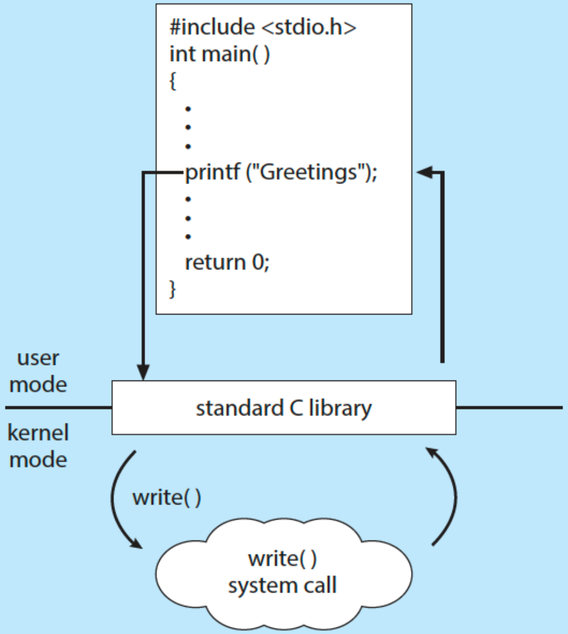
The following illustrates various equivalent system calls for Windows and UNIX operating systems.

| | **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

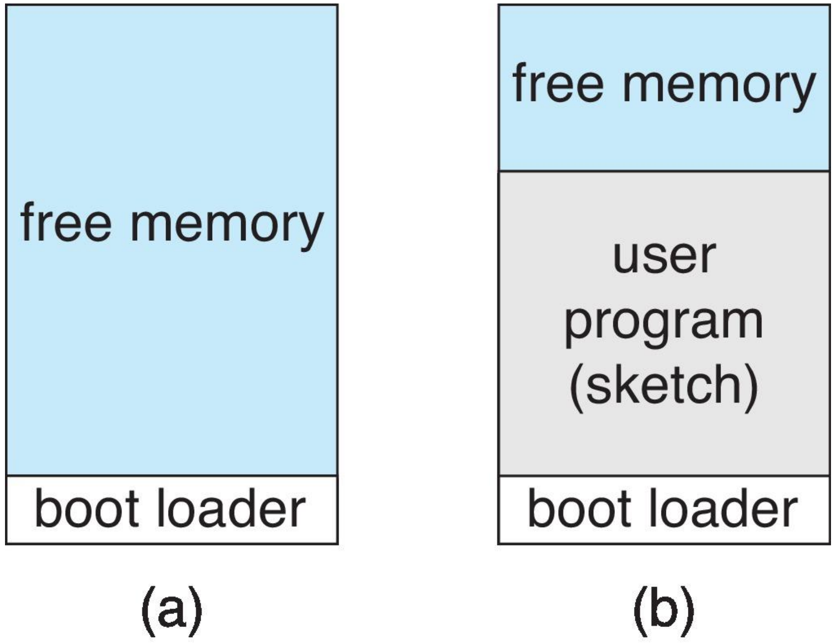
***THE STANDARD C LIBRARY***

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 Ardunio

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




At system startup running a program

# System Services

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

# System Services

- 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 Services

- **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 Services

- **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**
  - Don’t pertain to system
  - Run by users
  - Not typically considered part of OS
  - Launched by **command line, mouse click, finger poke**

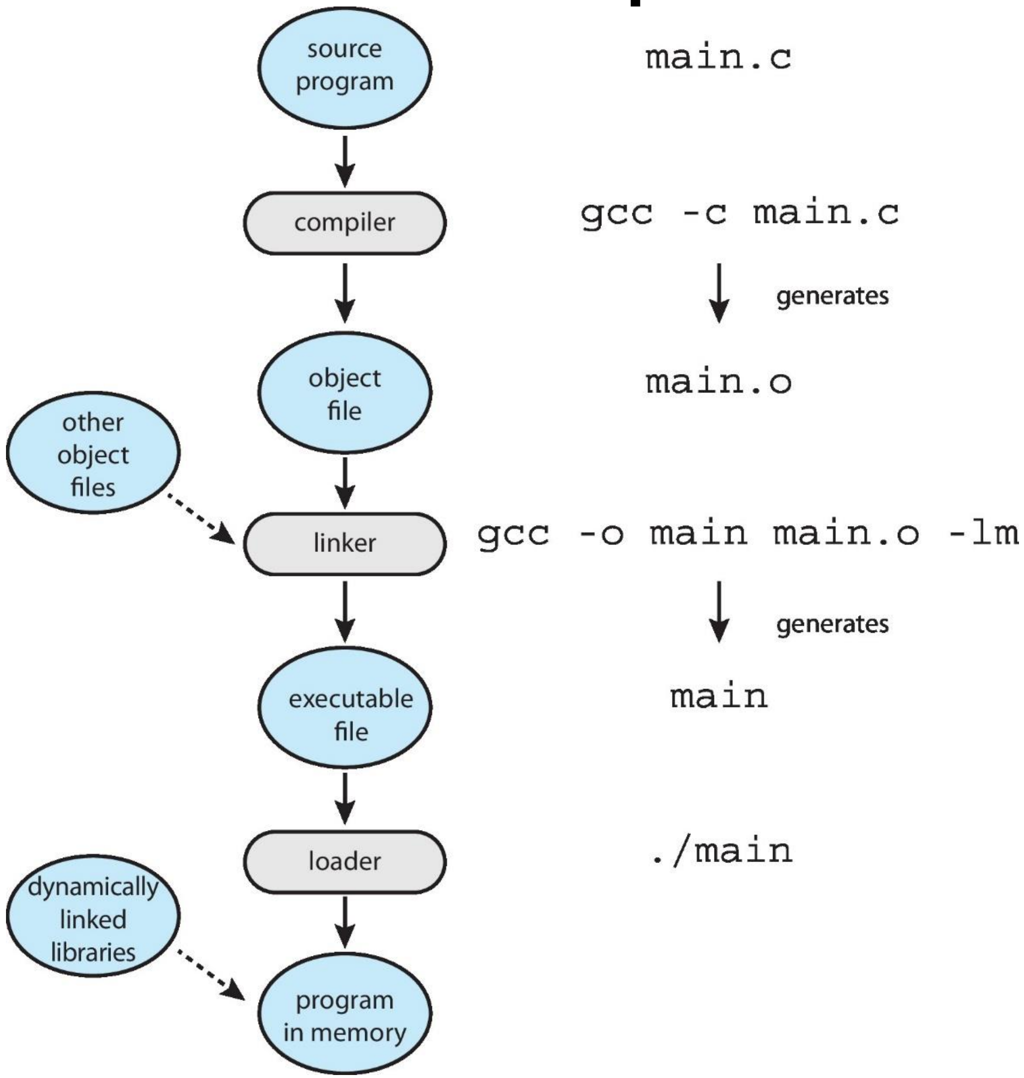
# Linkers and Loaders

- Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**
- **Linker** combines these into single binary **executable** file
  - Also brings in libraries
- Program resides on secondary storage as binary executable

# Linkers and Loaders

- 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
- Modern general purpose systems don't link libraries into executables
  - 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)
- Object, executable files have standard formats, so operating system knows how to load and start them

# Linker and Loaders Example



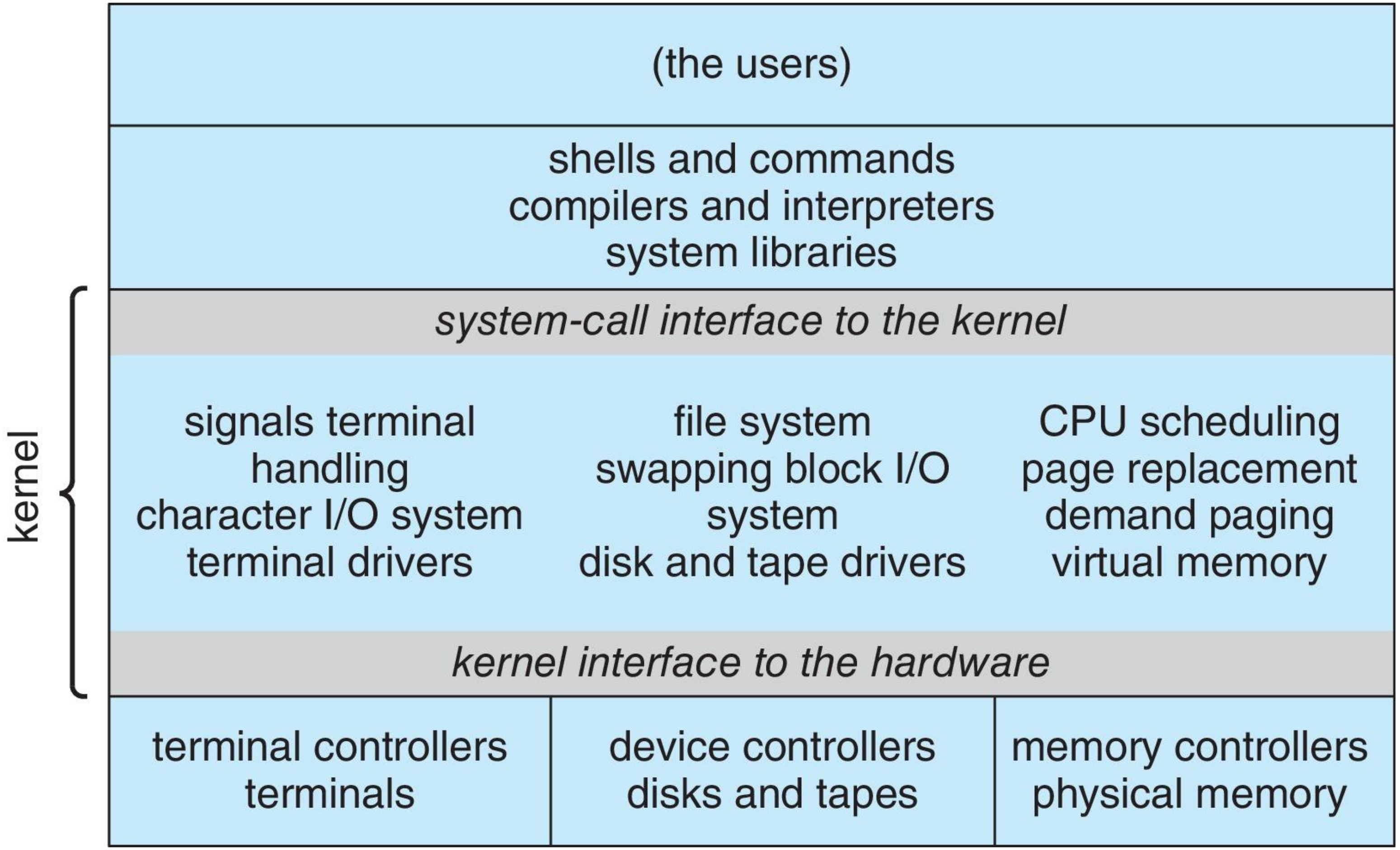
# Operating System Strcture

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

# Monolithic Structure – Original UNIX

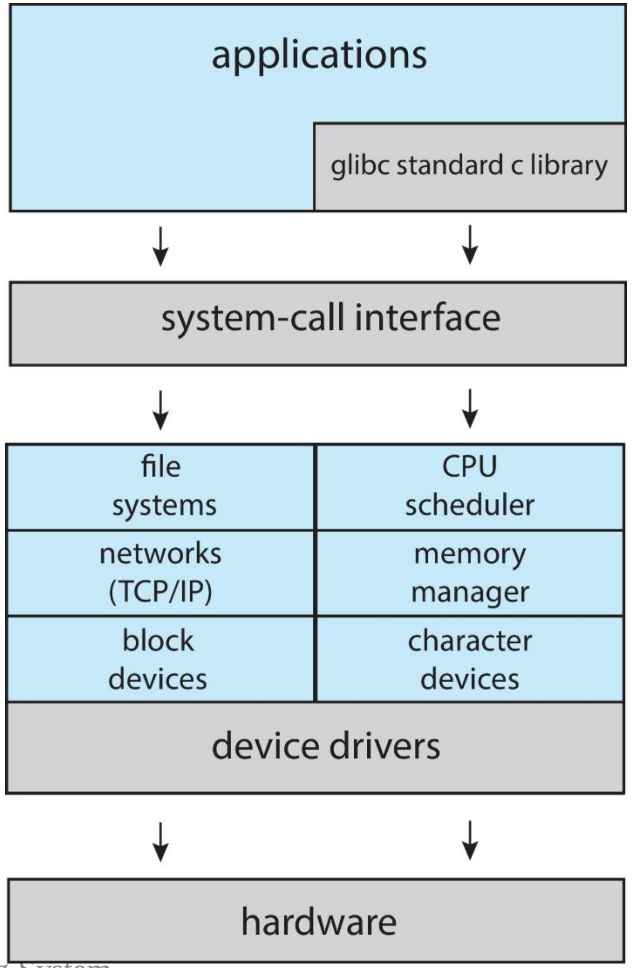
- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring.
- The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

# Traditional UNIX System Structure



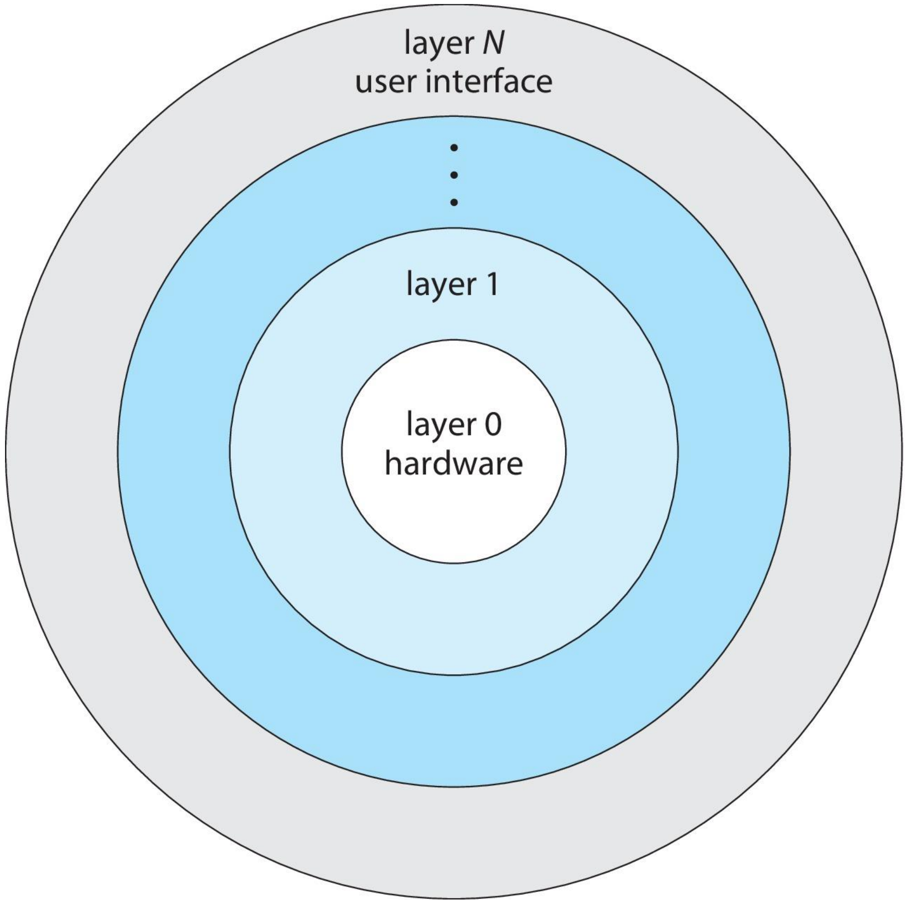
# Linux System Structure

- Monolithic + Modular Design



# Layered Approach

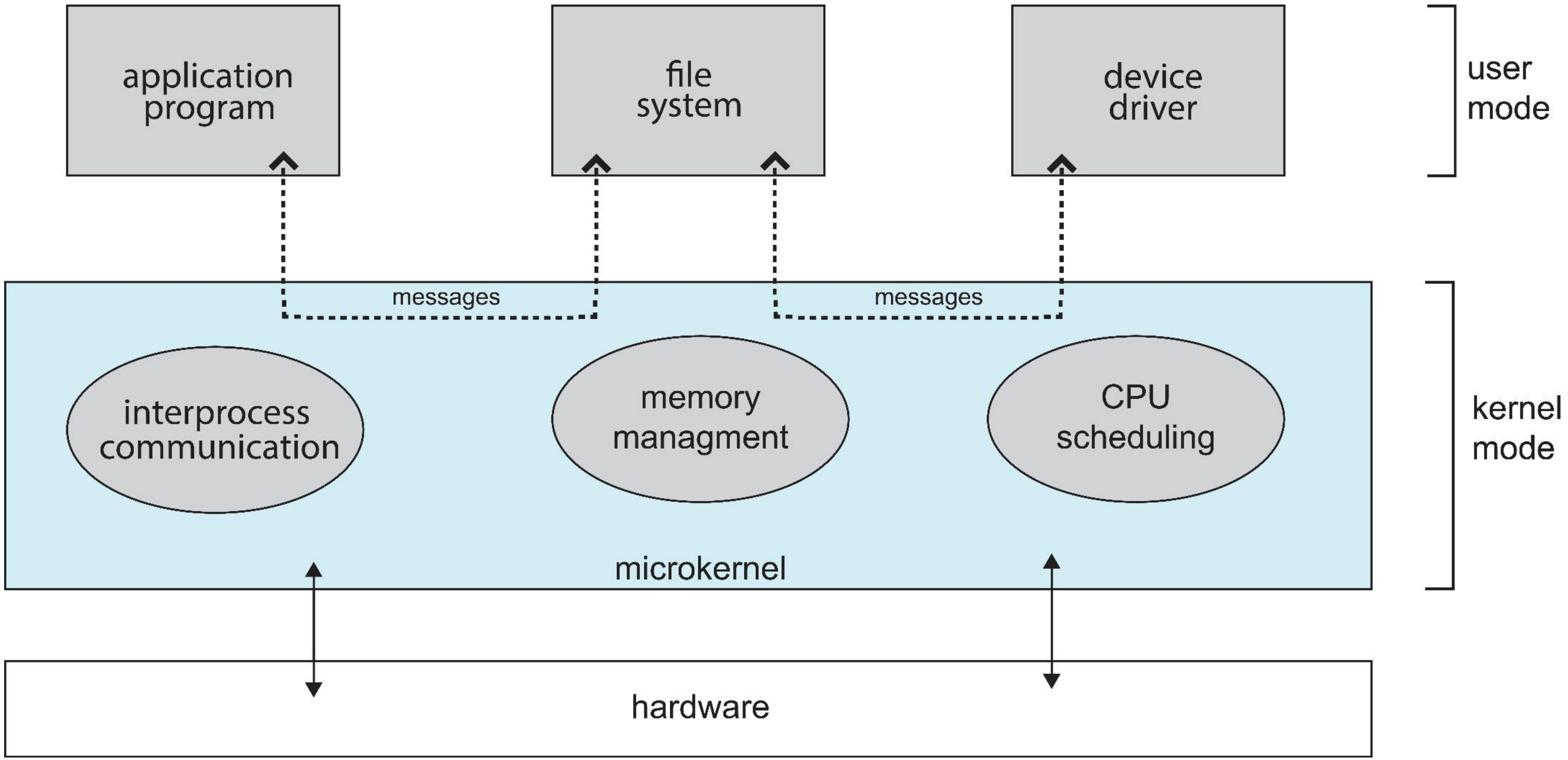
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level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



# Micro Kernels

- 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

# Micro Kernel System Structure



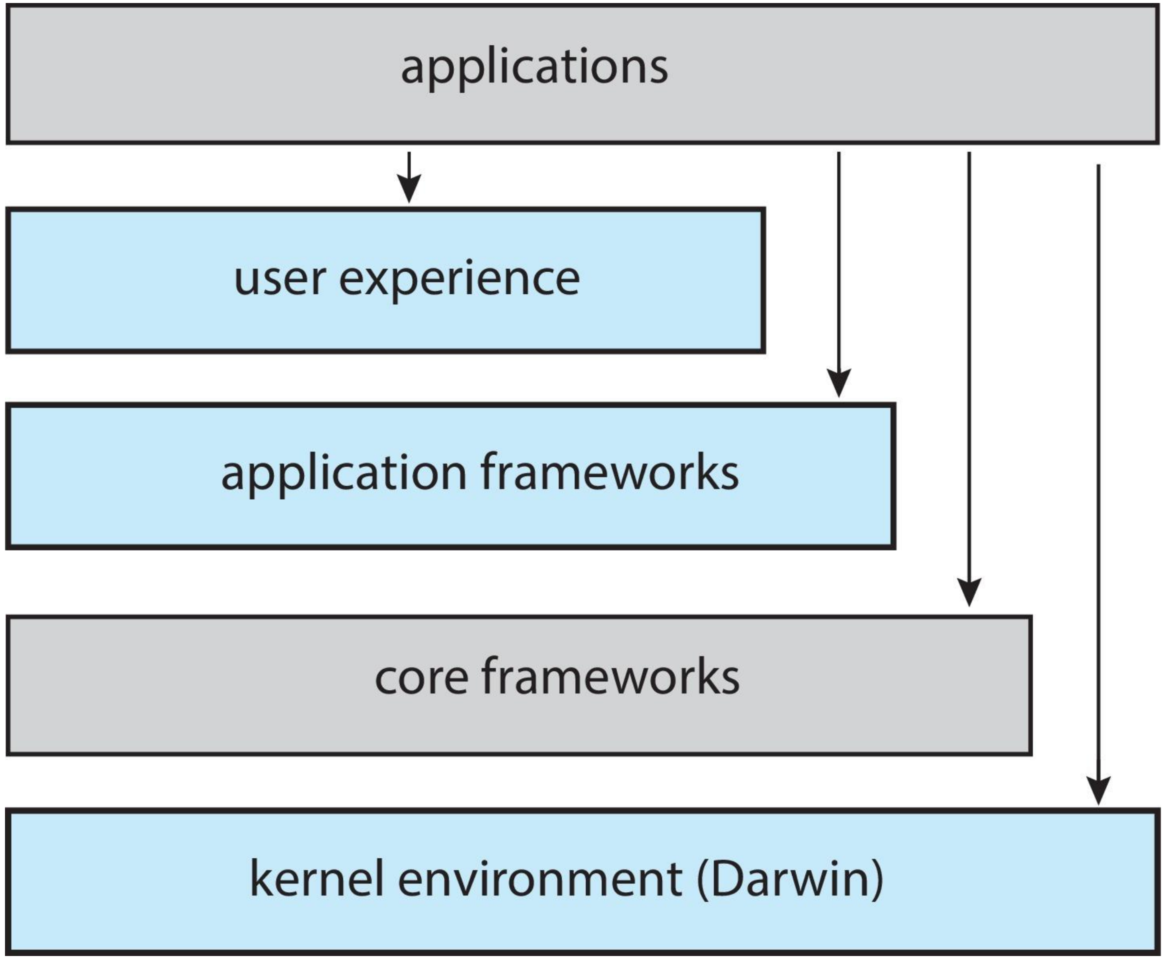
# Modules

- Many modern operating systems implement **loadable kernel** modules **(LKMs)**
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
  - Linux, Solaris, etc.

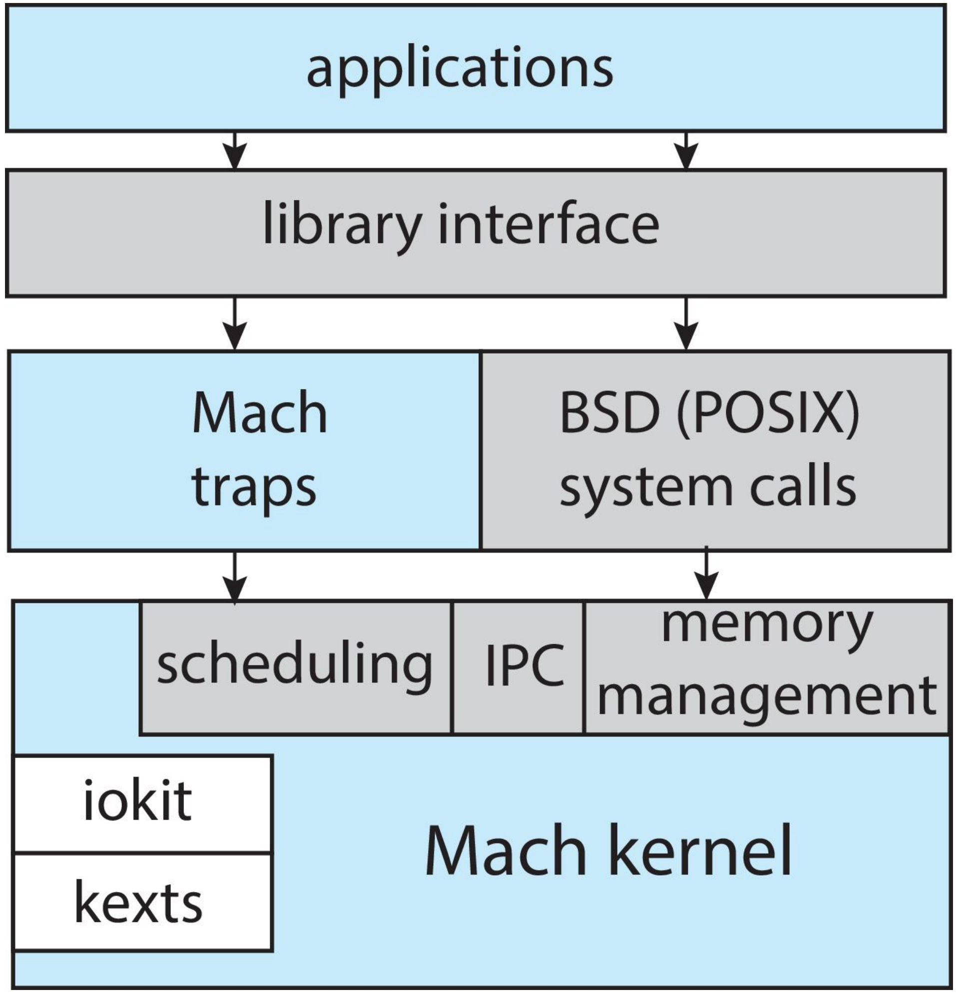
# Hybrid Systems

- Most modern operating systems are not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux 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**)

# macOS and iOS



# Darwin



applications
library interface
Mach
traps
BSD (POSIX)
system calls
scheduling
IPC
memory
management
iokit
kexts
Mach kernel

# 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 thnn runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia

# Android Architecture