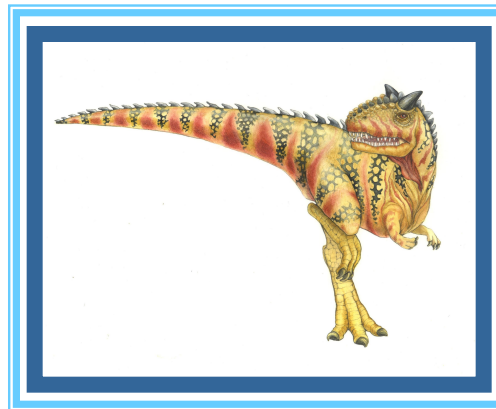# Chapter 2: Operating-System Structures

Chapter 2: Operating-System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Operating System Debugging
- Operating System Generation
- 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

- To explain how operating systems are installed and customized and how they boot

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**).
        - 4 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

    4 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

    4 May occur in the CPU and memory hardware, in I/O devices, in user program

    4 For each type of error, OS should take the appropriate action to ensure correct and consistent computing

    4 Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system
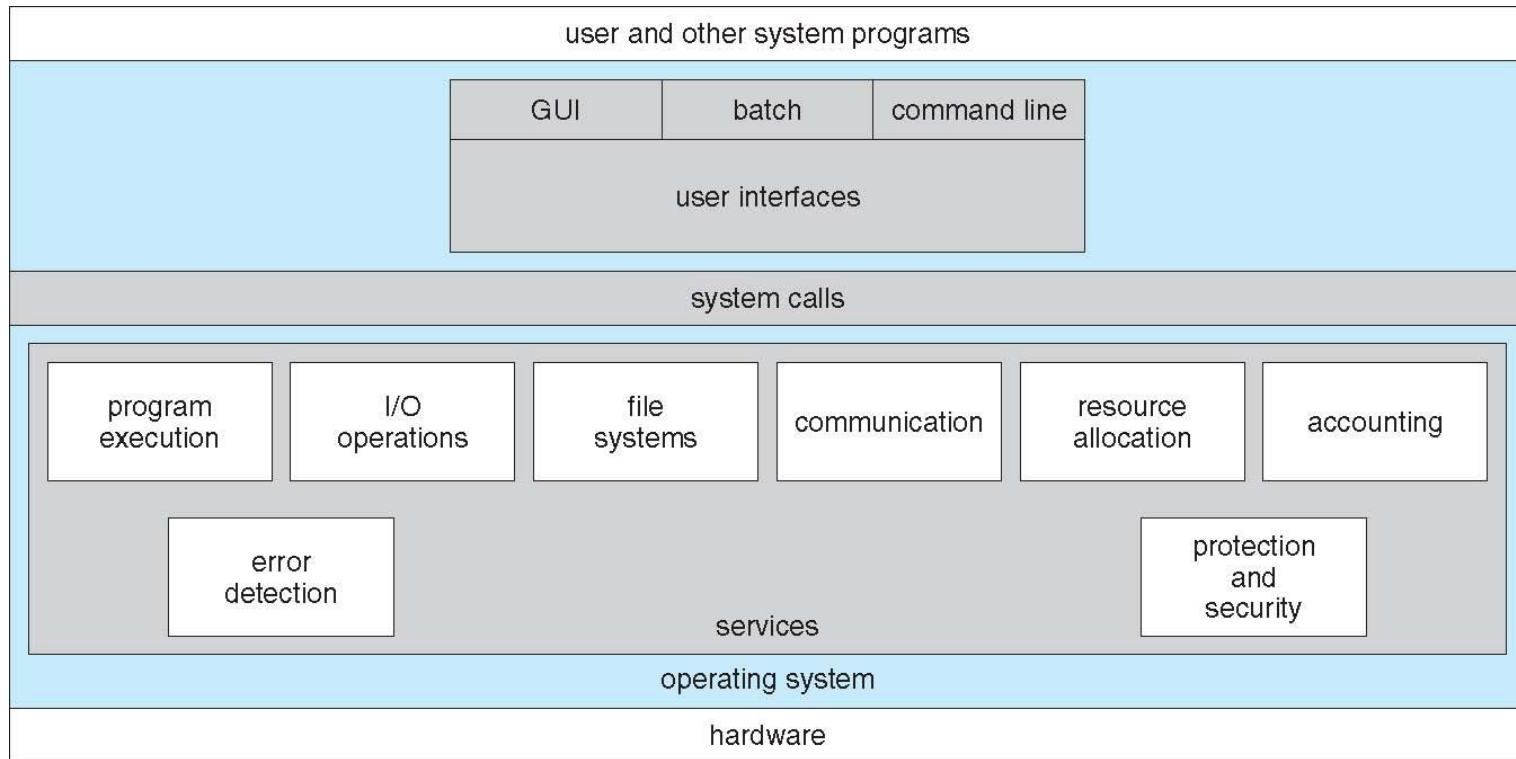
Operating System Services (Cont.)

- 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
    - 4 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
    - 4 **Protection** involves ensuring that all access to system resources is controlled
    - 4 **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

user and other system programs

GUI	batch	command line
user interfaces		

system calls

program execution	I/O operations	file systems	communication	resource allocation	accounting

error detection	protection and security
services	

operating system

hardware

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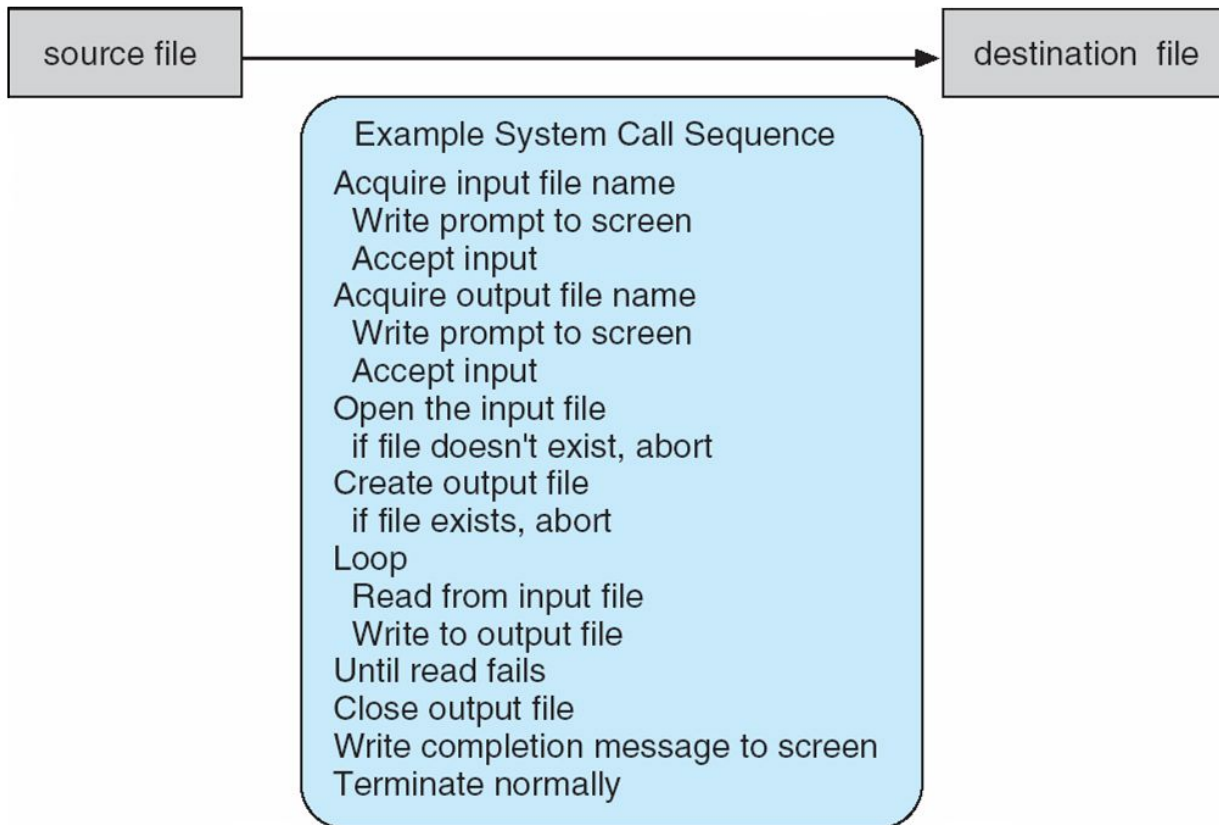
Note that the system-call names used throughout this text are generic

Example of System Calls

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

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 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        function              parameters
  value         name
```

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
    - 4 Managed by run-time support library (set of functions built into libraries included with compiler)
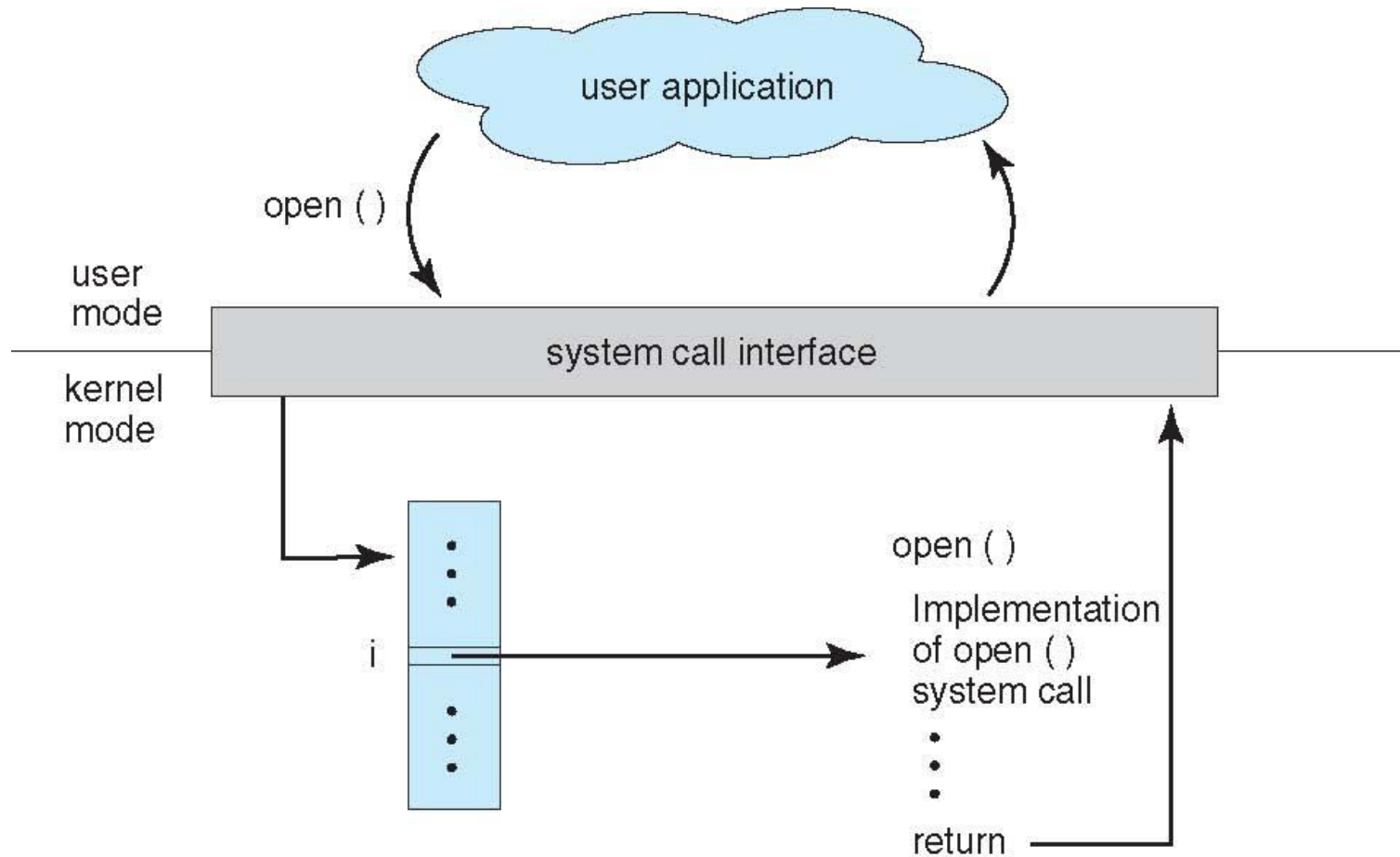
System Call Parameter Passing

- 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
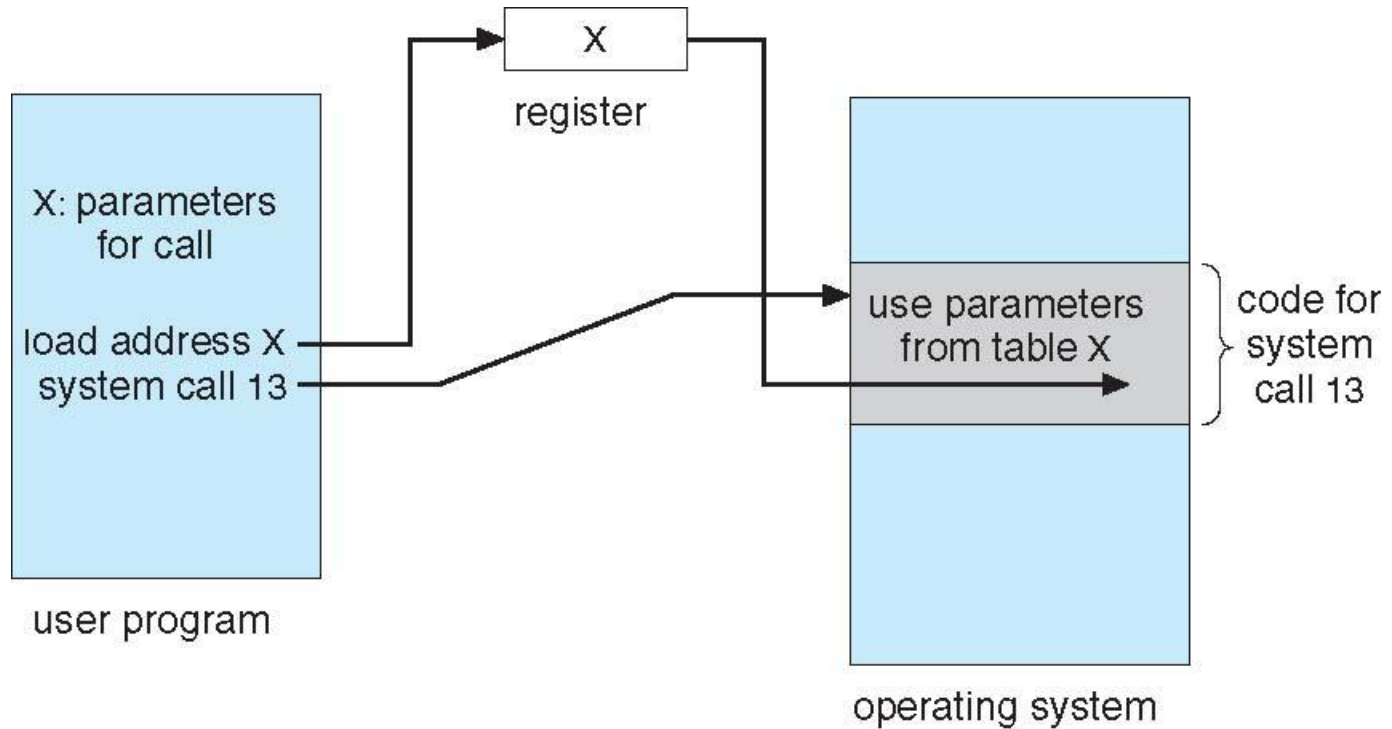
Types of System Calls

- 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

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**
        4 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

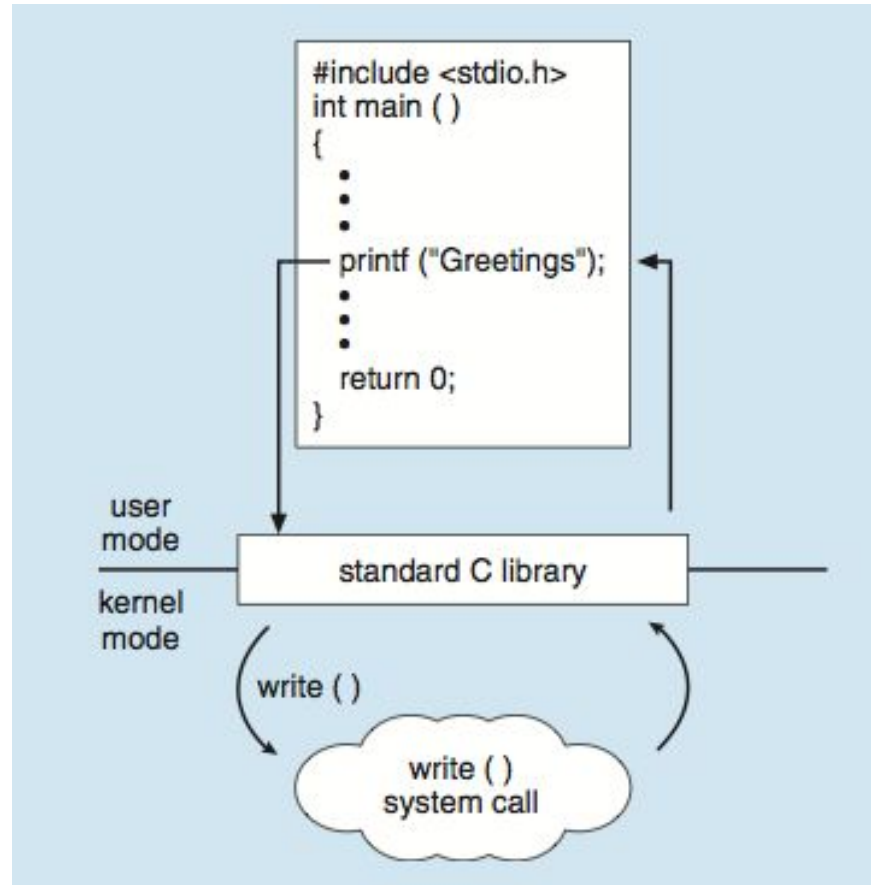# Examples of Windows and Unix System Calls

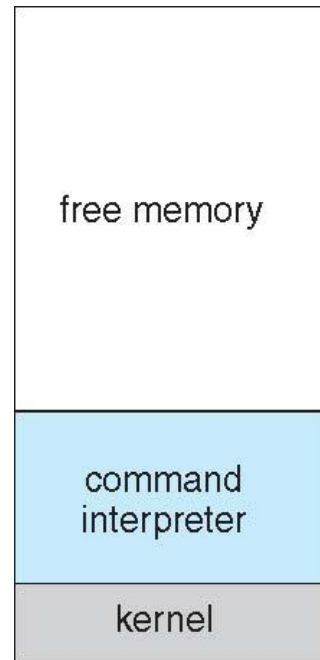|	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information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

Standard C Library Example

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


```
#include <stdio.h>
int main ( )
{
    •
    •
    •
    printf ("Greetings");
    •
    •
    •
    return 0;
}
```

user mode
kernel mode

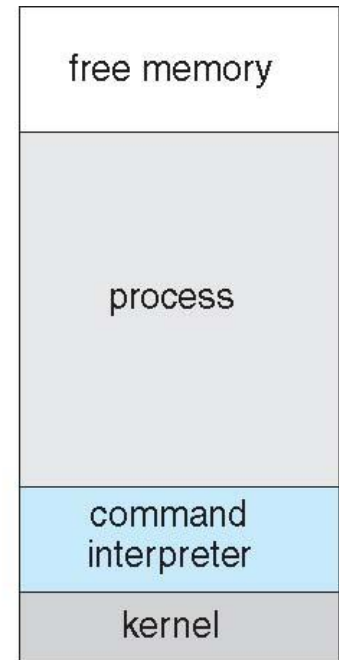standard C library

write ( )

write ( )
system call

Example: MS-DOS

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

free memory

command
interpreter

kernel

(a)

free memory

process

command
interpreter

kernel

(b)

At system startup   running a program

Example: FreeBSD

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

| process D |
|-----------|
| free memory |
| process C |
| interpreter |
| process B |
| kernel |
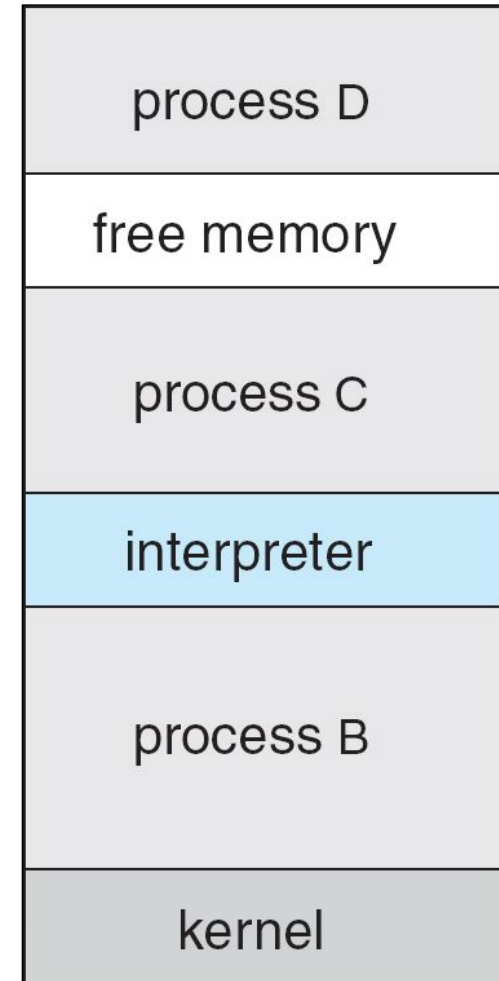
Operating System Structure

- General-purpose OS is very large program

- Various ways to structure ones

  - Simple structure – MS-DOS

  - More complex -- UNIX

  - Layered – an abstrcation

  - Microkernel -Mach

Simple Structure -- MS-DOS

- 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
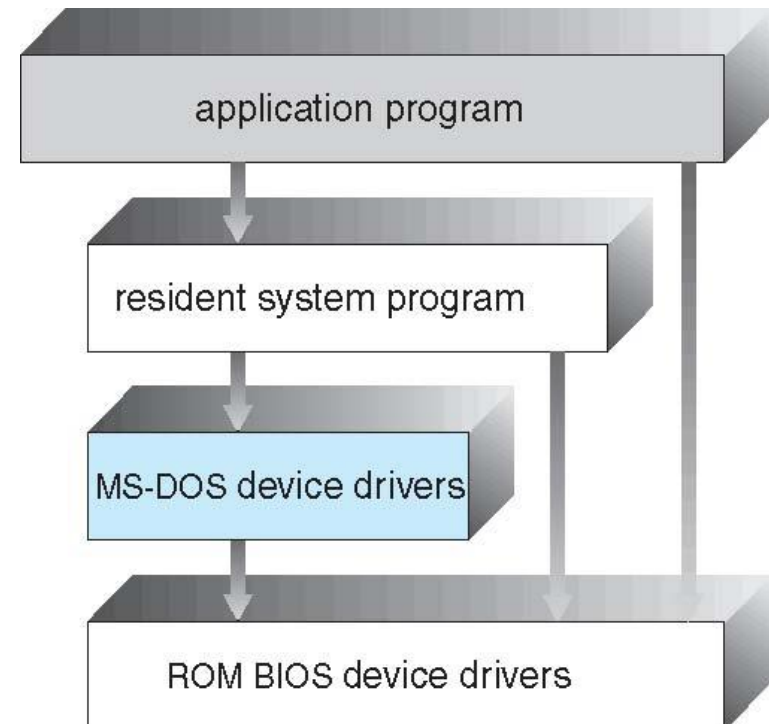
Non Simple Structure -- UNIX

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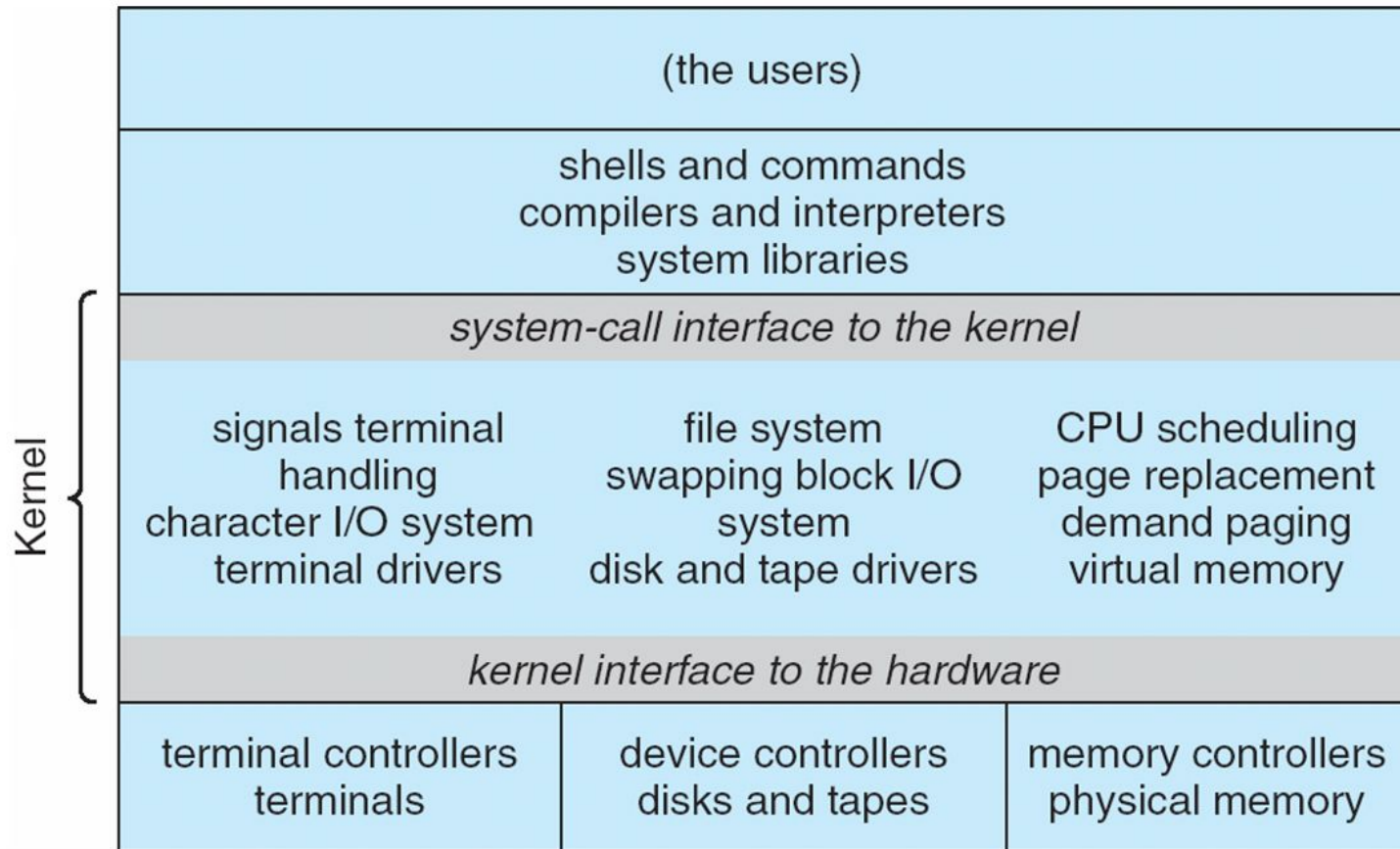
- Systems programs
- The kernel
  - Consists of everything below the system-call interface and above the physical hardware
  - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

Traditional UNIX System Structure

Beyond simple but not fully layered

(the users)		
shells and commands compilers and interpreters system libraries		
system-call interface to the kernel		
signals terminal handling character I/O system terminal drivers	file system swapping block I/O system disk and tape drivers	CPU scheduling page replacement demand paging virtual memory
kernel interface to the hardware		
terminal controllers terminals	device controllers disks and tapes	memory controllers physical memory

Kernel (spans from system-call interface to kernel interface to the hardware)

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
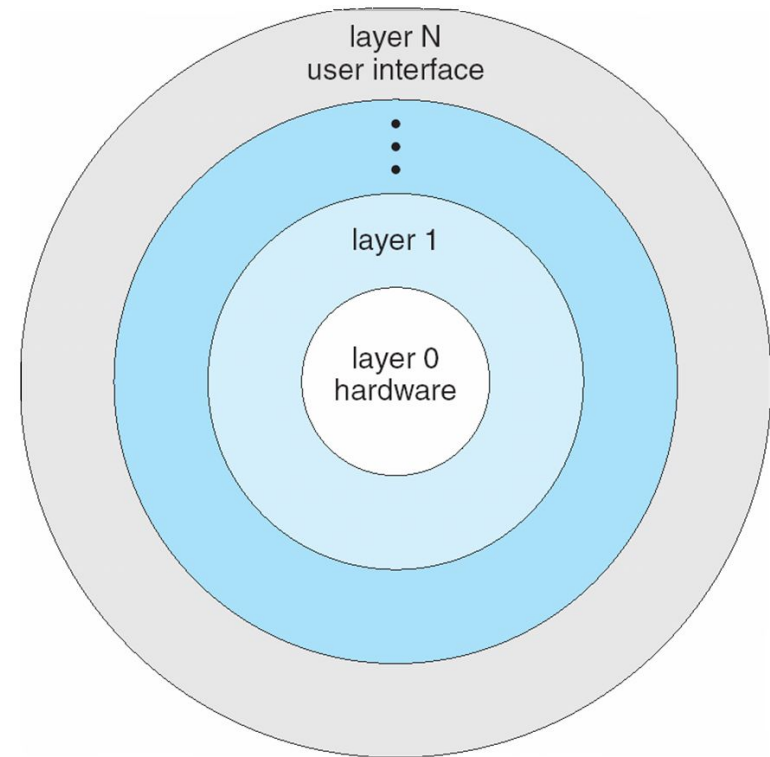
Microkernel System Structure

- Moves as much from the kernel into user space

- **Mach** 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
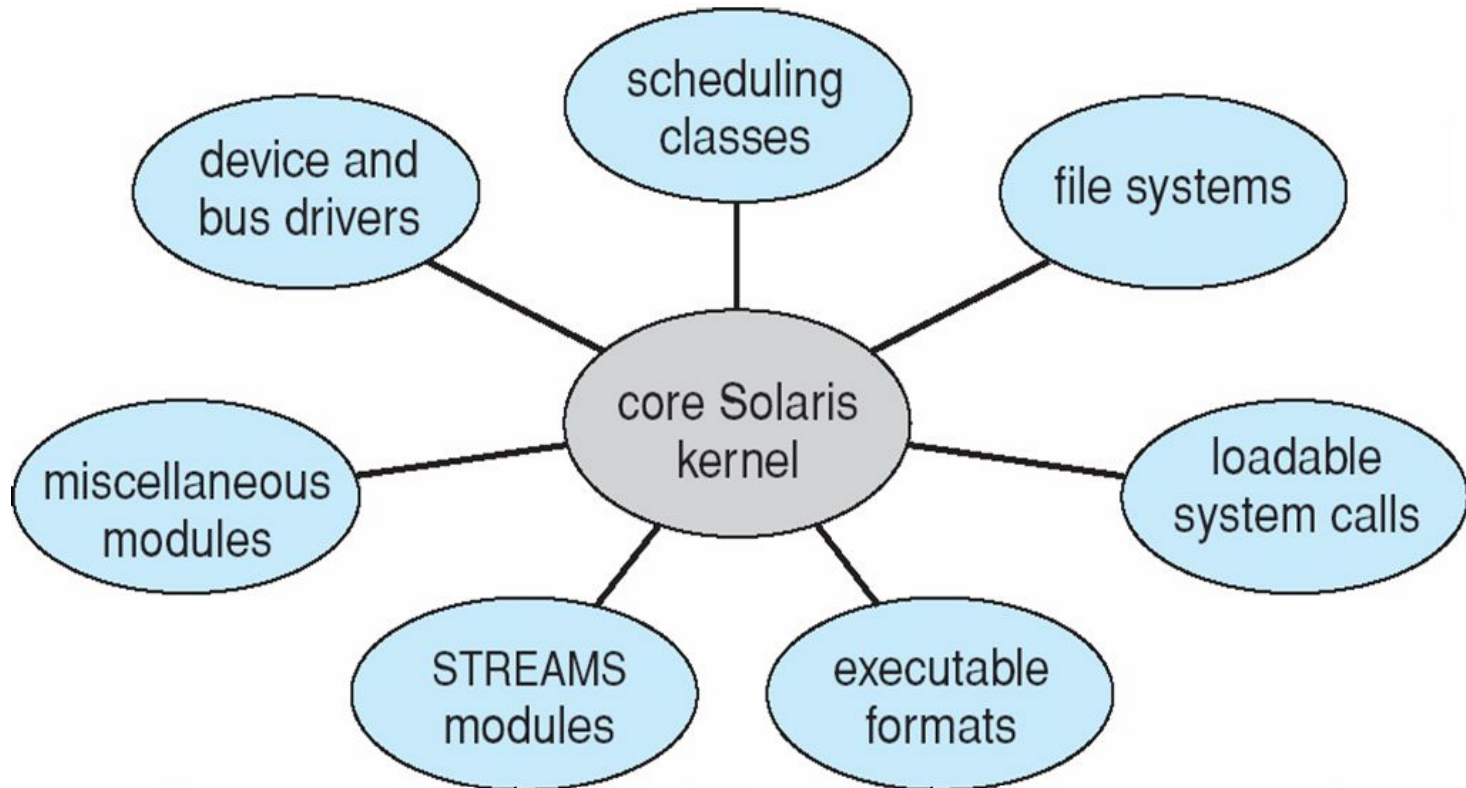
Microkernel System Structure

2.29
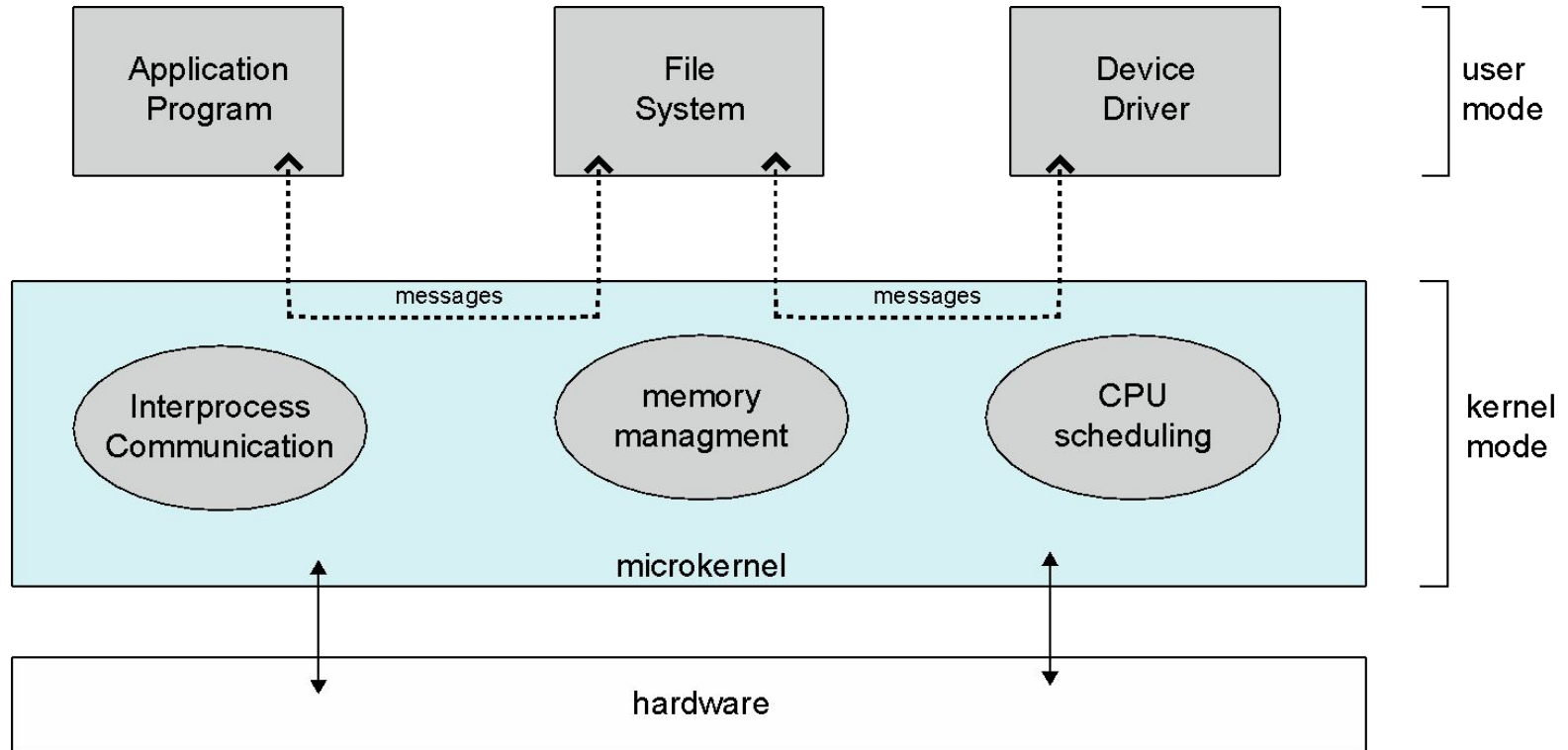
Modules

- Many modern operating systems implement **loadable kernel modules**
  - 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
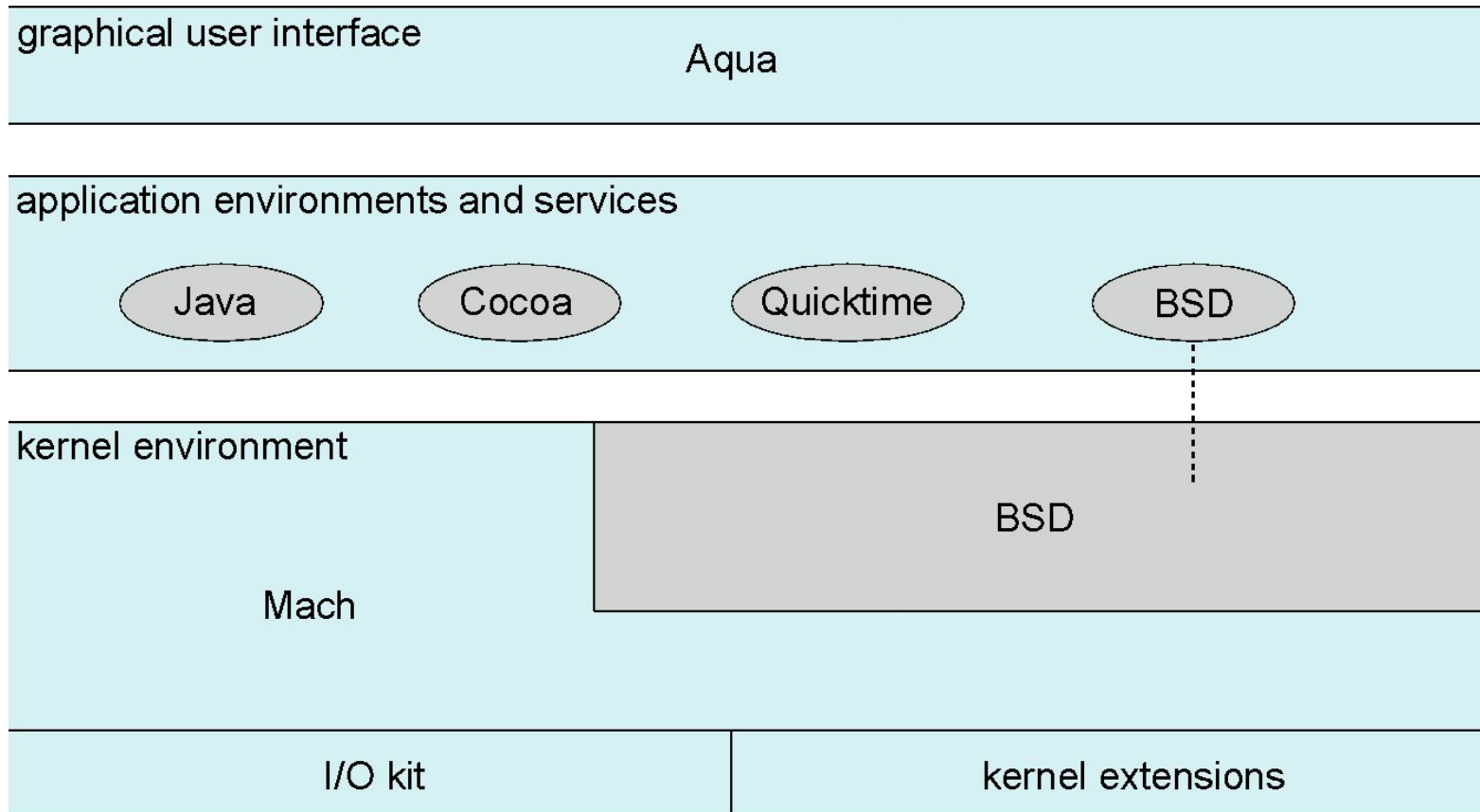
Solaris Modular Approach

Hybrid Systems

- Most modern operating systems are actually 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**)

Mac OS X Structure

graphical user interface			
Aqua			

application environments and services			
Java	Cocoa	Quicktime	BSD

kernel environment		
		BSD
	Mach	
I/O kit	kernel extensions	

iOS

- Apple mobile OS for *iPhone*, *iPad*
  - Structured on Mac OS X, added functionality
  - Does not run OS X applications natively
    - 4 Also runs on different CPU architecture (ARM vs. Intel)
  - **Cocoa Touch** Objective-C API for developing apps
  - **Media services** layer for graphics, audio, video
  - **Core services** provides cloud computing, databases
  - Core operating system, based on Mac OS X kernel

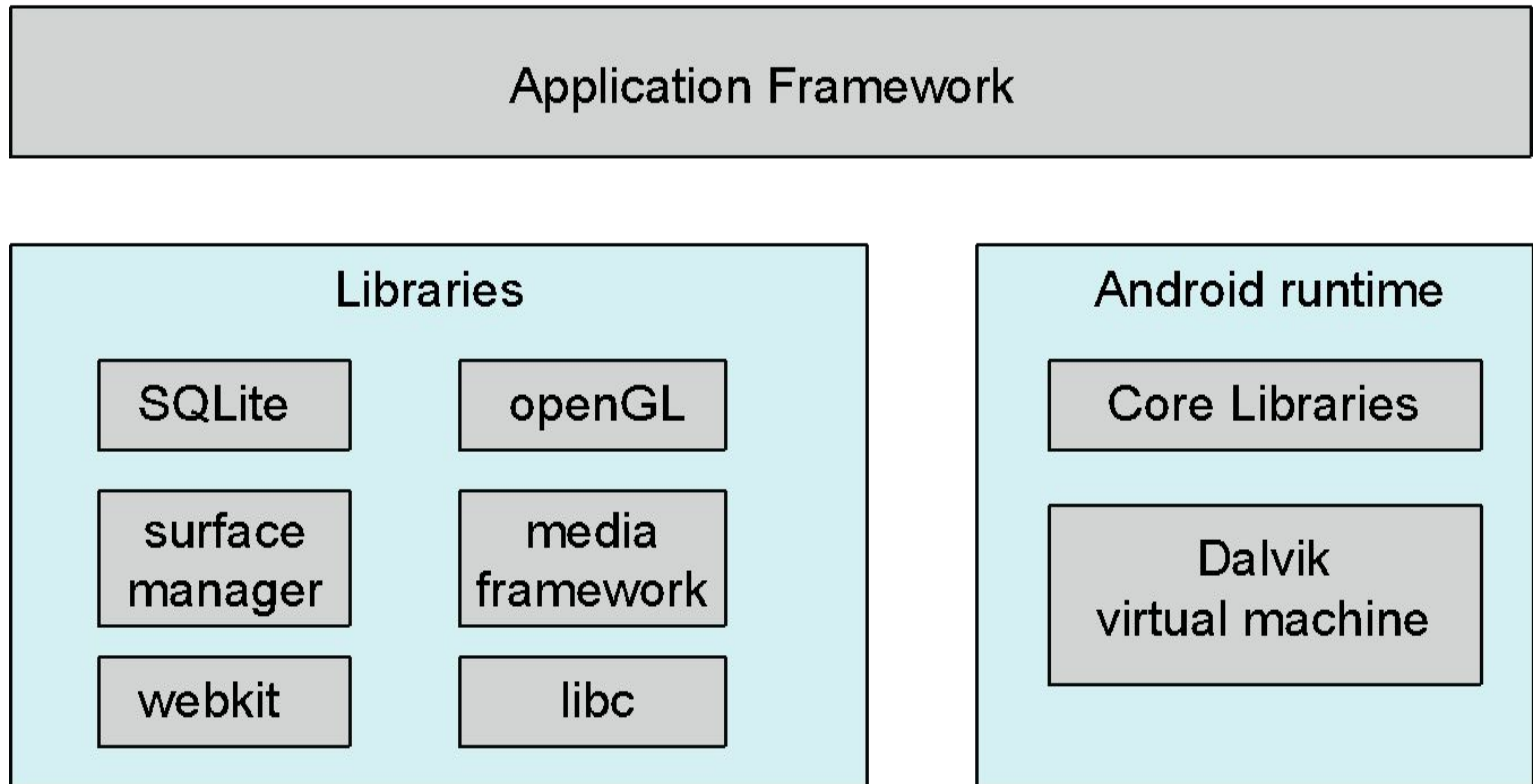| Cocoa Touch |
| :---: |
| Media Services |
| Core Services |
| Core OS |

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
    - 4 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

Android Architecture

Application Framework

Libraries
- SQLite
- openGL
- surface manager
- media framework
- webkit
- libc

Android runtime
- Core Libraries
- Dalvik virtual machine

End of Chapter 2