System Structures

Operating Systems (CS-220)
Spring 2021, FAST NUCES

COURSE SUPERVISOR: ANAUM HAMID

anaum.hamid@nu.edu.pk

Roadmap

1. **Operating-System Services**
2. **User Operating-System Interfaces**
3. **System Calls**
4. **Types of System Calls**
5. **System Programs**
6. **OS design and Implementation**
7. **Operating System Structures**
8. **System Debugging**
9. **System Generation**
10. **System Boot**

Operating System Services

- 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** - The OS is responsible for transferring data to and from I/O devices, including keyboards, terminals, printers, and storage devices.

Operating System Services

- **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.

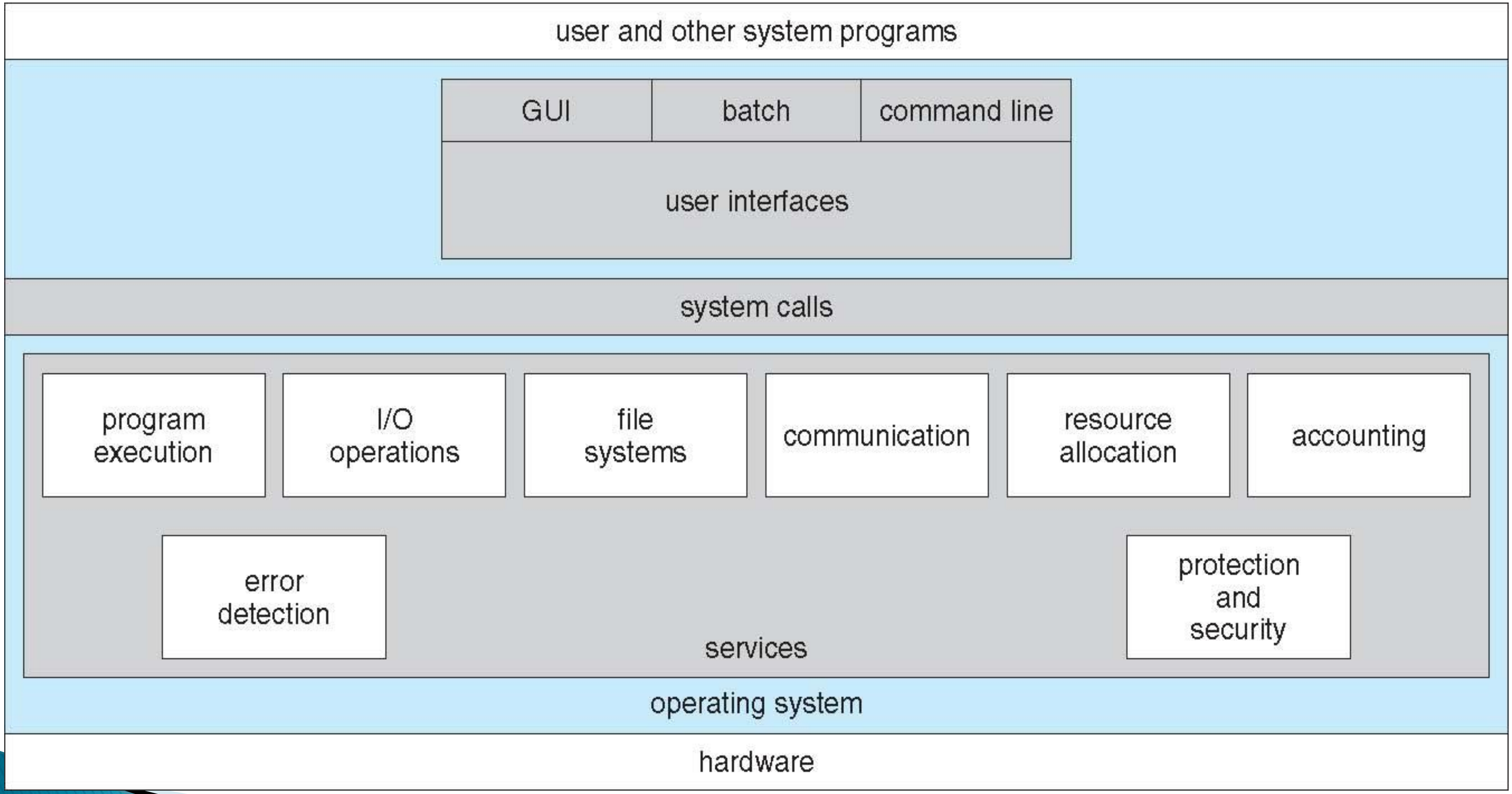
- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS).

- **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

Operating System Services

- **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.
  - E.g., CPU cycles, main memory, storage space, and peripheral devices. Some resources are managed with generic systems and others with very carefully designed and specially tuned systems, customized for a particular resource and operating environment.
- **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

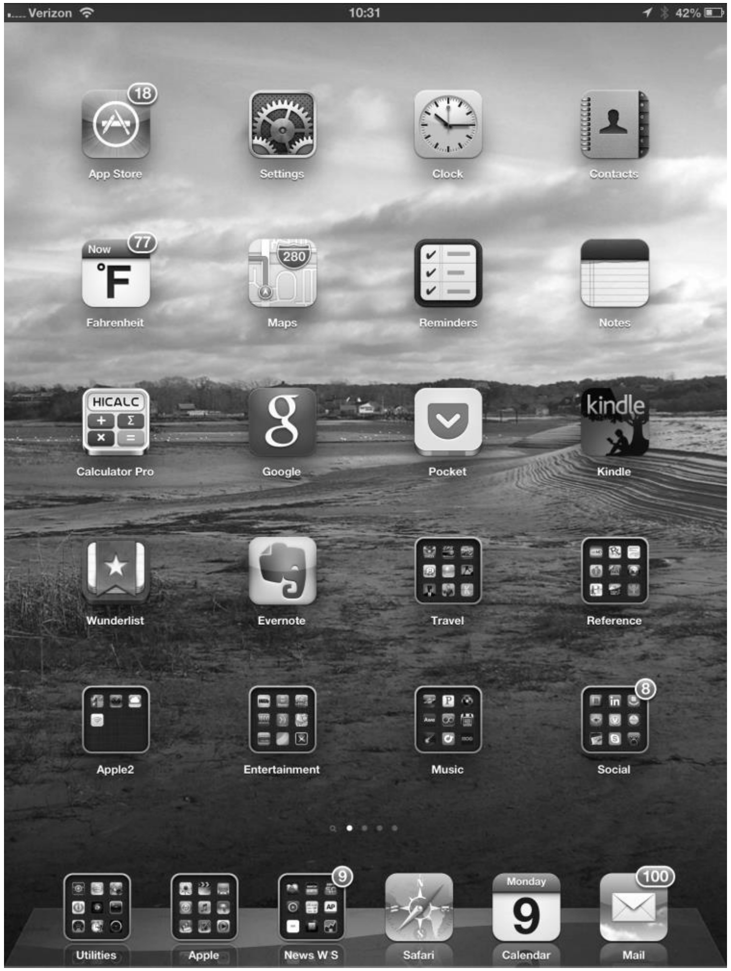
- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs

User Operating System Interface - GUI

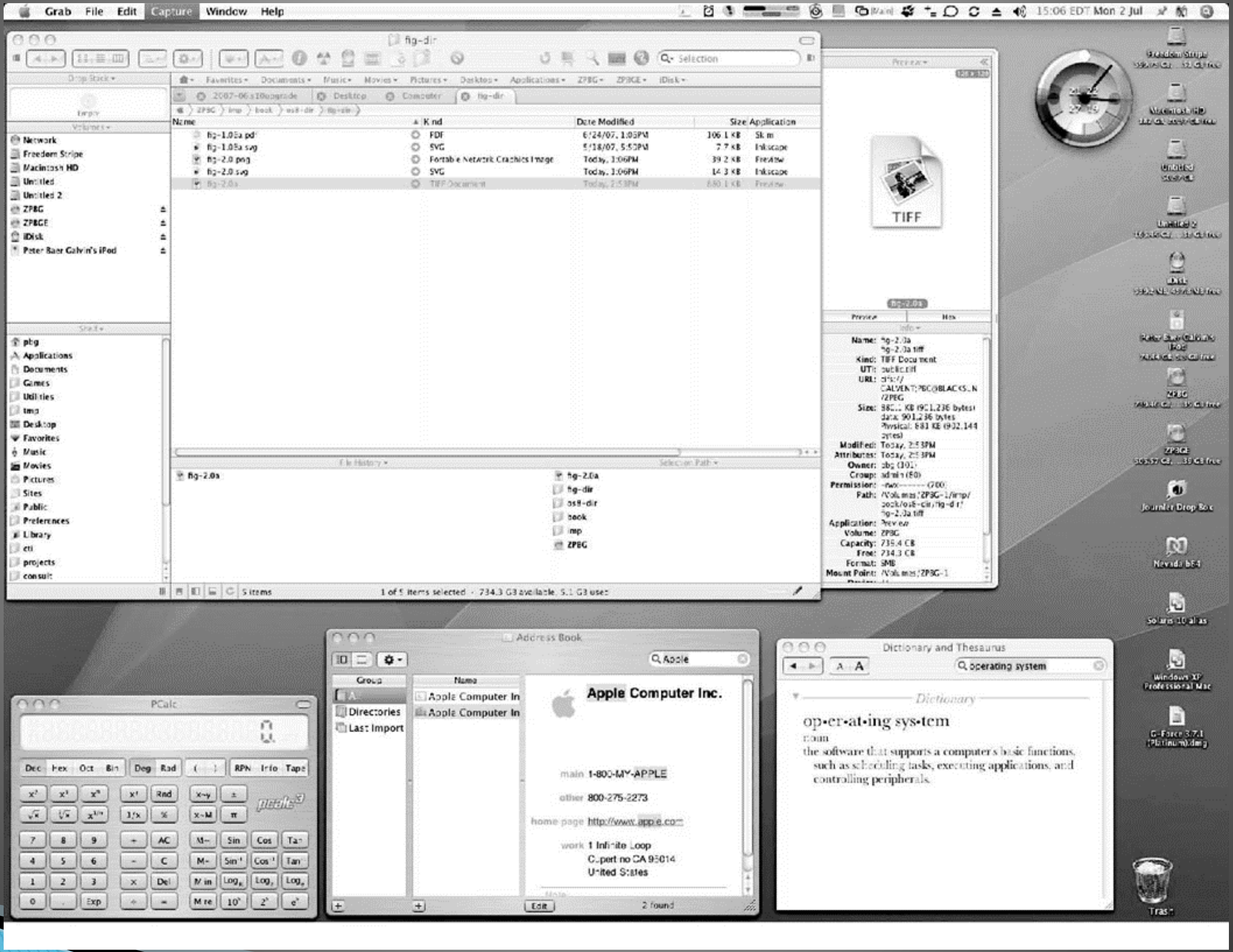
- User-friendly desktop metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
- Many systems now include both CLI and GUI interfaces
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)
  - Common Desktop Environment (**CDE**) – Unix based
  - K Desktop Environment (KDE) – Linux based
  - GNOME Unix variant

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



Name	Kind	Date Modified	Size	Application
fig-1.05a pdf	PDF	6/24/07, 1:05PM	106.1 KB	Skim
fig-1.05a svg	SVG	5/18/07, 5:53PM	7.7 KB	Inkscape
fig-2.0 png	Portable Network Graphics Image	Today, 3:06PM	39.2 KB	Preview
fig-2.0 svg	SVG	Today, 3:06PM	14.3 KB	Inkscape
fig-2.0a	TIFF Document	Today, 2:53PM	880.1 KB	Preview

Name: fig-2.0a
fig-2.0a tiff
Kind: TIFF Document
UTI: public.tiff
URL: cifs://CALVENT;PBG@BLACKS_N/ZPBG
Size: 880.1 KB (901,236 bytes) data: 901,236 bytes Physical: 881 KB (902,144 bytes)
Modified: Today, 2:53PM
Attributes: Today, 2:53PM
Owner: pbg (301)
Group: admin (80)
Permission: -rwx------ (700)
Path: /Volumes/ZPBG-1/tmp/book/os8-dir/fig-dir/fig-2.0a tiff
Application: Preview
Volume: ZPBG
Capacity: 735.4 GB
Free: 734.3 GB
Format: SMB
Mount Point: /Volumes/ZPBG-1

Apple Computer Inc.

main 1-800-MY-APPLE

other 800-275-2273

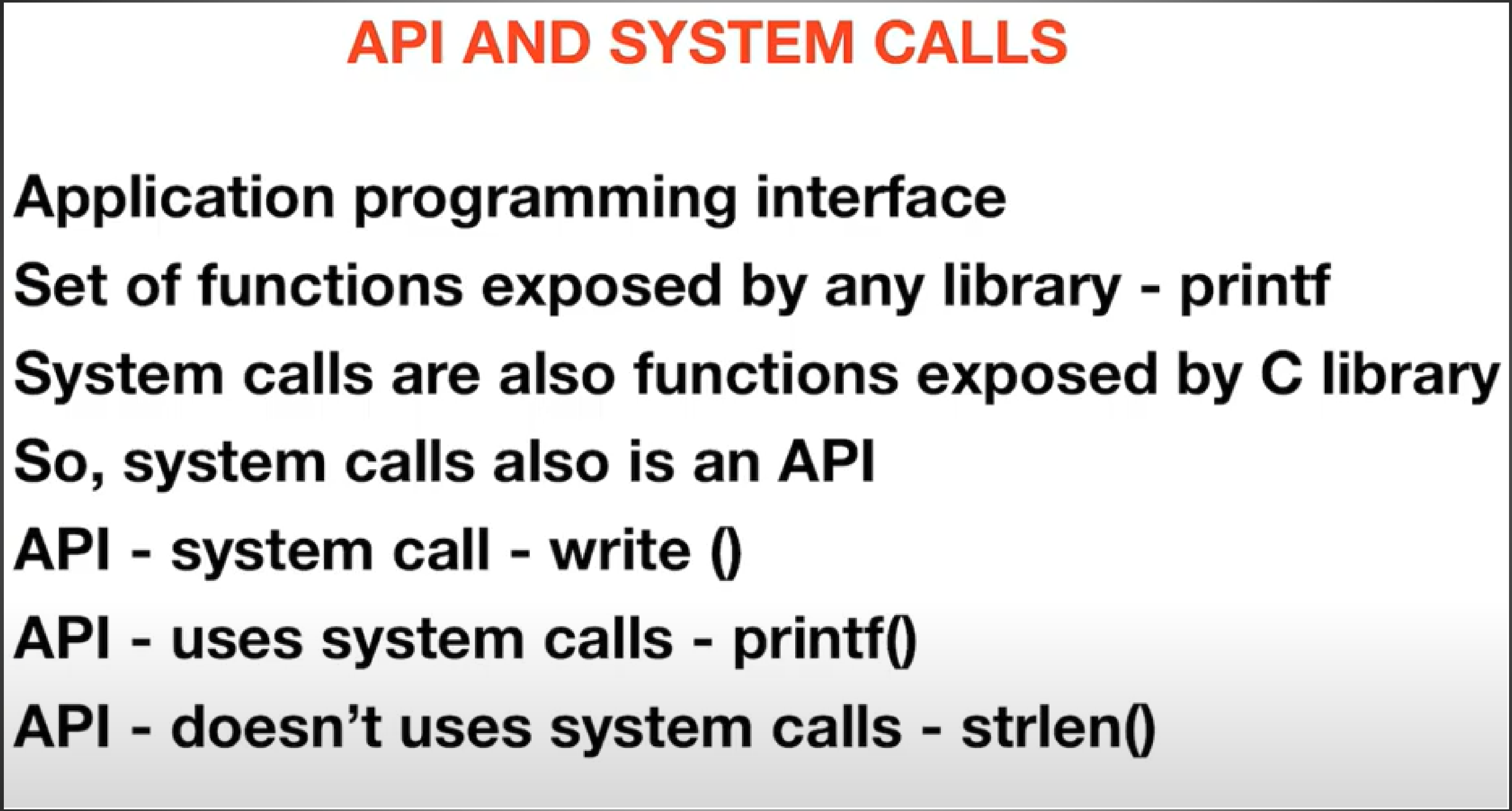
home page http://www.apple.com

work 1 Infinite Loop
Cupertino CA 95014
United States

op•er•at•ing sys•tem
noun
the software that supports a computer's basic functions, such as scheduling tasks, executing applications, and controlling peripherals.

System Calls

- A **system call** is a mechanism that provides the interface between a process and the operating **system.**
- It is a programmatic method in which a computer program requests a service from the kernel of the OS.
- **System calls** are the only entry points for the kernel **system.**
- 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)



API AND SYSTEM CALLS

Application programming interface

Set of functions exposed by any library - printf

System calls are also functions exposed by C library

So, system calls also is an API

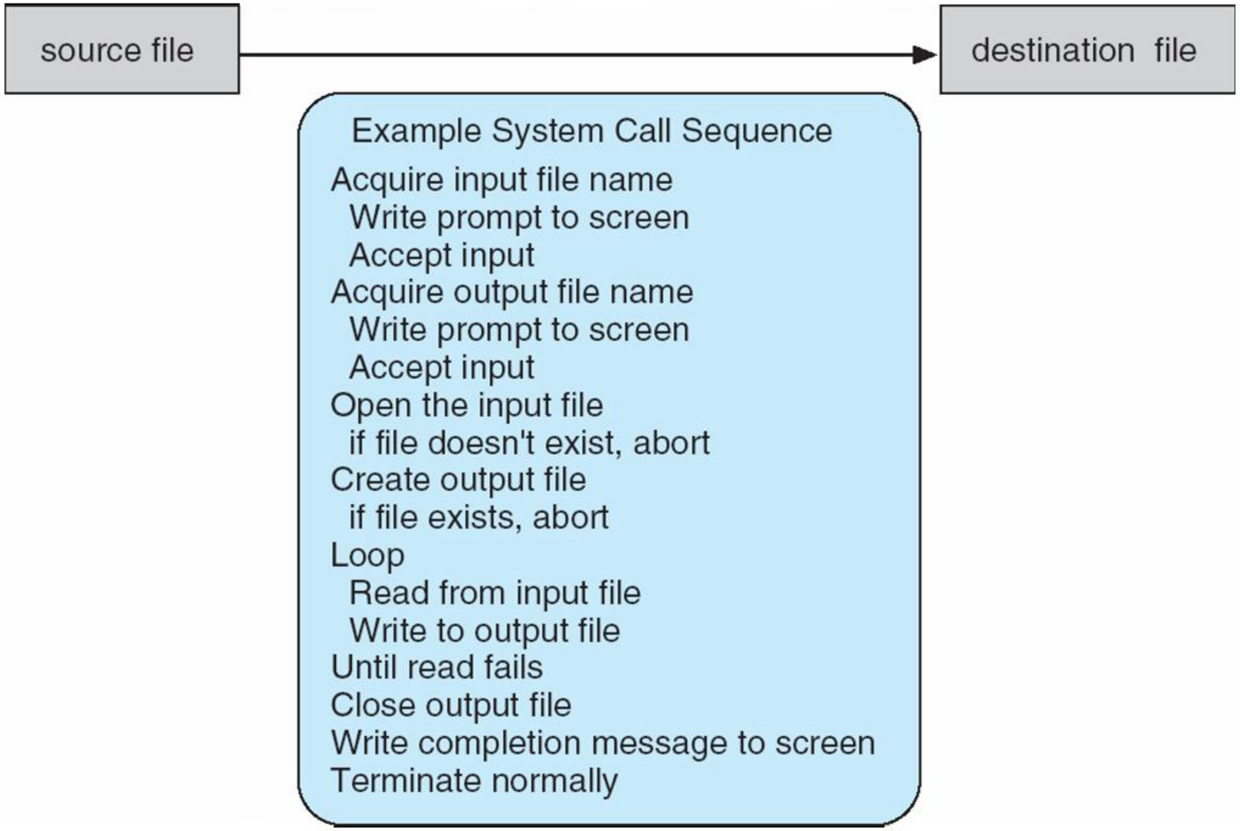
API - system call - write ()

API - uses system calls - printf()

API - doesn't uses system calls - strlen()

Example of System Calls

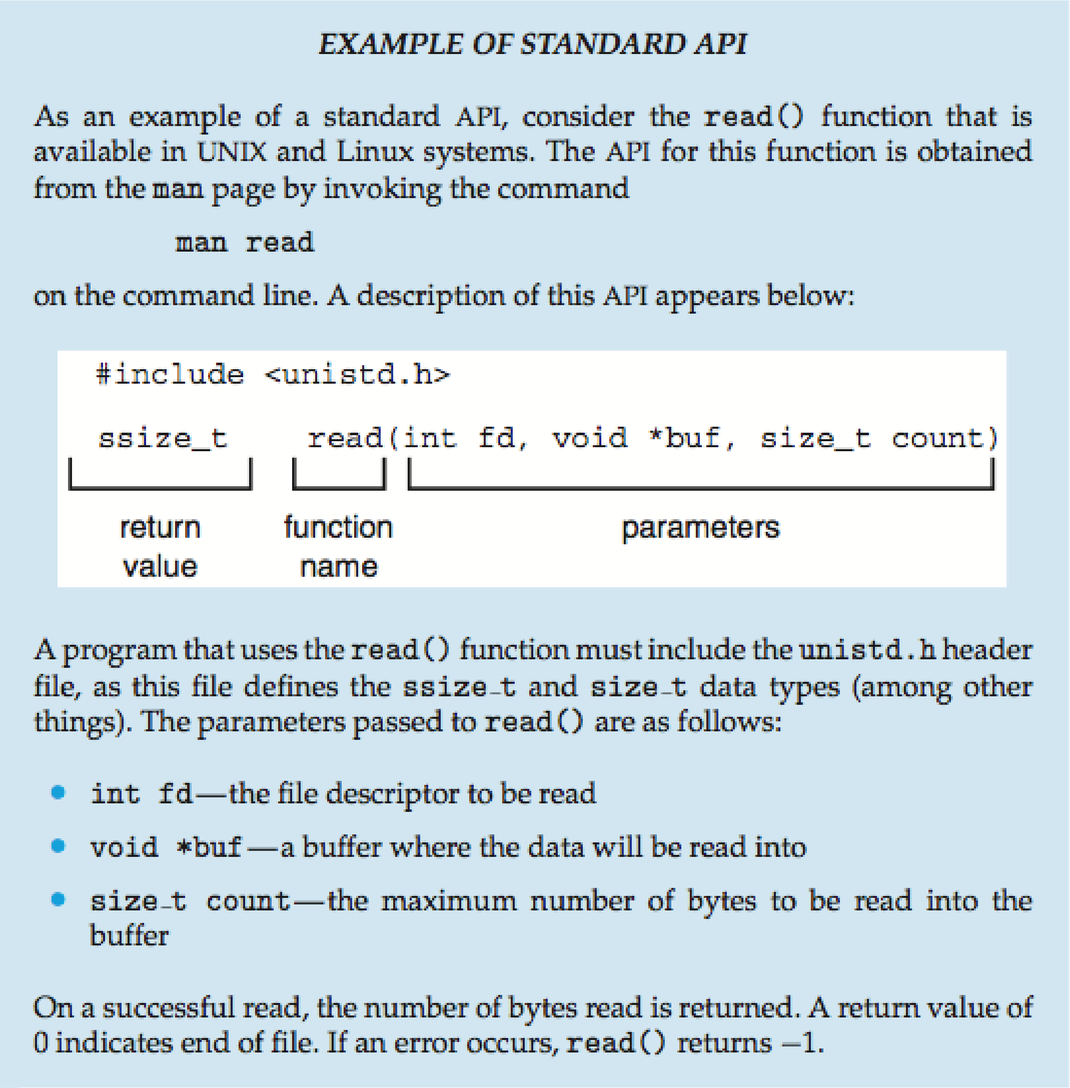
- System call sequence to copy the contents of one file to another file



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:

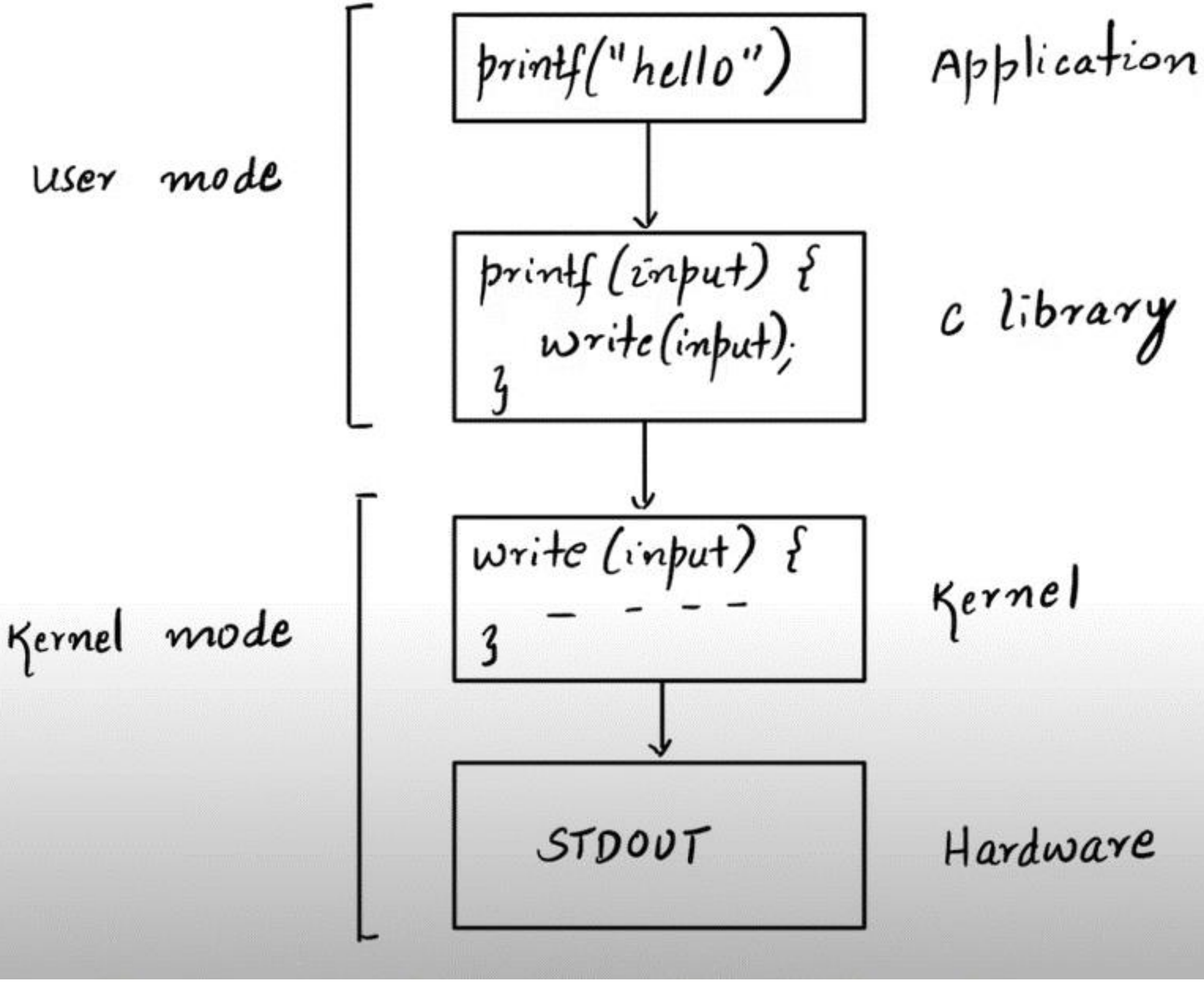
```
#include <unistd.h>

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

return 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:

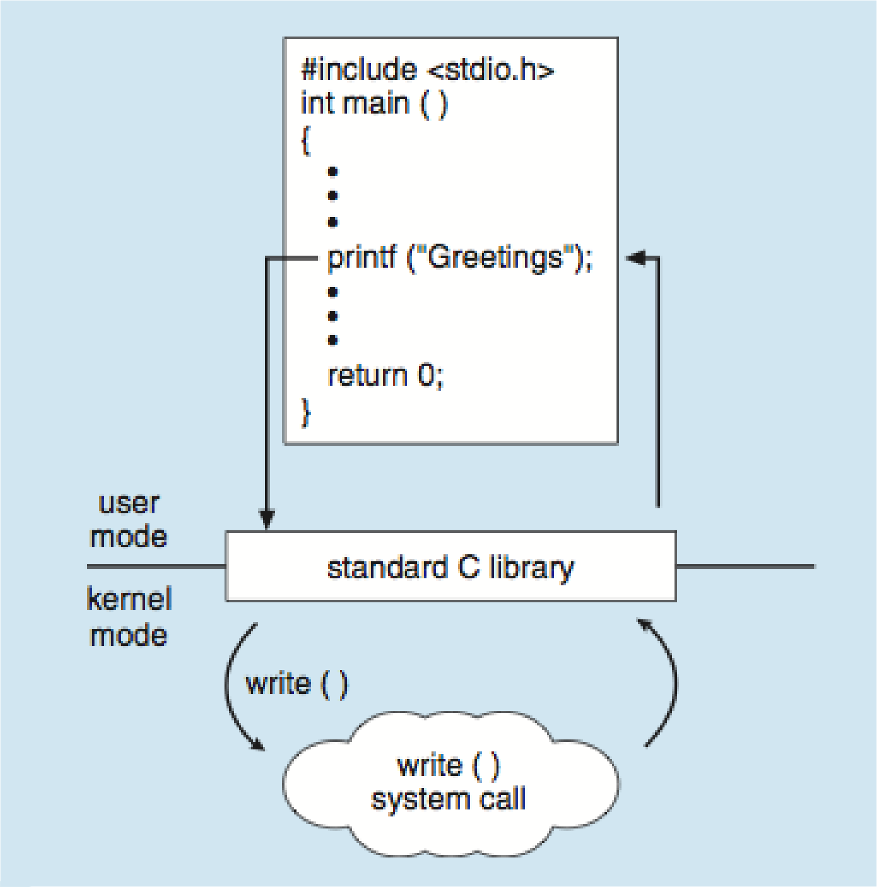
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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.


User mode
Kernel mode
printf("hello")
Application
printf (input) {
write(input);
}
C library
write (input) {
- - - -
}
Kernel
STDOUT
Hardware

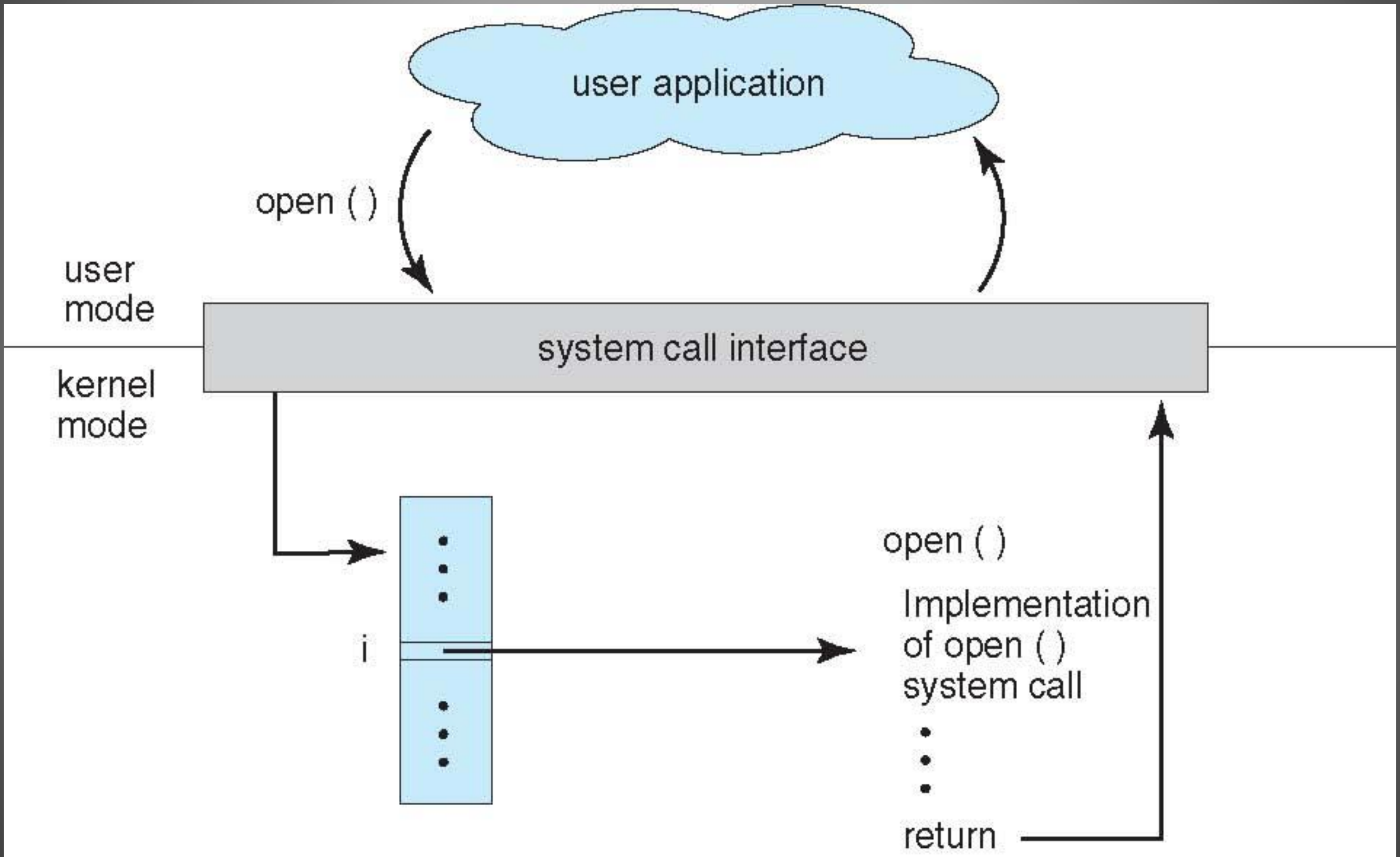
Standard C Library Example

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



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

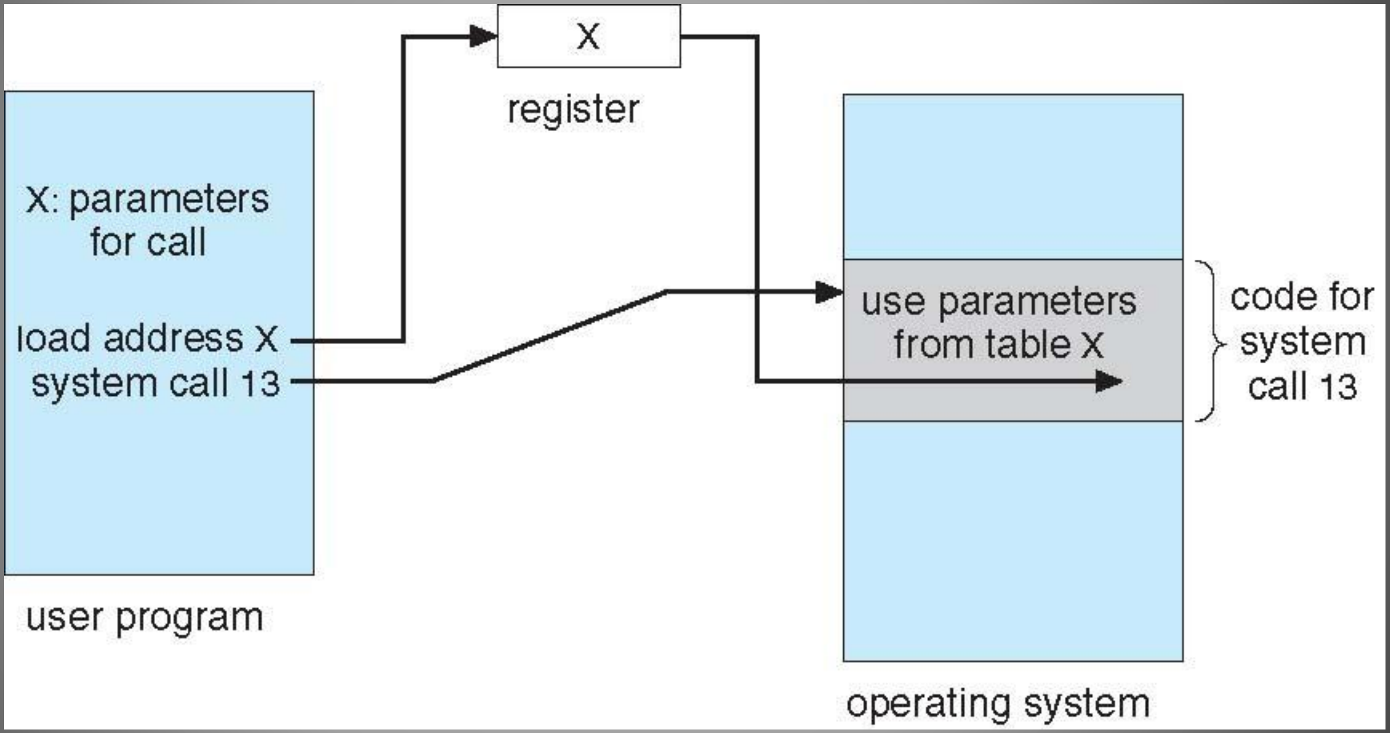
API – System Call – OS Relationship



System Call Parameter Passing

- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system

System Call Parameter Passing via Table



Types of System Calls

System calls can be roughly grouped into five major categories:

1. Process Control
2. File management
3. Device Management
4. Information Maintenance
5. Communication
6. protection

Types of System Calls

1.Process Control

- load
- execute
- create process (for example, fork on Unix-like systems or NtCreateProcess in the Windows NT Native API)
- terminate process
- get/set process attributes
- wait for time, wait event, signal event
- allocate free memory

Types of System Calls

2.File management

- create file, delete file
- open, close
- read, write, reposition
- get/set file attributes

3.Device Management

- request device, release device
- read, write, reposition
- get/set device attributes
- logically attach or detach devices

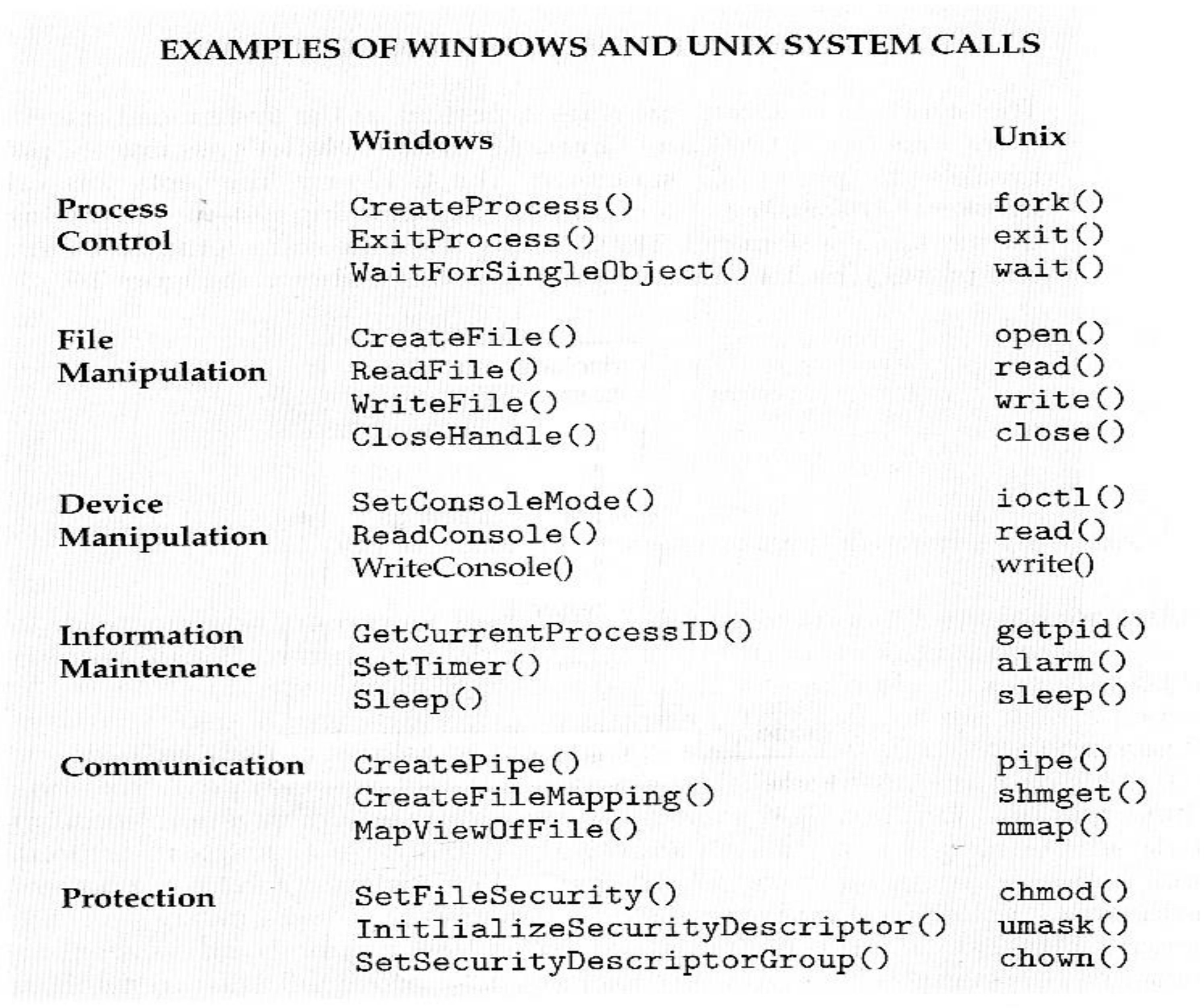
Types of System Calls

4.Information Maintenance

- get/set time or date
- get/set system data
- get/set process, file, or device attributes

5.Communication

- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote device



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()

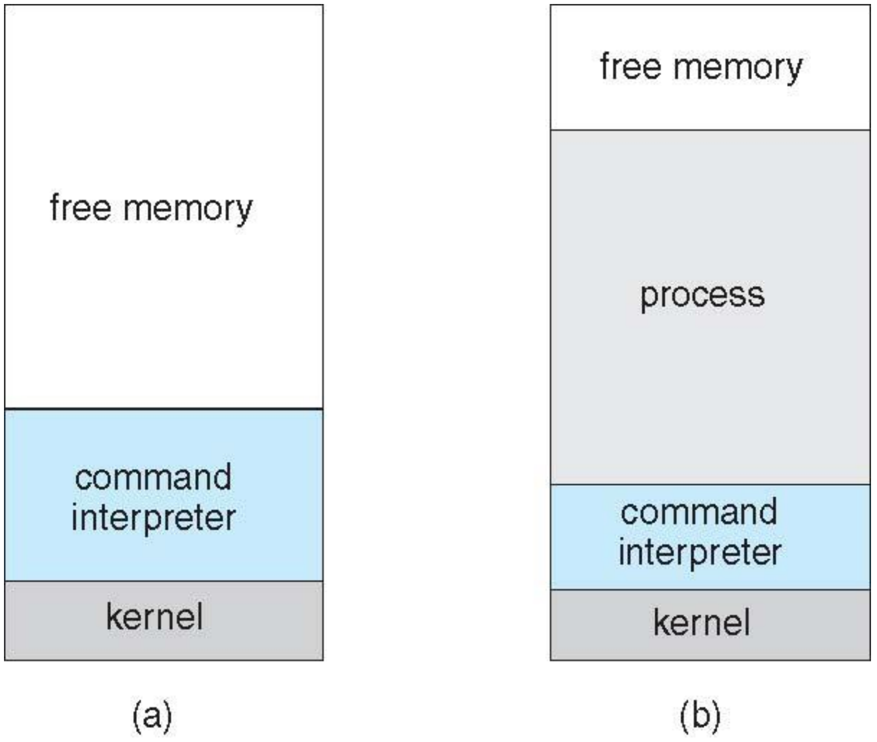
Information Maintenance	GetCurrentProcessID()
	SetTimer()
	Sleep()
		getpid()
		alarm()
		sleep()

Communication	CreatePipe()
	CreateFileMapping()
	MapViewOfFile()
		pipe()
		shmget()
		mmap()

Protection	SetFileSecurity()
	InitlializeSecurityDescriptor()
	SetSecurityDescriptorGroup()
		chmod()
		umask()
		chown()

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

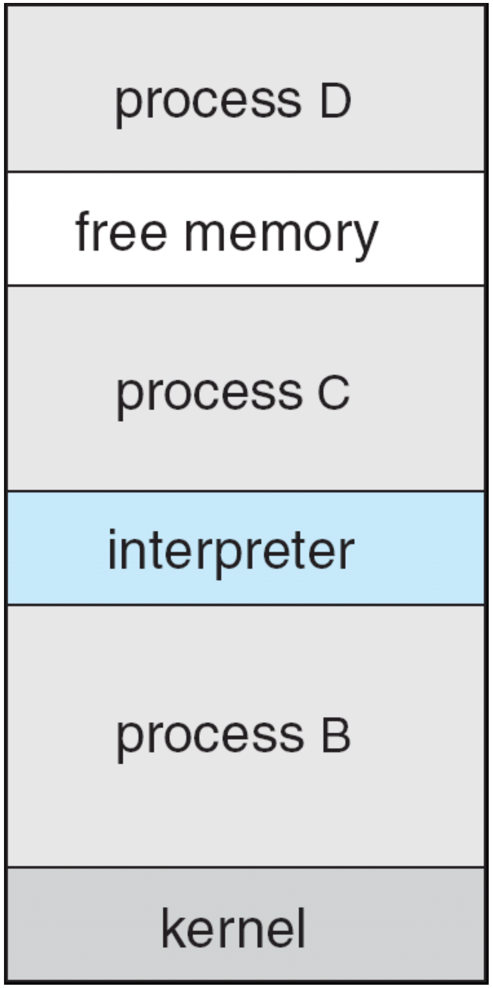


At system startup

running a program

Example: FreeBSD (Berkeley Software Distribution)

- 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:
  1. File manipulation
  2. Status information
  3. File modification
  4. Programming language support
  5. Program loading and execution
  6. Communications
  7. Background services
  8. Application programs

System Programs

- **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

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**– debugging systems for higher-level and machine language

- **Communications** – Provide the mechanism for creating virtual connections among processes, users, and computer systems

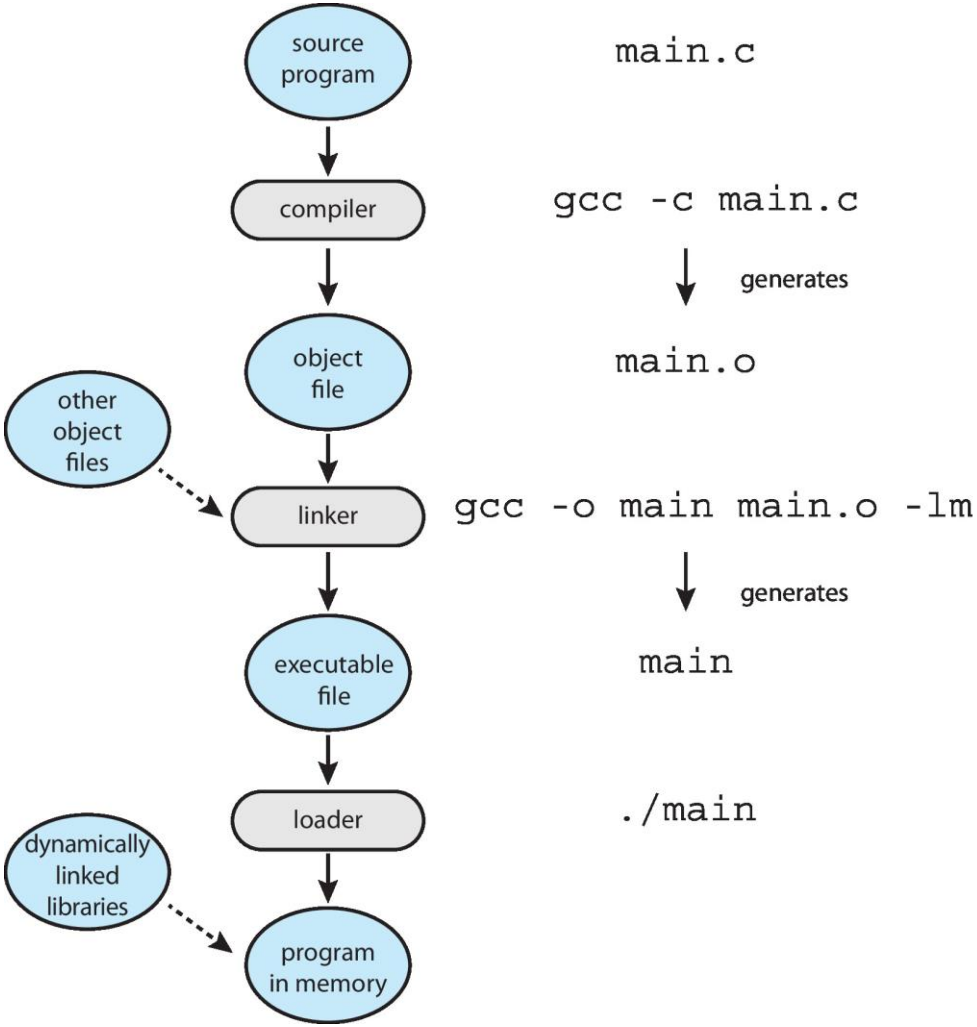
System Programs (Cont.)

- **Background Services**
  - Launch at boot time
  - 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

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
- 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

The Role of the Linker and Loader



source program

compiler

object file

other object files

linker

executable file

loader

dynamically linked libraries

program in memory

```
main.c
gcc -c main.c
```

generates

```
main.o
gcc -o main main.o -lm
```

generates

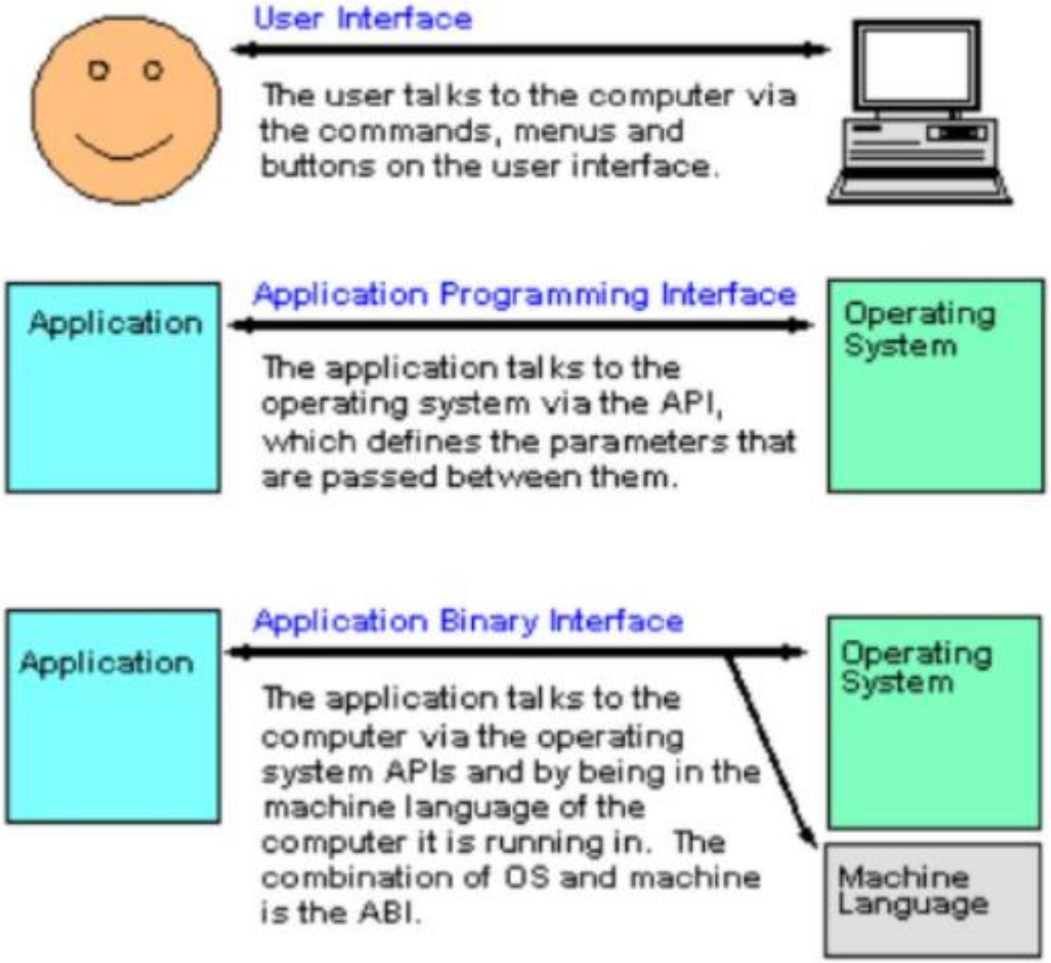
```
main
./main
```

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 be multi-operating system
  - Written in interpreted language like Python, Ruby, and interpreter available on multiple operating systems
  - App written in language that includes a VM containing the running app (like Java)
  - Use standard language (like C), compile separately on each operating system to run on each

- **Application Binary Interface** (**ABI**) is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc

GUI vs API vs ABI



The user talks to the computer via the commands, menus and buttons on the user interface.

The application talks to the operating system via the API, which defines the parameters that are passed between them.

The application talks to the computer via the operating system APIs and by being in the machine language of the computer it is running in. The combination of OS and machine is the ABI.

Operating System Design and Implementation

- 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 OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware
  - But slower
- **Emulation** can allow an OS to run on non-native hardware

Operating System Design and Implementation (Cont.)

- Important principle to separate

  Policy: *What* will be done?
  Mechanism: *How* to do it?

- Mechanisms determine how to do something; policies decide what will be done

- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)

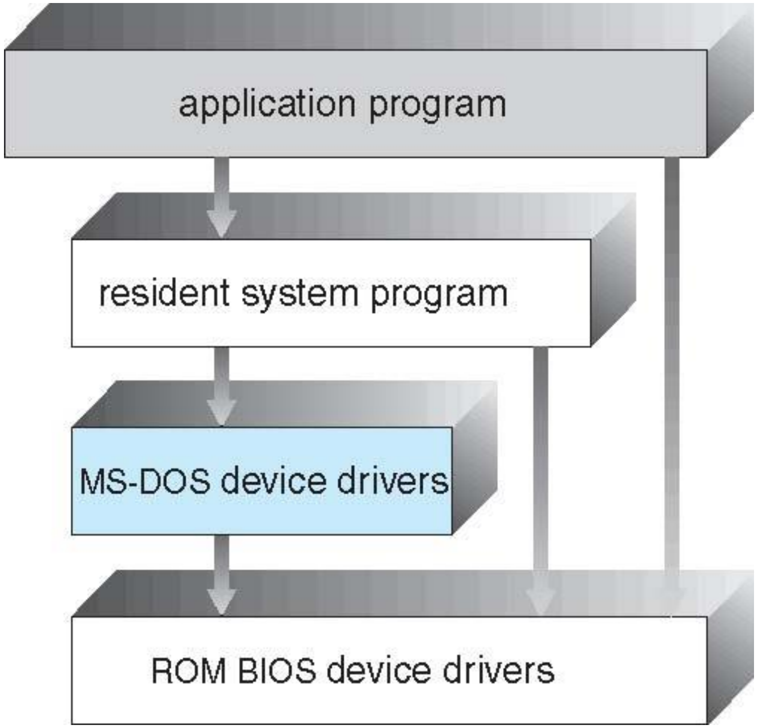
Operating System Structure

There are six different structures:

1. Simple Structure
2. Monolithic Structures
3. Layered Systems
4. Micro kernels
5. Modules
6. Hybrid Machines

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



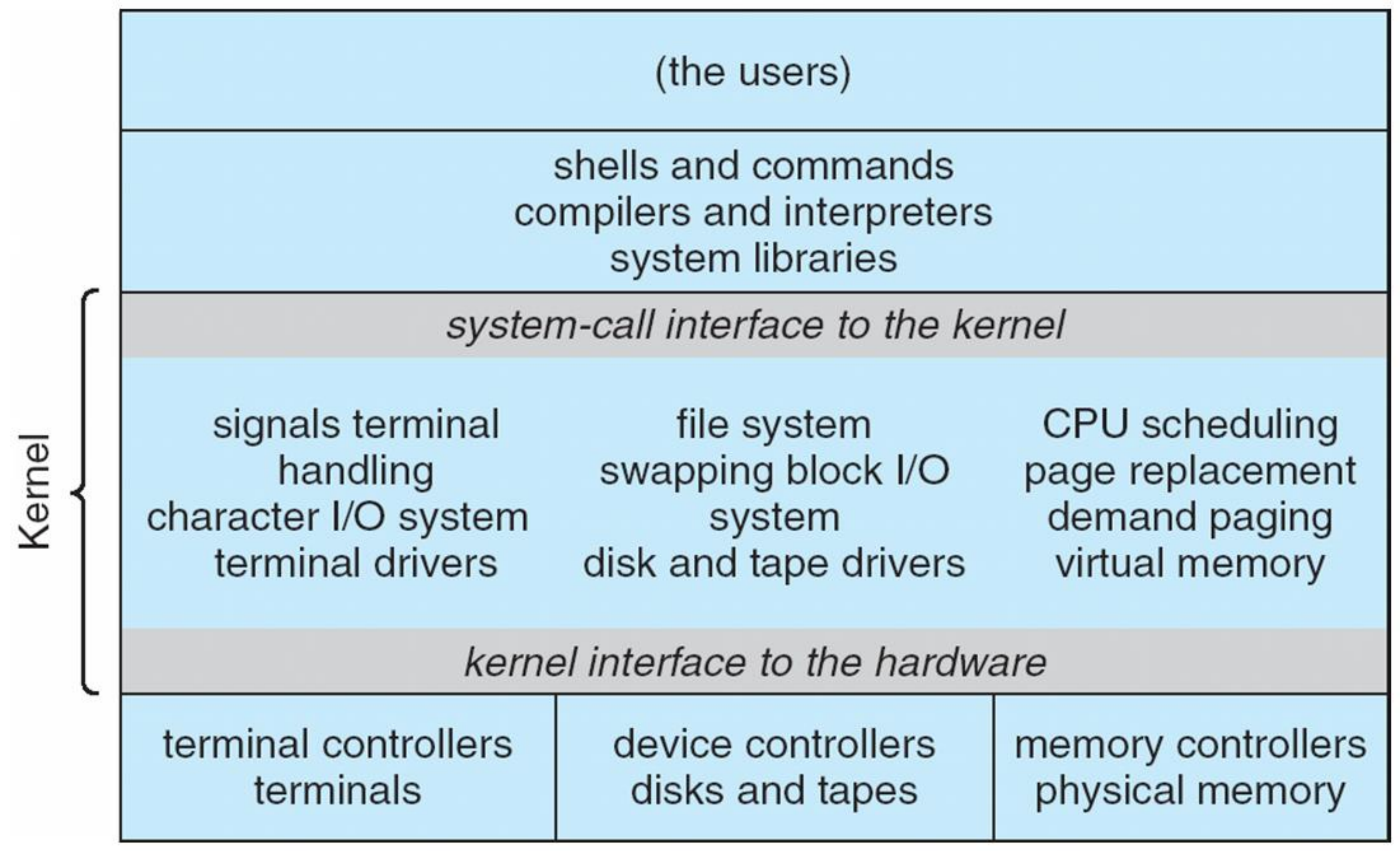
Monolithic Structure -- UNIX

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

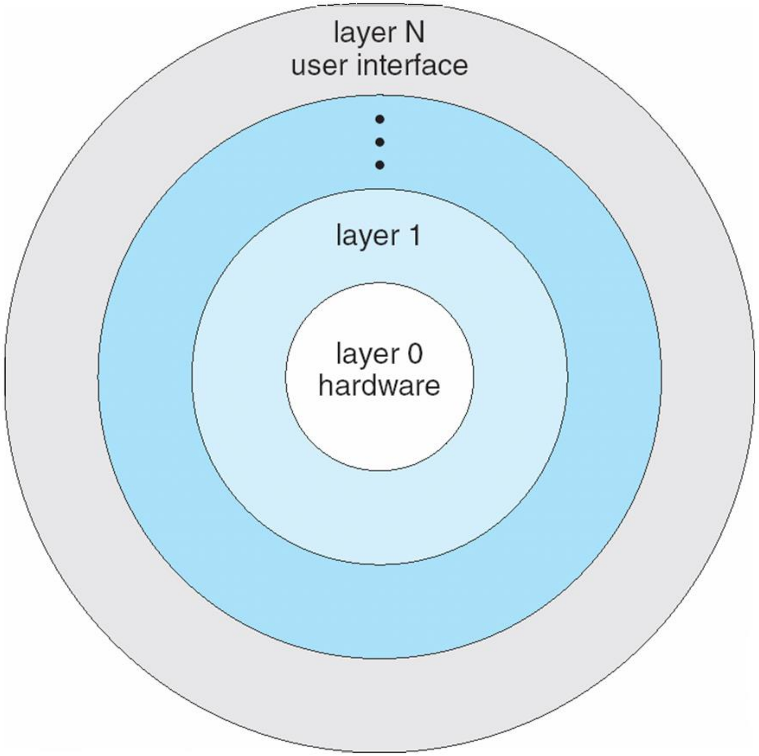
device controllers
disks and tapes

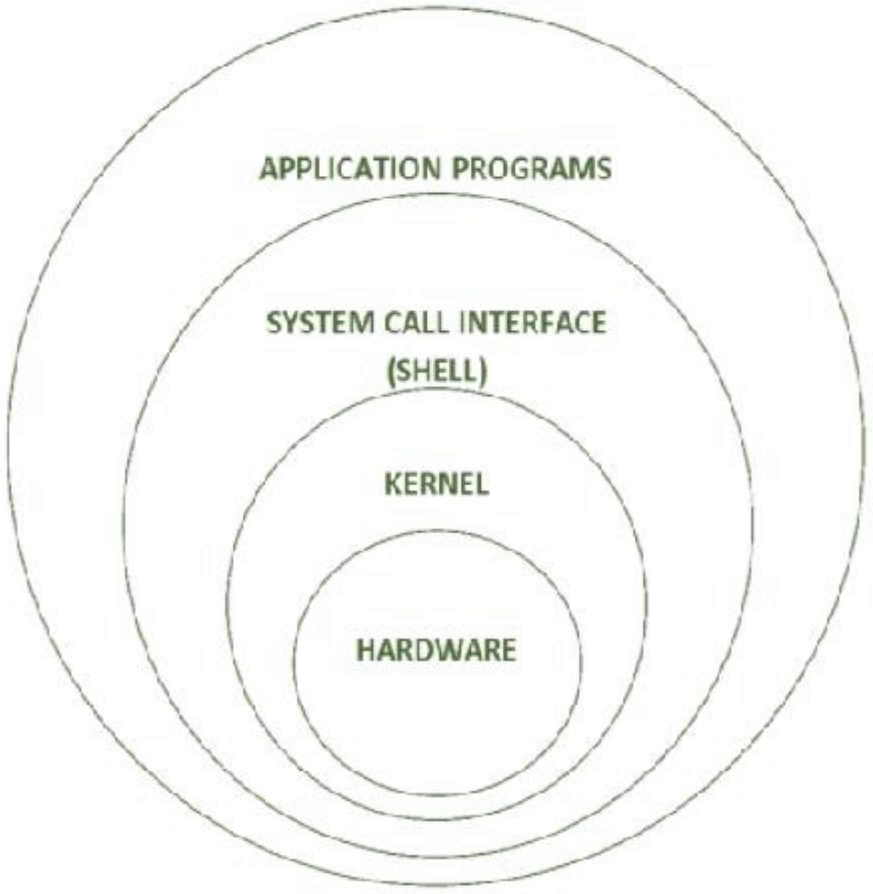
memory controllers
physical memory

Kernel

Layered Approach

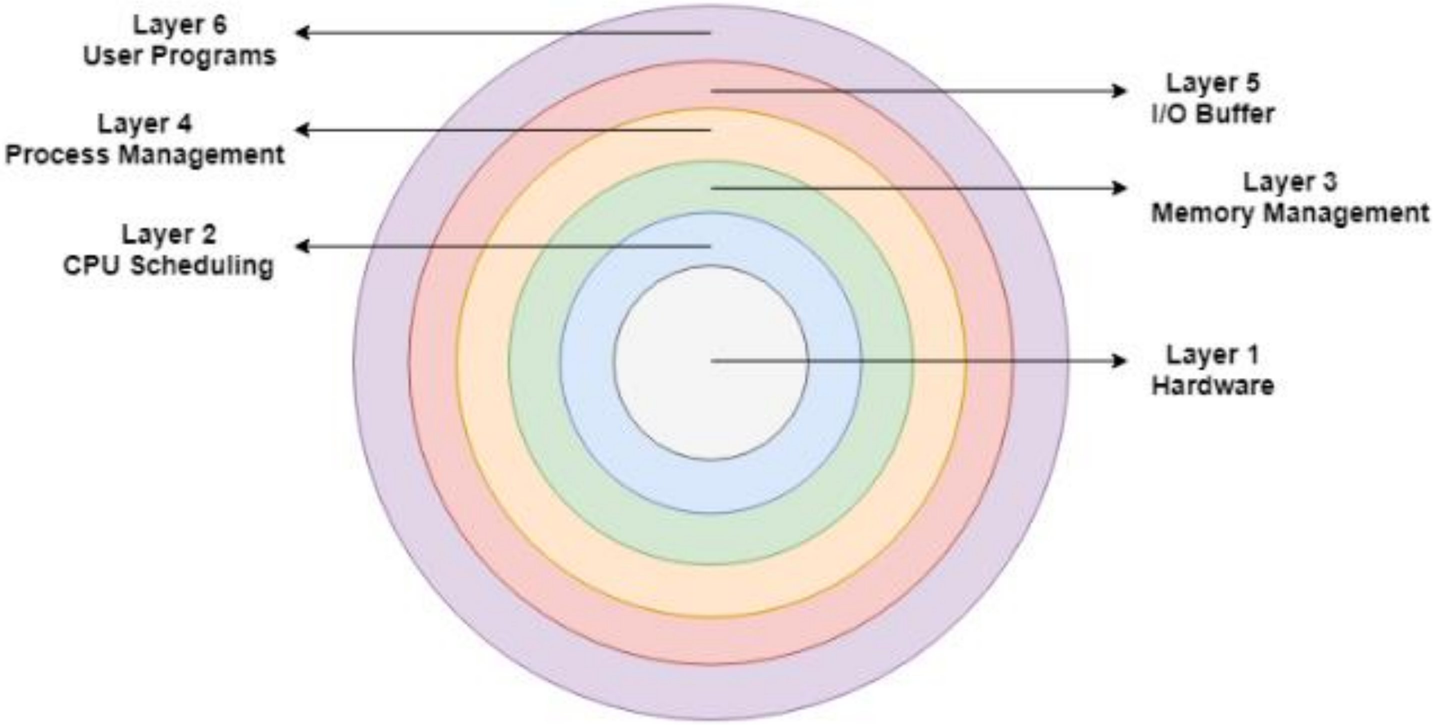
- The operating system is divided into several layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level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





UNIX Architecture

Layered Approach

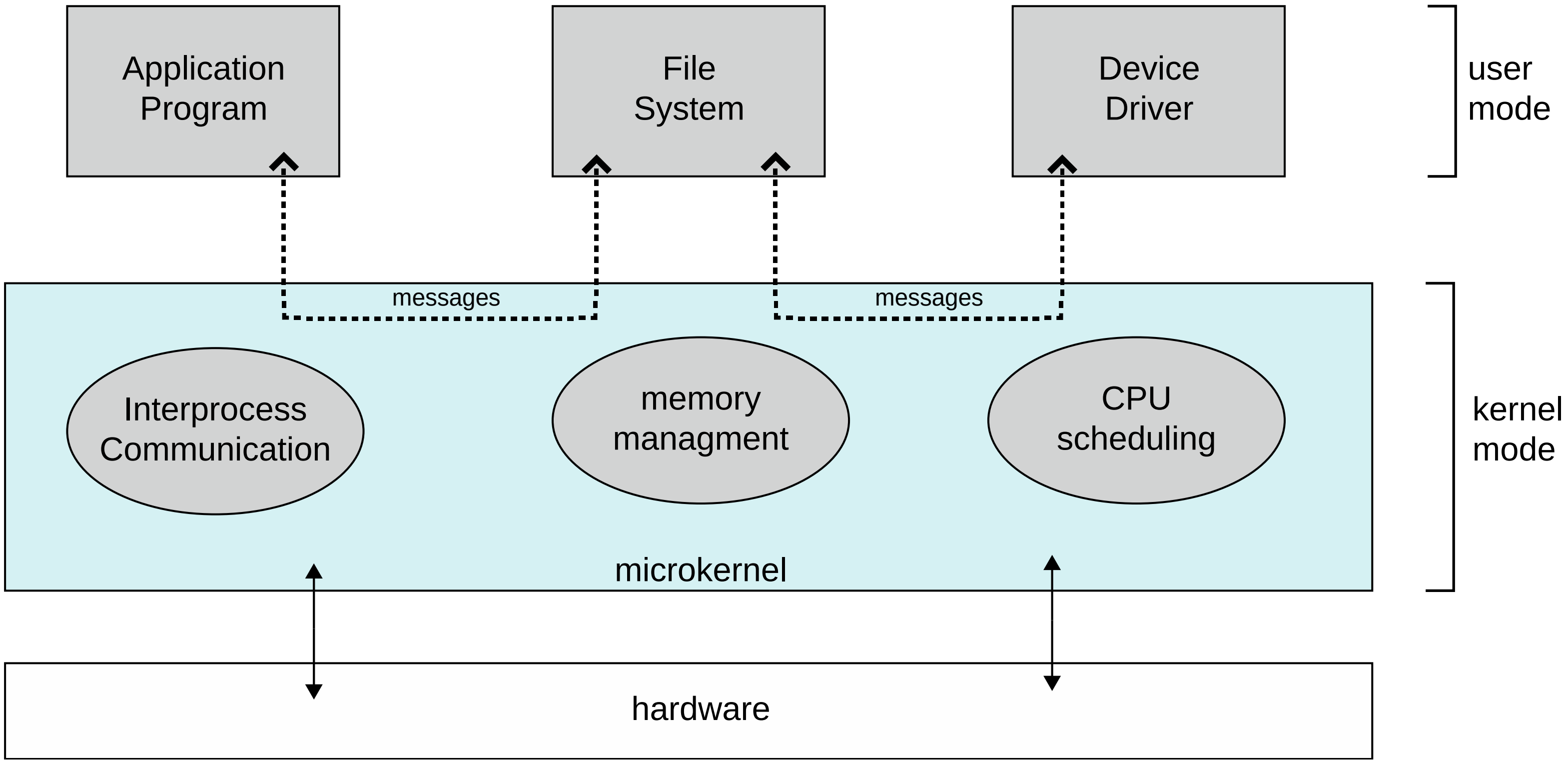


LAYERED OPERATING SYSTEM

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

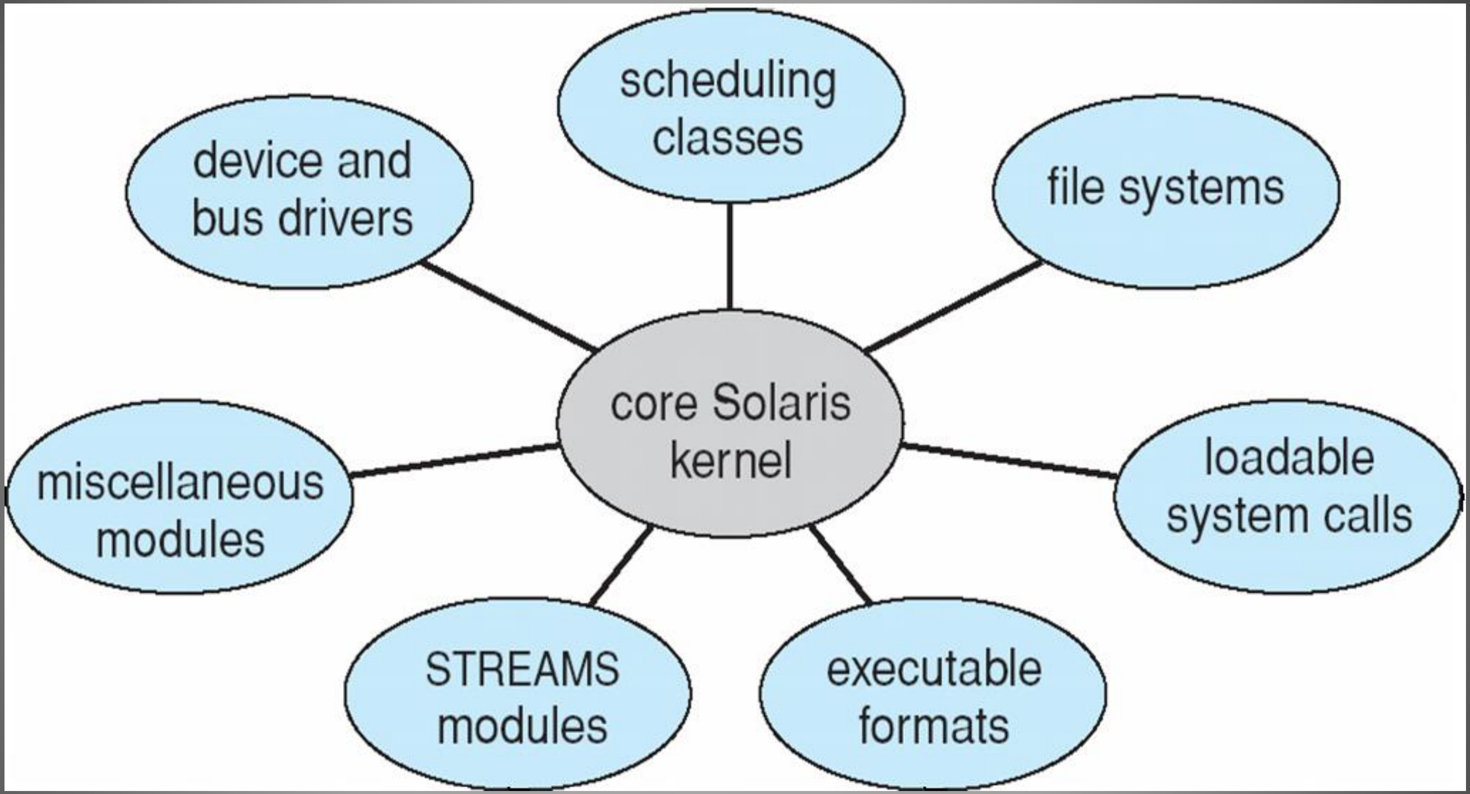
Microkernel System Structure



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, like layers but with more flexible
  - Linux, Solaris, etc

Solaris Modular Approach

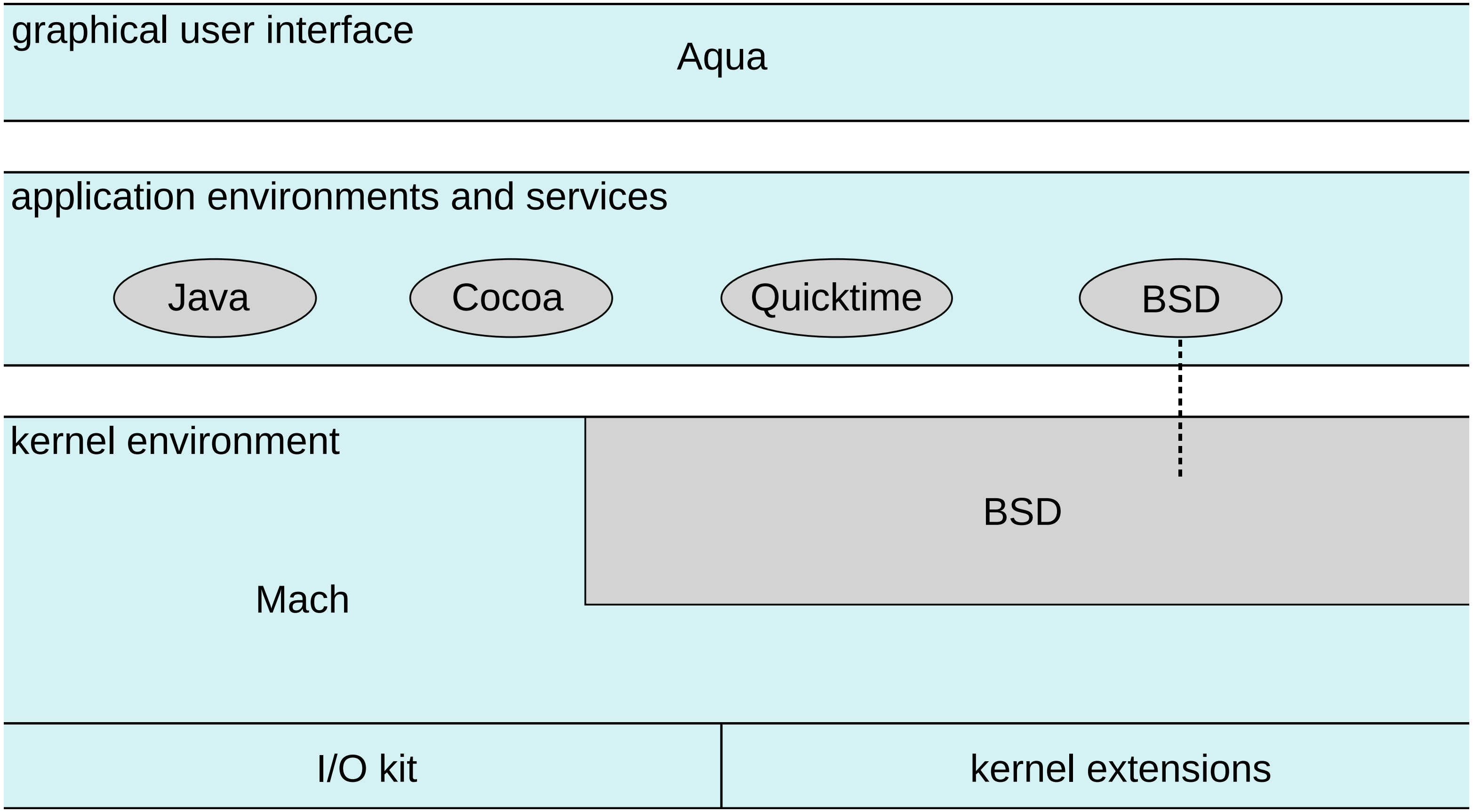


Hybrid Systems

- 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)

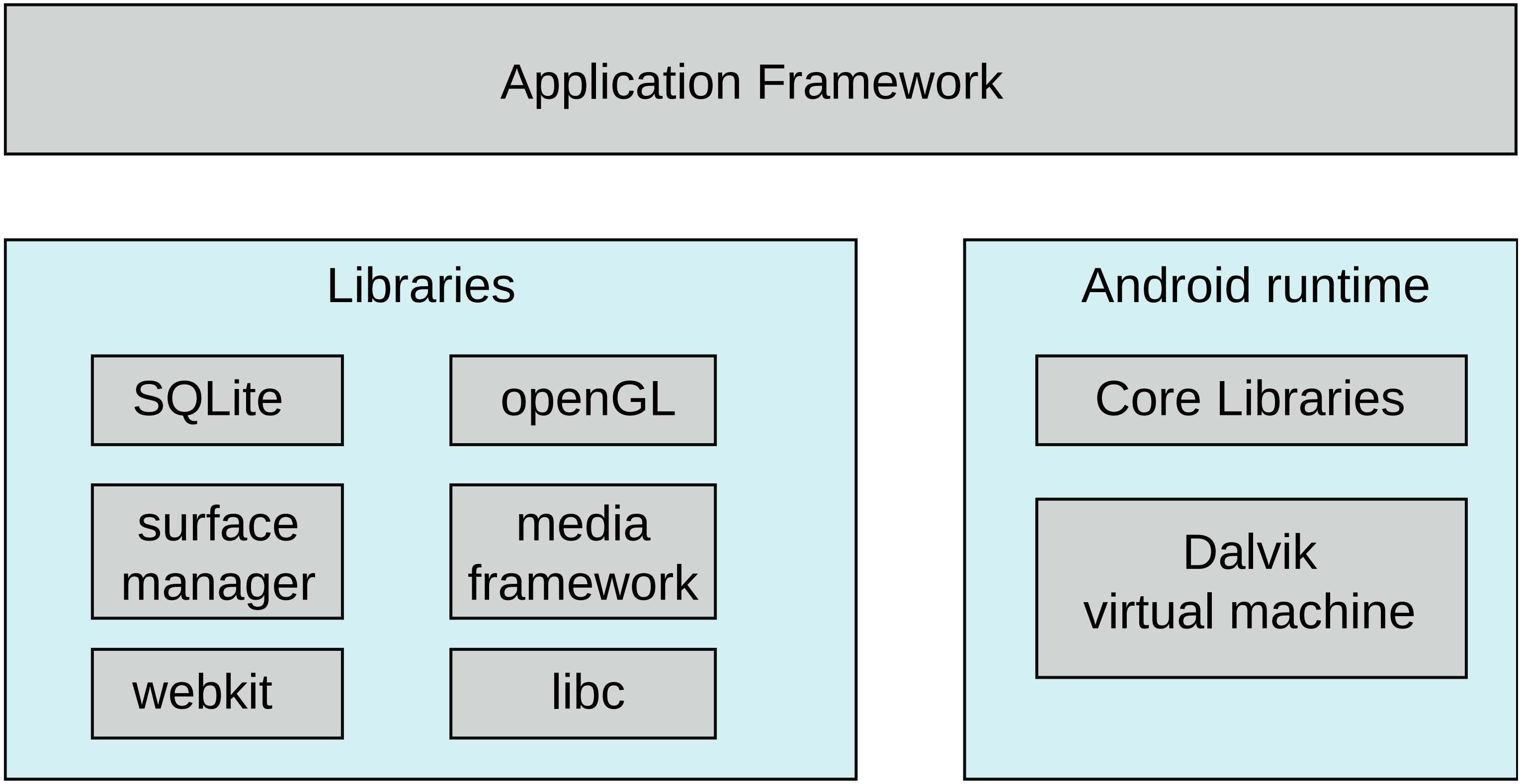
Mac OS X Structure



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

Android Architecture



Application Framework

Libraries

SQLite

openGL

surface manager

media framework

webkit

libc

Android runtime

Core Libraries

Dalvik virtual machine

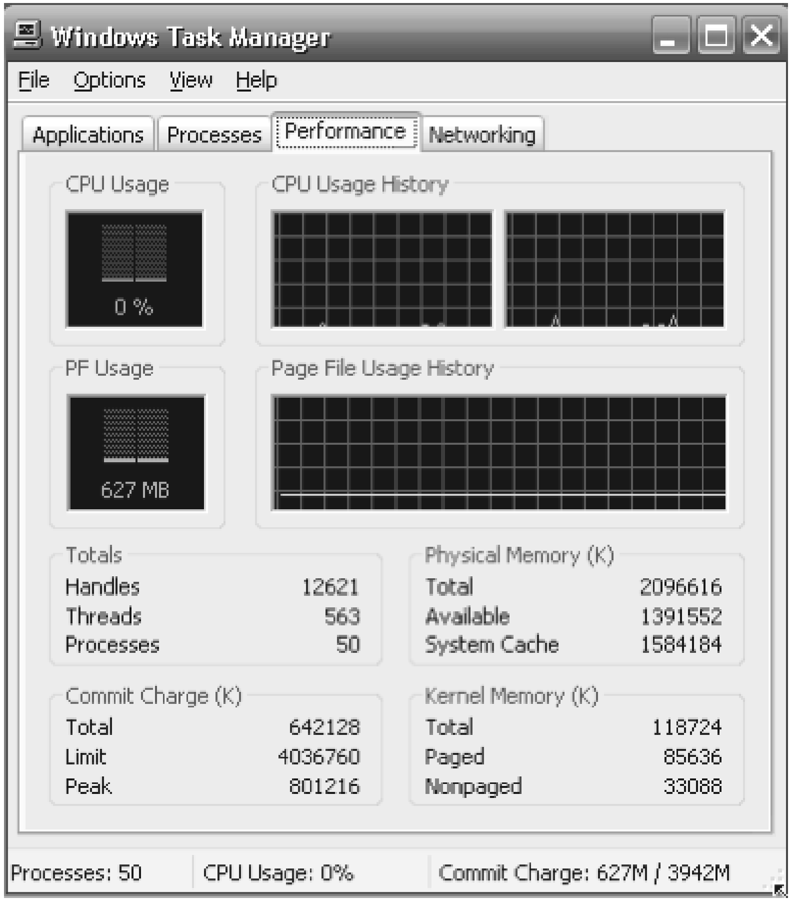
Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- OS generate **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
  - Sometimes using ***trace listings*** of activities, recorded for analysis
  - **Profiling** is periodic sampling of instruction pointer to look for statistical trends

Kernighan’ s Law: “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

Performance Tuning

- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- For example, “top” program or Windows Task Manager



CPU Usage: 0 %

PF Usage: 627 MB

Totals	
Handles	12621
Threads	563
Processes	50

Physical Memory (K)	
Total	2096616
Available	1391552
System Cache	1584184

Commit Charge (K)	
Total	642128
Limit	4036760
Peak	801216

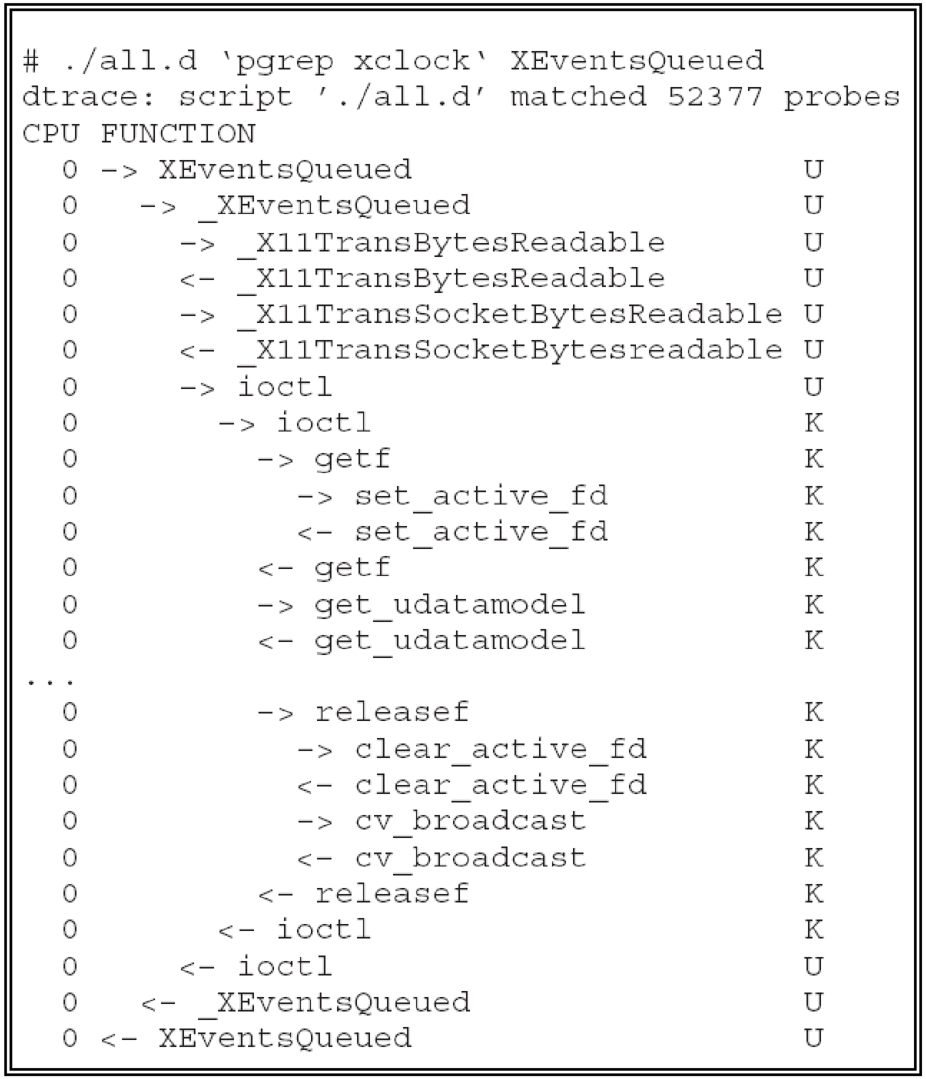
Kernel Memory (K)	
Total	118724
Paged	85636
Nonpaged	33088

Processes: 50 CPU Usage: 0% Commit Charge: 627M / 3942M

DTrace

- DTrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems.

- DTrace code to record amount of time each process with UserID 101 is in running mode (on CPU) in nanoseconds



```
# ./all.d ‘pgrep xclock‘ XEventsQueued
dtrace: script ’./all.d’ matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                    U
  0   -> _XEventsQueued                 U
  0     -> _X11TransBytesReadable       U
  0     <- _X11TransBytesReadable       U
  0     -> _X11TransSocketBytesReadable U
  0     <- _X11TransSocketBytesreadable U
  0     -> ioctl                        U
  0       -> ioctl                      K
  0         -> getf                     K
  0           -> set_active_fd          K
  0           <- set_active_fd          K
  0         <- getf                     K
  0         -> get_udatamodel           K
  0         <- get_udatamodel           K
...
  0         -> releasef                 K
  0           -> clear_active_fd        K
  0           <- clear_active_fd        K
  0           -> cv_broadcast           K
  0           <- cv_broadcast           K
  0         <- releasef                 K
  0       <- ioctl                      K
  0     <- ioctl                        U
  0   <- _XEventsQueued                 U
  0 <- XEventsQueued                    U
```

Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site

SYSGEN

- Information that is needed to configure an OS include:
- What CPU(s) are installed on the system, and what optional characteristics does each have?
- How much RAM is installed?
- What devices are present? The OS needs to determine which device drivers to include, as well as some device-specific characteristics and parameters.
- What OS options are desired, and what values to set for OS parameters.

System Boot

- When power initialized on system, execution starts at a fixed memory location
  - Firmware ROM used to hold initial boot code
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – bootstrap loader, stored in ROM or EEPROM locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where boot block at fixed location loaded by ROM code, which loads bootstrap loader from disk
- Common bootstrap loader, GRUB, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then running