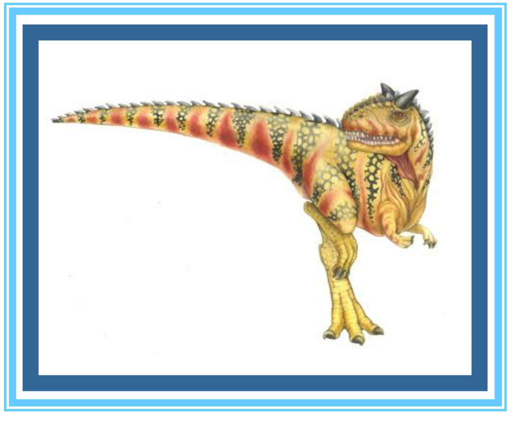
# Chapter 2: Operating-System Structures



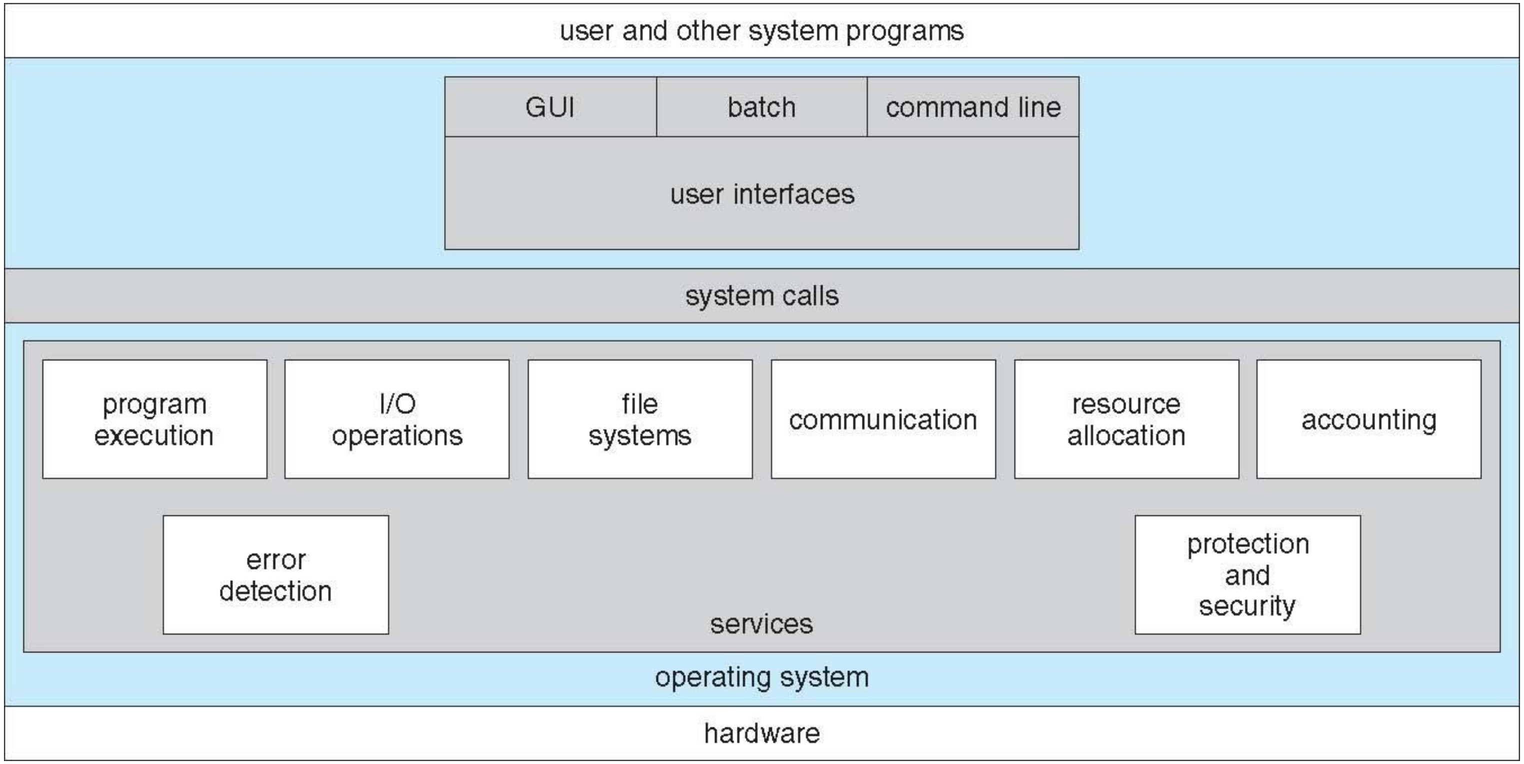
# Chapter 2:  Operating-System Structures

- 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 services an operating system provides to users, processes, and other systems
- To discuss various ways of structuring an operating system
- To explain how operating systems are installed and customized and how they boot

# A View of Operating System Services



# Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (UI).
    - Varies between **Command-Line Interpreter** (CLI), **Batch**, **Graphical User Interface** (GUI)
  - **Program execution** - The system must be able to load a program into main memory to run the program and to finish 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.)

- **File-system manipulation** – OS should support programs to manipulate files and directories as follows: read, write, create, delete, search, list, access permission.
- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be done via **shared memory** or through **message passing** (packets moved by the OS)
- **Error detection**
  - Errors may occur everywhere in a computer system
  - For each type of error, OS should take appropriate actions to ensure correct and consistent computing
  - Debugging 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** – How and when to allocate resources to multiple processes or users
    - 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 multi-user or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - Protection ensures 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, usually considered as an internal and external threat handler

# User Operating System Interface - CLI

- CLI or **command line interpreter** allows direct command input
  - Sometimes implemented in kernel, sometimes as a system program
  - User customizable interface – **shell**s
  - Primarily fetches a command from a user and executes it
  - Two types of commands
    - Built-in commands in OS
    - User program names to execute it

| echo "hello world" | userApplication –all -withoutWindow |
|---|---|
| **built-in command example** | **program name** |

# Bourne Shell Command Interpreter

Default

New  Info  Close  Execute  Bookmarks

Default  Default

```
PBG-Mac-Pro:~ pbg$ w
15:24  up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER     TTY      FROM              LOGIN@  IDLE WHAT
pbg      console  -                14:34      50 -
pbg      s000     -                15:05       - w
PBG-Mac-Pro:~ pbg$ iostat 5
          disk0           disk1           disk10        cpu      load average
    KB/t tps  MB/s     KB/t tps  MB/s     KB/t tps  MB/s  us sy id   1m   5m   15m
   33.75 343 11.30    64.31  14  0.88    39.67   0  0.02  11  5 84  1.51 1.53 1.65
    5.27 320  1.65     0.00   0  0.00     0.00   0  0.00   4  2 94  1.39 1.51 1.65
    4.28 329  1.37     0.00   0  0.00     0.00   0  0.00   5  3 92  1.44 1.51 1.65
^C
PBG-Mac-Pro:~ pbg$ ls
Applications                    Music                           WebEx
Applications (Parallels)        Pando Packages                  config.log
Desktop                         Pictures                        getsmartdata.txt
Documents                       Public                          imp
Downloads                       Sites                           log
Dropbox                         Thumbs.db                       panda-dist
Library                         Virtual Machines                prob.txt
Movies                          Volumes                         scripts
PBG-Mac-Pro:~ pbg$ pwd
/Users/pbg
PBG-Mac-Pro:~ pbg$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBG-Mac-Pro:~ pbg$ 
```

# 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 or folder)
  - Invented at Xerox PARC

- 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)

# Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse is not possible to be used or not desired
  - Gestures are mapped to actions and selection of objects on the screen
  - Virtual keyboard for text entry

# System Calls

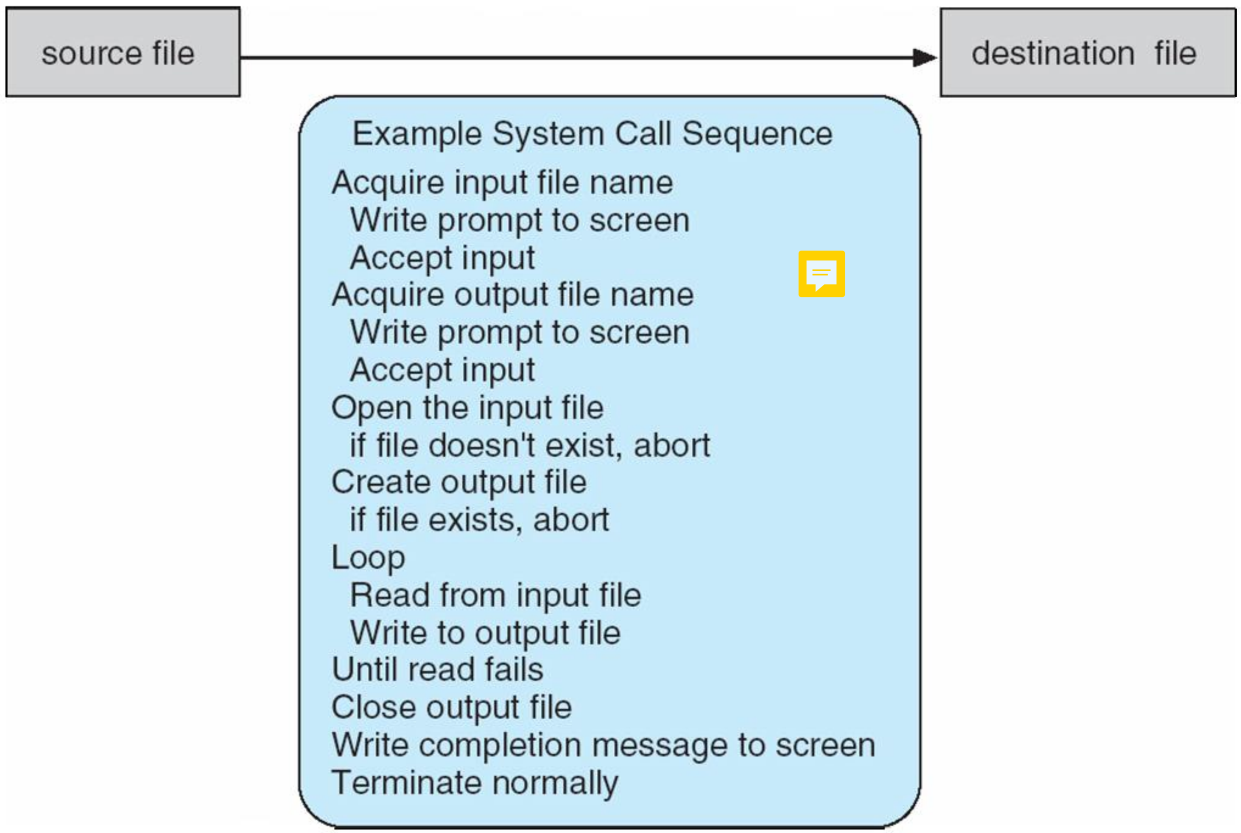
- Programming interface to the services **provided by the OS**
- Typically written in a high-level language (C or C++)

- Mostly accessed by programs via a high-level **Application Programming Interface** (API) rather than via a direct call
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

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

# Example of System Calls

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



# Example of Standard API

Read from input file

***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:

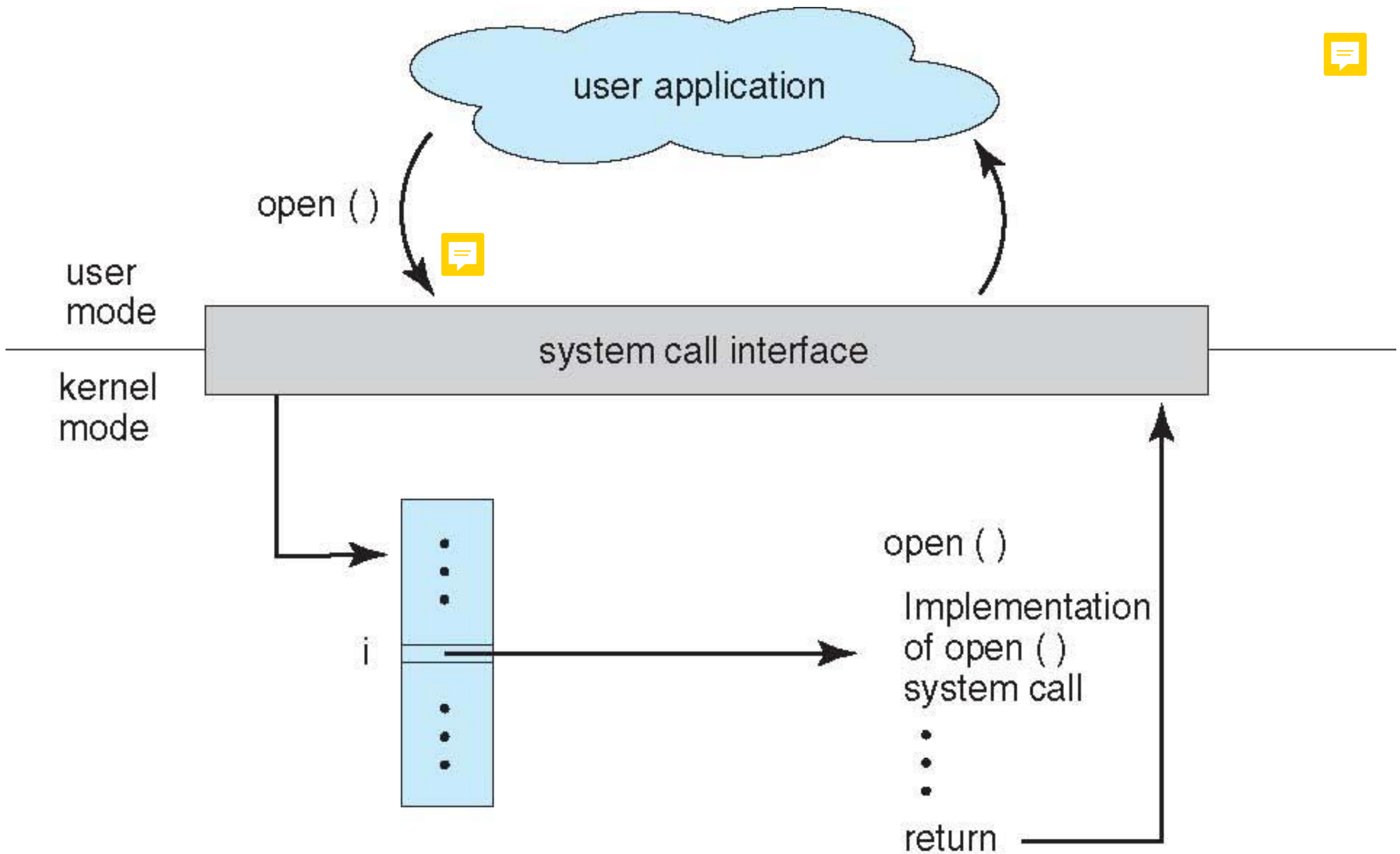
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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 is associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers (ex> int 21h)
- 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 needs nothing to know about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result
  - Most details of OS interface are hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with a compiler)

# API – System Call – OS Relationship



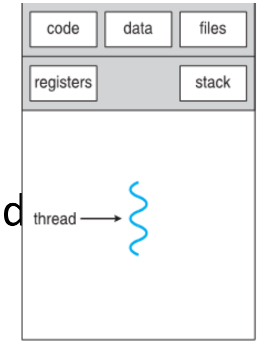
# System Call Parameter Passing

- Often, some system calls require additional information (parameters)
  - Exact type and amount of information vary according to OS and call

- Three general methods are used to pass parameters to the OS
  - Simplest: pass the parameters stored in registers

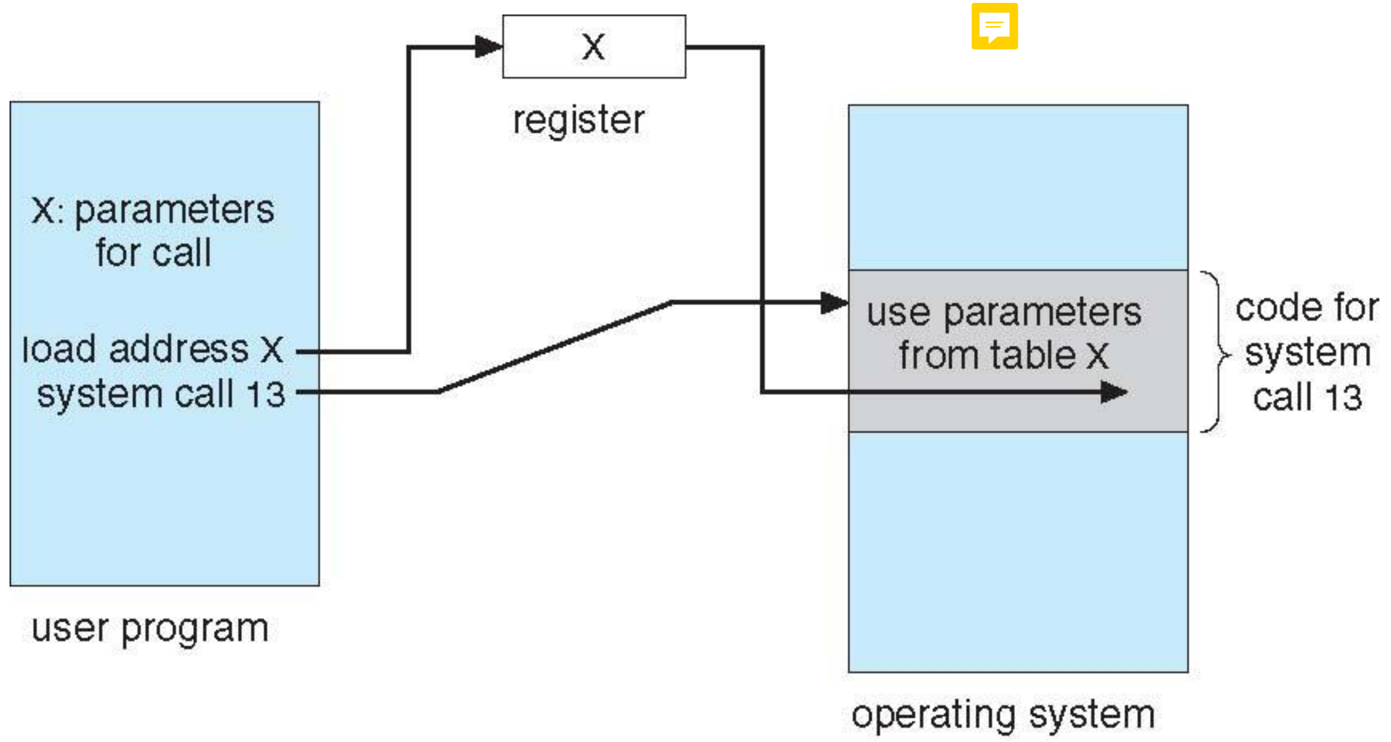
    - The number of registers are limited
  - Parameters are stored in a block or a table in memory, and address of the block is passed as a parameter in a register

    - This approach is taken by Linux and Solaris
  - Parameters are 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 (1)

- 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 (2)

- 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 (3)

- Information maintenance
  - Getting and setting
    - Time, date, system data, process, file, device attributes

- Communications
  - In the **message passing model**,
    - Get identifiers (ID) for host name and process name
    - Open/close, accept/wait(daemons) communication connection
    - Send(write)/receive(read) messages between client and server
  - In **shared-memory model**,
    - Create and gain access to memory regions
    - Transfer status information
    - Attach and detach remote devices

# Types of System Calls (4)

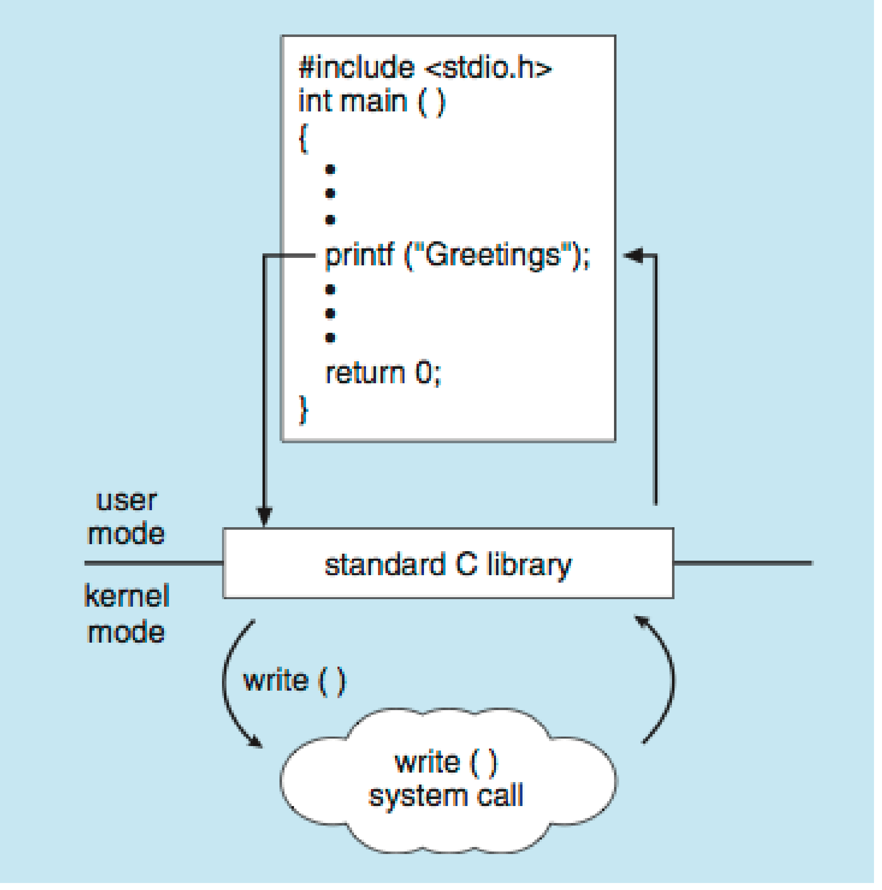
- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

# Examples of Windows and  Unix System Calls

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

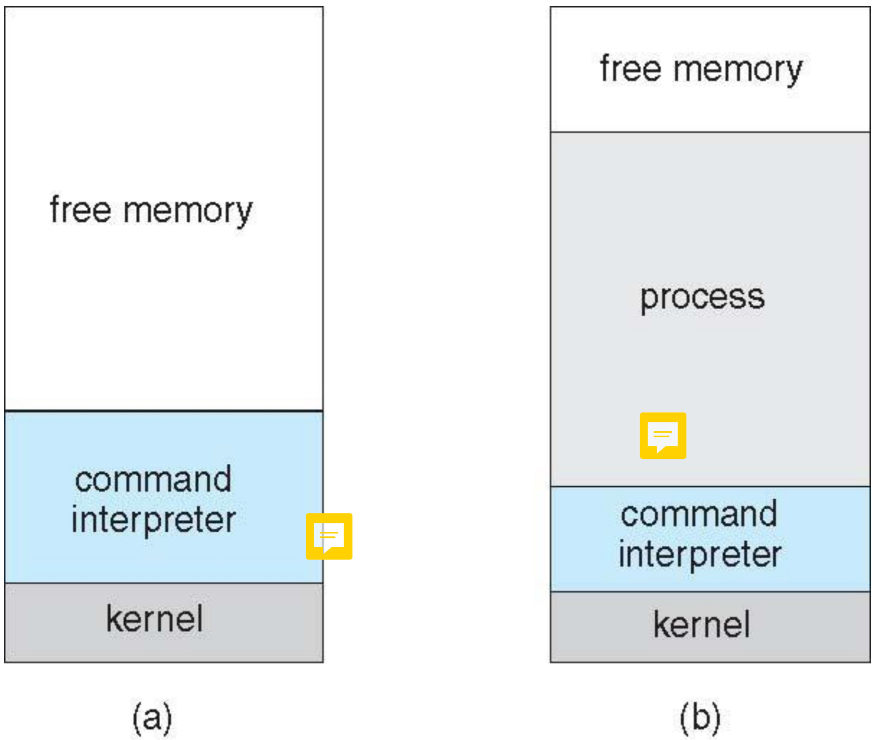
# Standard C Library Example

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



# Example: MS-DOS

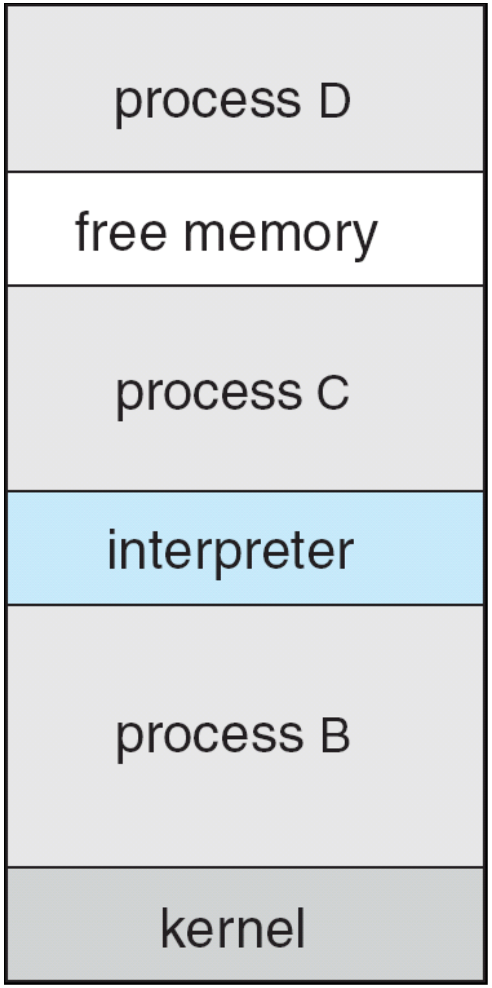
- Single-tasking
- Shell invoked when system booted
- Simple method to run program
  - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded




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



# System Programs

- System programs provide a convenient environment for program development and execution
  - File manipulation
  - Status information: time, space, performance, …
  - File modification: text editors
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs

- Most users' view of the operation system is defined by system programs, not the actual system calls

# System Programs

- Some of system programs just indirect 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 asks the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a **registry** - used to store and retrieve configuration information

# System Programs (Cont.)

- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

# System Programs (Cont.)

- **Background Services**
  - Launch at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown
  - Provide facilities like disk checking, process scheduling, error logging, printing
  - Run in user context not kernel context
  - Known as **services** (in Windows), subsystems, **daemons** (in Unix)

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

# Operating System Design and Implementation

- No perfect solution exists, but some approaches could be successful in some environment
  - Ex> iOS -> iPhone (o), Android -> microwave, gps (x)
- Internal structure of different Operating Systems varies widely
- Start the design by defining goals and specifications
- Affected by choice of hardware and 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

# Operating System Design and Implementation (Cont.)

- Important principle to separate

  **Policy**: **What** will be done?

  **Mechanism**: **How** to do it?

- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later
  - Ex> The timer is a mechanism for ensuring CPU protection, but deciding how long OS should wait for a particular user is a policy decision
- Specifying and designing an OS is highly creative task of software engineering

# Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Usually multiple languages are mixed up to build OS
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- High-level languages are easier to **port** to other hardware
  - But slower
- **Emulation** can allow an OS to run on non-native hardware

# Operating System Structure

- General-purpose OS is a very large program

  - To function properly and be modified easily, OS is designed by partitioning the task into small components, or **modules**, rather than have one **monolithic** system.

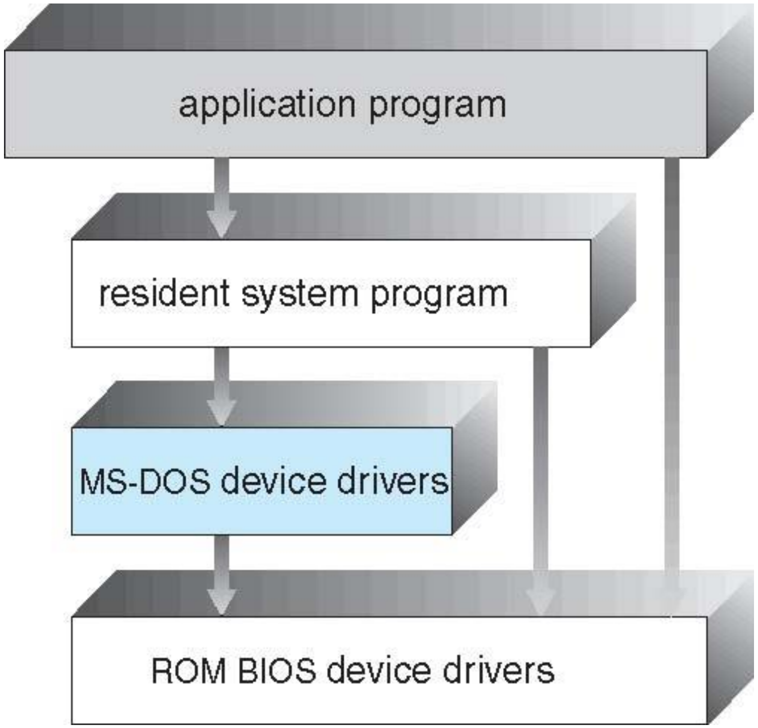
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex – UNIX
  - Layered – an abstraction
  - Microkernel – Mach (not Mac)

# Simple Structure -- MS-DOS

- MS-DOS – written to provide the most functionality in the least space

  - Not divided into modules

  - Although MS-DOS has a structure, its interfaces and levels of functionality are not well separated

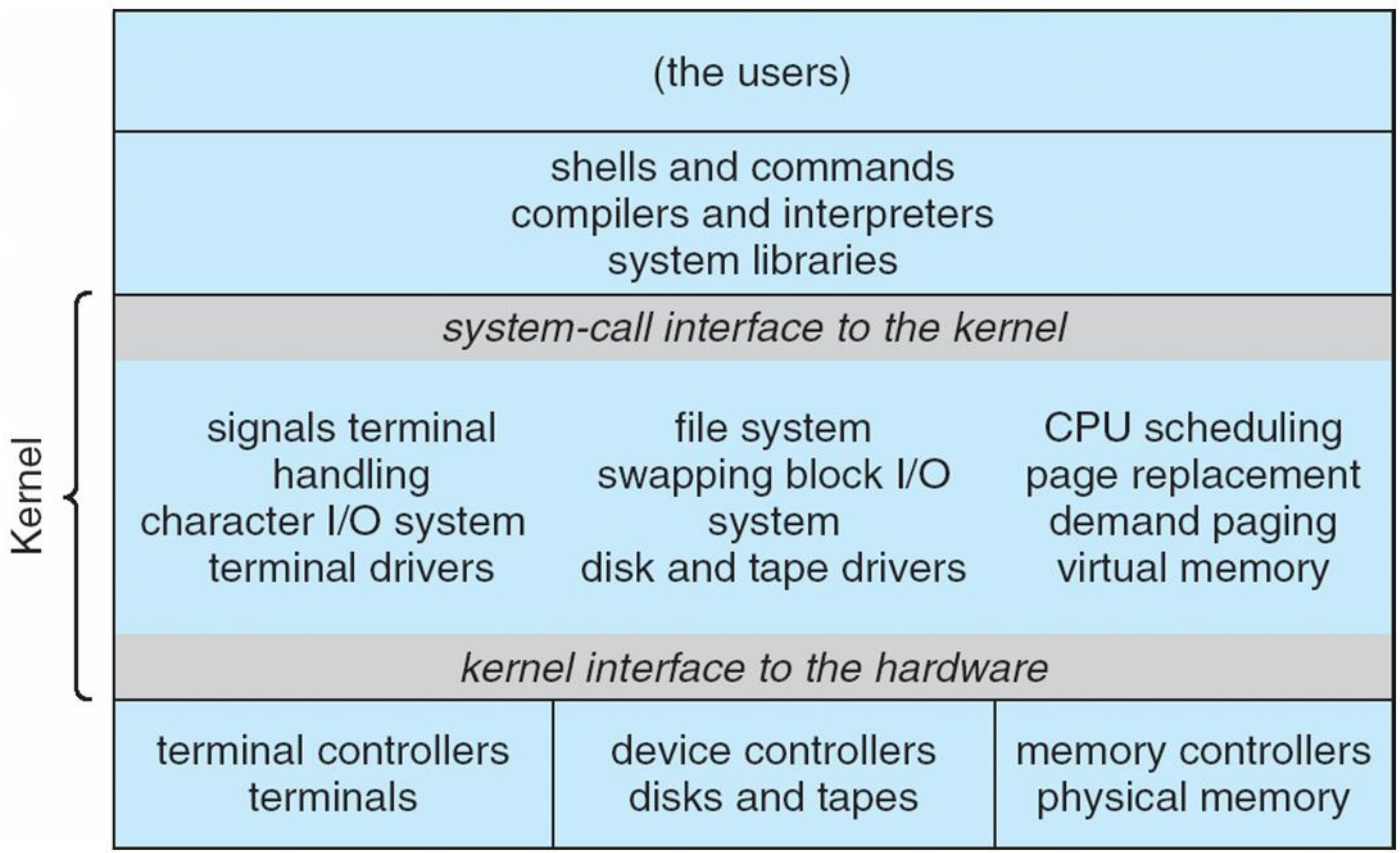


# Non Simple Structure -- UNIX

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

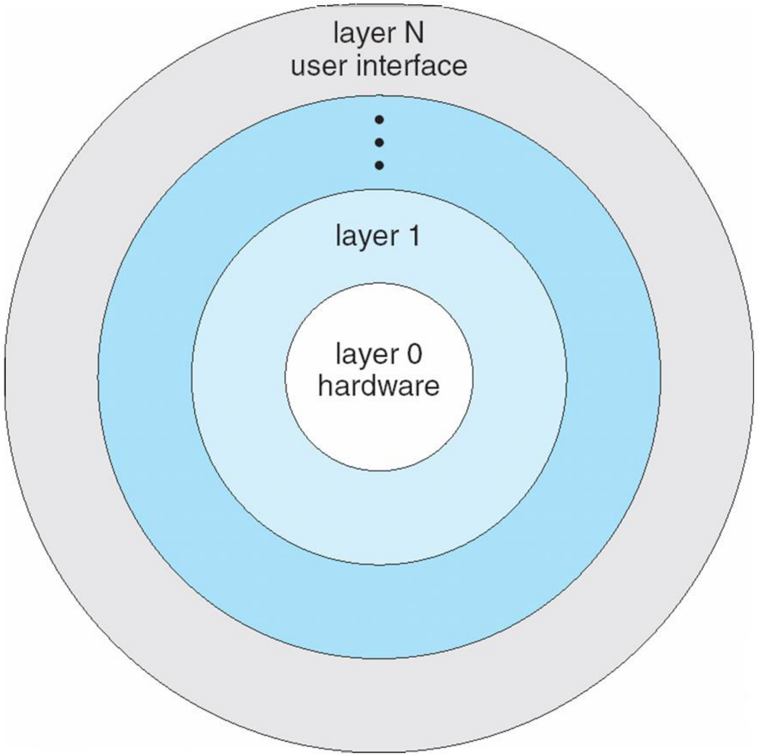


# Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers.

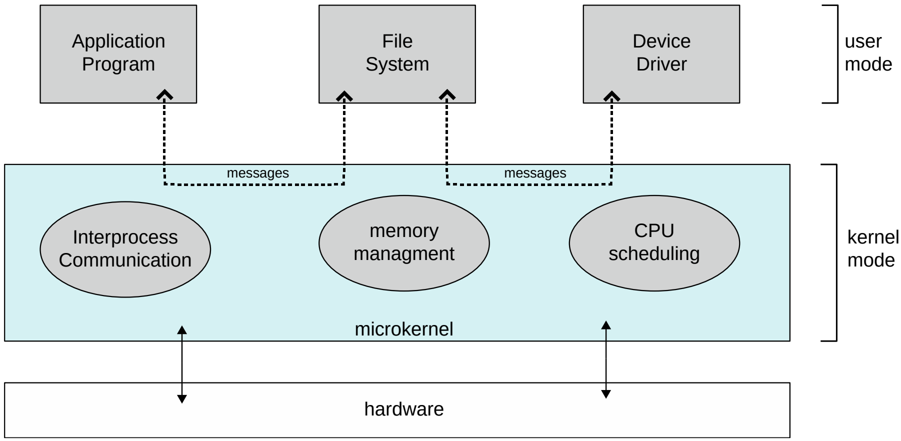
- The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.

- 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 as possible 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

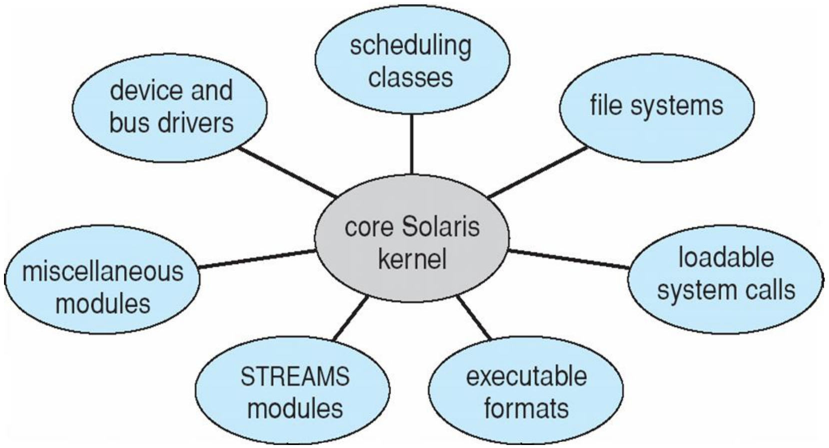



- Detriments:
  - Performance overhead of user space to kernel space communication

# 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

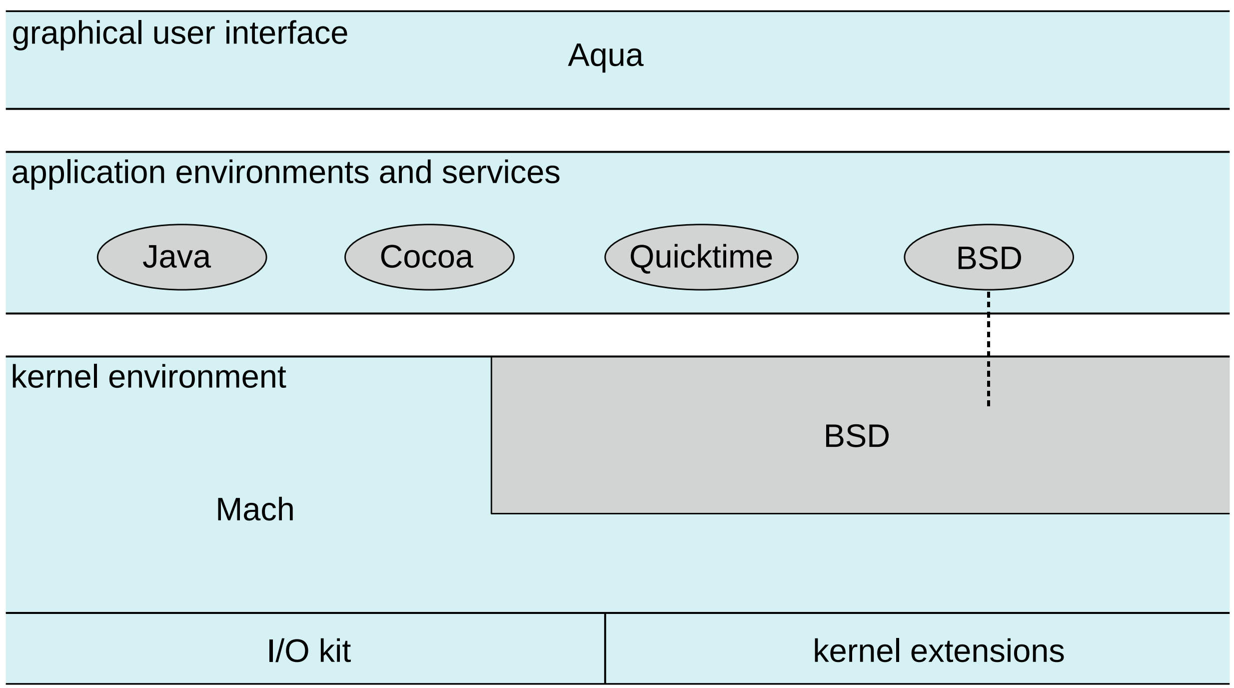


# Hybrid Systems

- Most modern operating systems actually do not use 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, but retains some microkernel behavior such as separate subsystems (known as operating-system ***personalities***). It supports dynamically loadable kernel modules

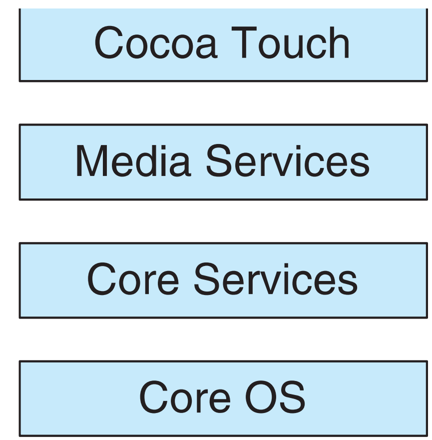
# Mac OS X Structure

- Apple Mac OS X: hybrid structures, layered system
  - Top layers include **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**)



# iOS

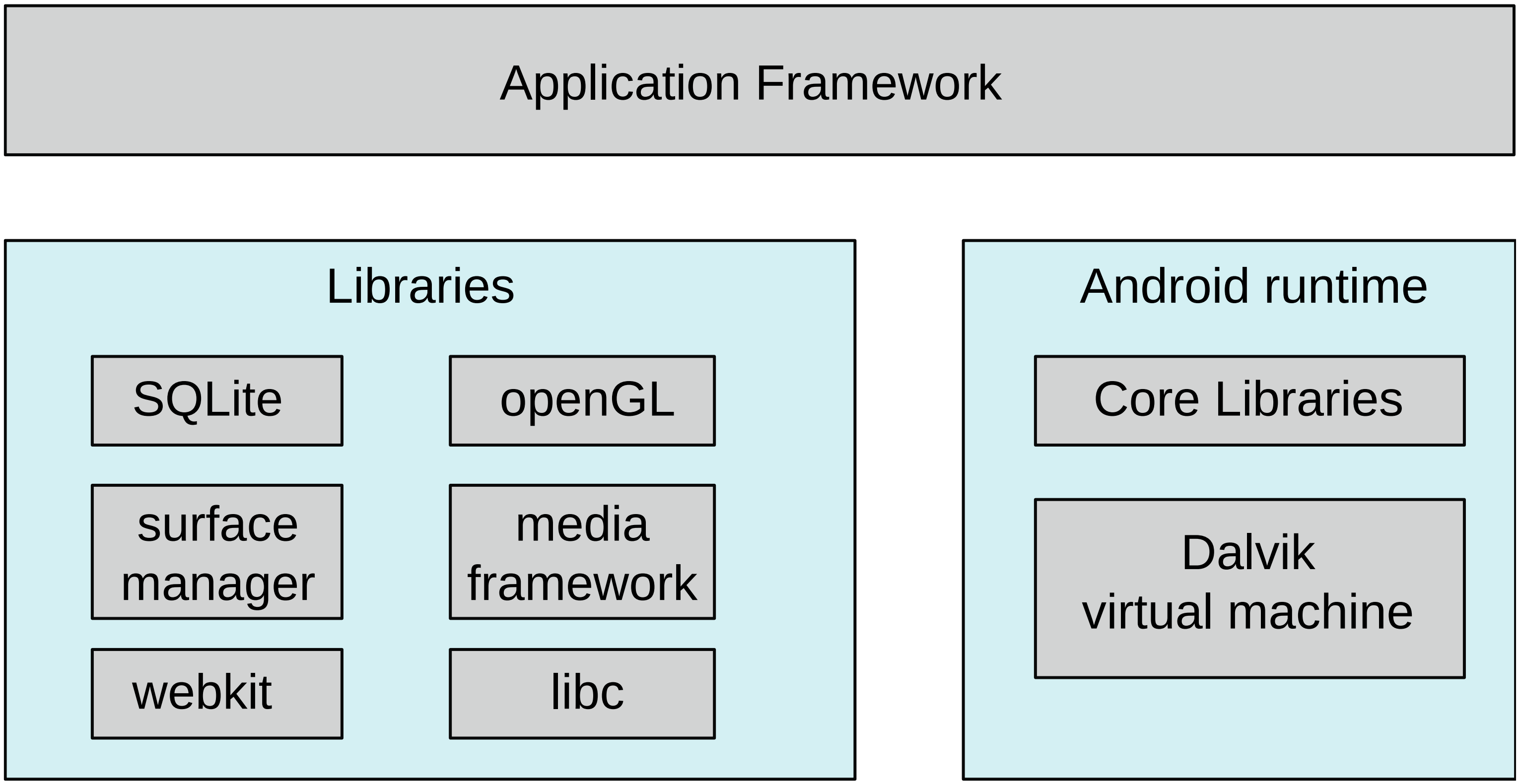
- Apple mobile OS for iPhone, iPad
  - Structured on Mac OS X, added functionality
  - Does not run OS X applications natively
    - Runs on different CPU architecture (ARM vs. Intel)
  - Cocoa Touch Objective-C API for developing apps
  - Media services layer for graphics, audio, video
  - Core services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



# 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 (replaced by Android Runtime (ART) from 5.0)
  - 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



# 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

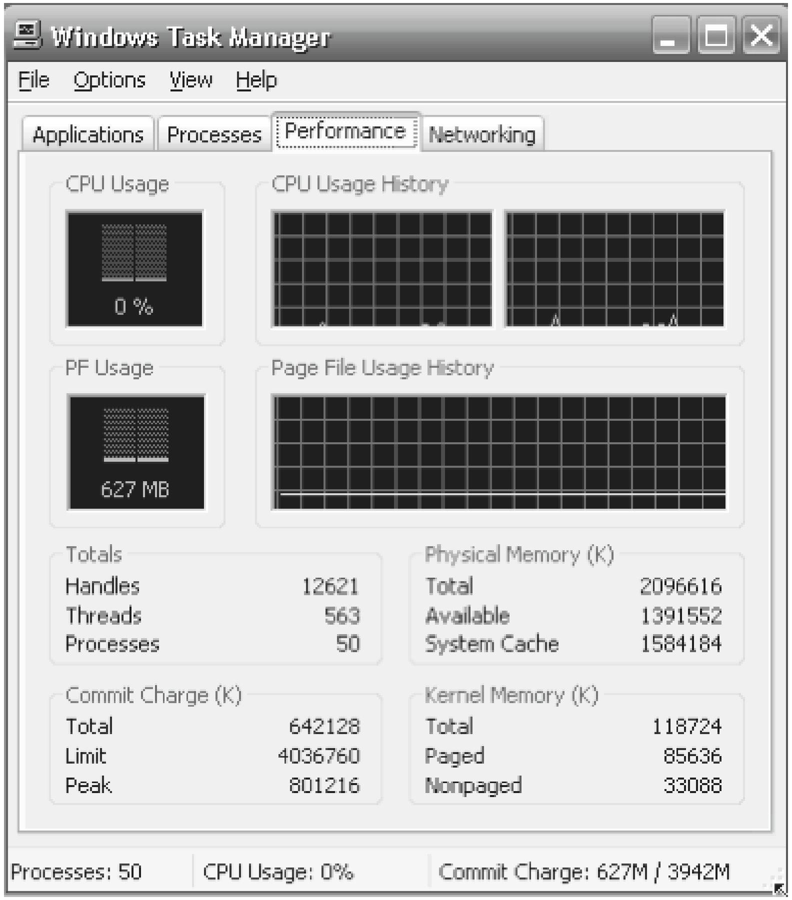
- Operating system failure can generate crash dump file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
  - Sometimes using trace listings of activities, recorded for analysis
  - Profiling is periodic sampling of instruction pointer to look for statistical trends

# Operating-System Debugging

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" (table of processes) program or Windows Task Manager



# DTrace

- DTrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems
  - A **consumer** requests providers create probes. **Probes** fire when code is executed, capturing state data and sending it to the consumer of those probes
- Example of following XEventsQueued system call (ioctl()) move from libc library to kernel and back (showing the functional calls)

```
# ./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
```

# Dtrace (Cont.)

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

```
sched:::on-cpu
uid == 101
{
   self->ts = timestamp;
}

sched:::off-cpu
self->ts
{
   @time[execname] = sum(timestamp - self->ts);
   self->ts = 0;
}
```

"sched.d"

```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
   gnome-settings-d         142354
   gnome-vfs-daemon         158243
   dsdm                     189804
   wnck-applet              200030
   gnome-panel              277864
   clock-applet             374916
   mapping-daemon           385475
   xscreensaver             514177
   metacity                 539281
   Xorg                    2579646
   gnome-terminal          5007269
   mixer_applet2           7388447
   java                   10769137
```

**Figure 2.21** Output of the D code.

# 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 (system generation)** program obtains information concerning the specific configuration of the hardware system
  - Used to build system-specific compiled kernel or system-tuned
  - Can generate more efficient code than one general kernel

# 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
  - The loader is updated to main memory and starts its code
  - 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

# End of Chapter 2