Chapter 2: Operating-System Structures

Coverage:

1. User OS Interface
2. System Calls
3. Types of System Calls
4. System Programs
5. Operating System Design and Implementation
6. Operating System Structure
7. Operating System Debugging
8. Operating System Generation
9. System Boot

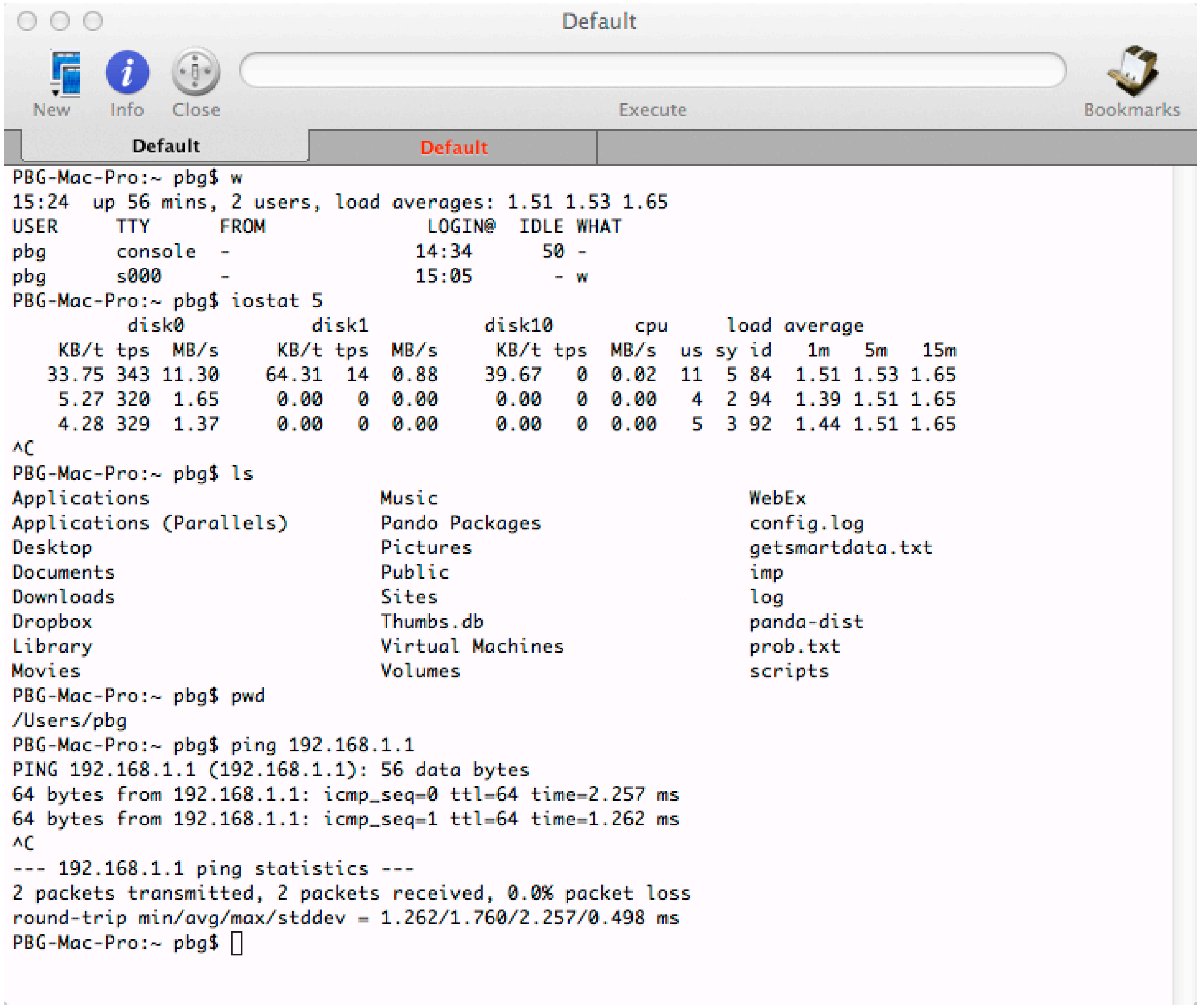
Objectives

- To describe the 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, customized and how they boot.

User Operating System Interface - CLI

- CLI (Command Language Interface) or **command interpreter** allows direct command entry.
    - Sometimes implemented in kernel, sometimes by systems programs.
    - Sometimes multiple flavors implemented – **shells.**
    - Primarily fetches a command from user and executes it.
        - Sometimes commands are built-in, sometimes just names of programs.
            - If the latter, adding new features doesn't require shell modification.
- Graphical User Interfaces (GUIs)
- Batch OS (processing) Interfaces. Still in use with main frame computers, and others. Read here, here, and here.

Bourne Shell Command Interpreter



```
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$ 
```

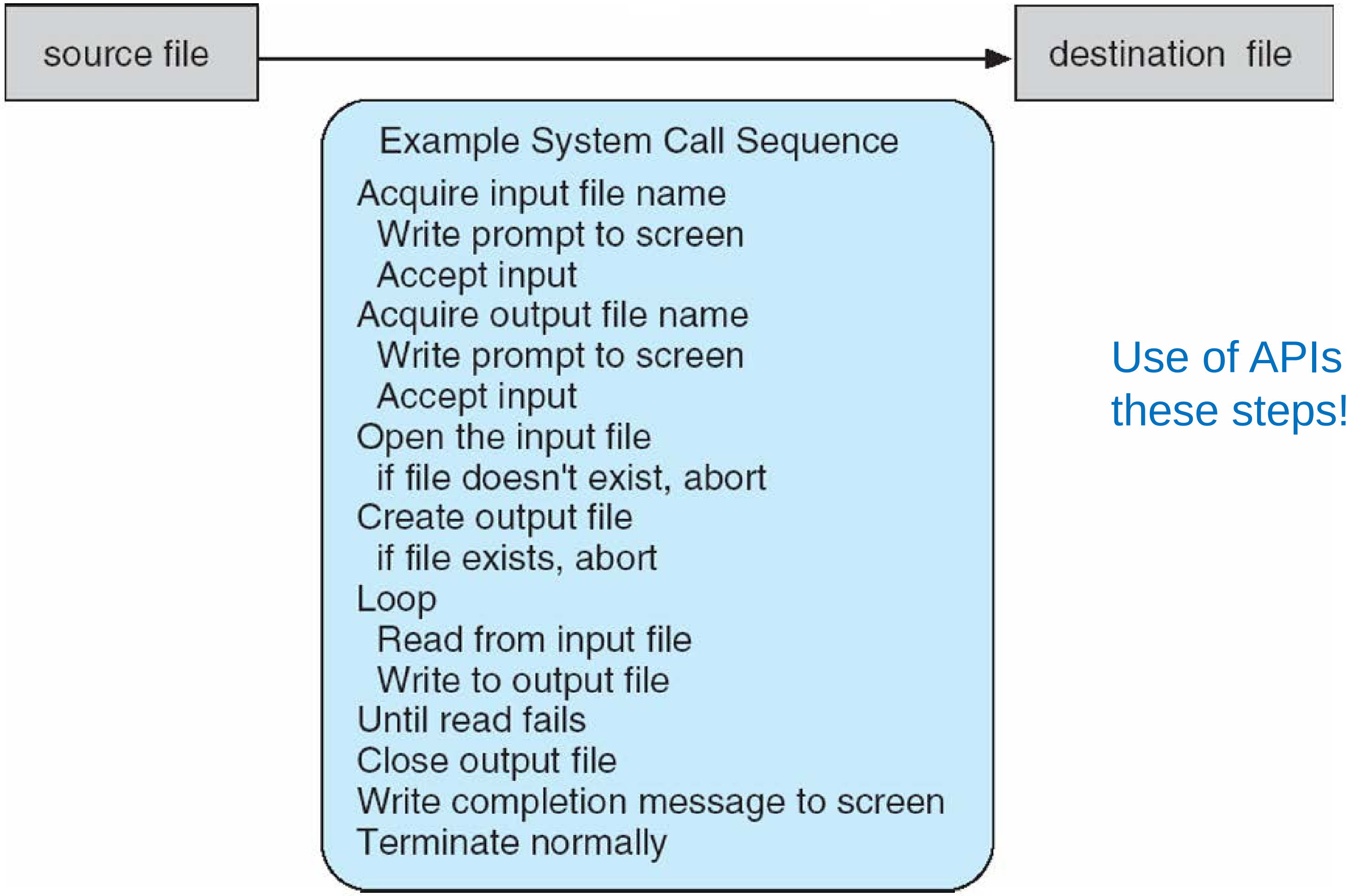
System Calls

- 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 Program 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).
- Why use APIs rather than system calls?

(Note that the system-call names used throughout your textbook are generic.)

Example of System Calls

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

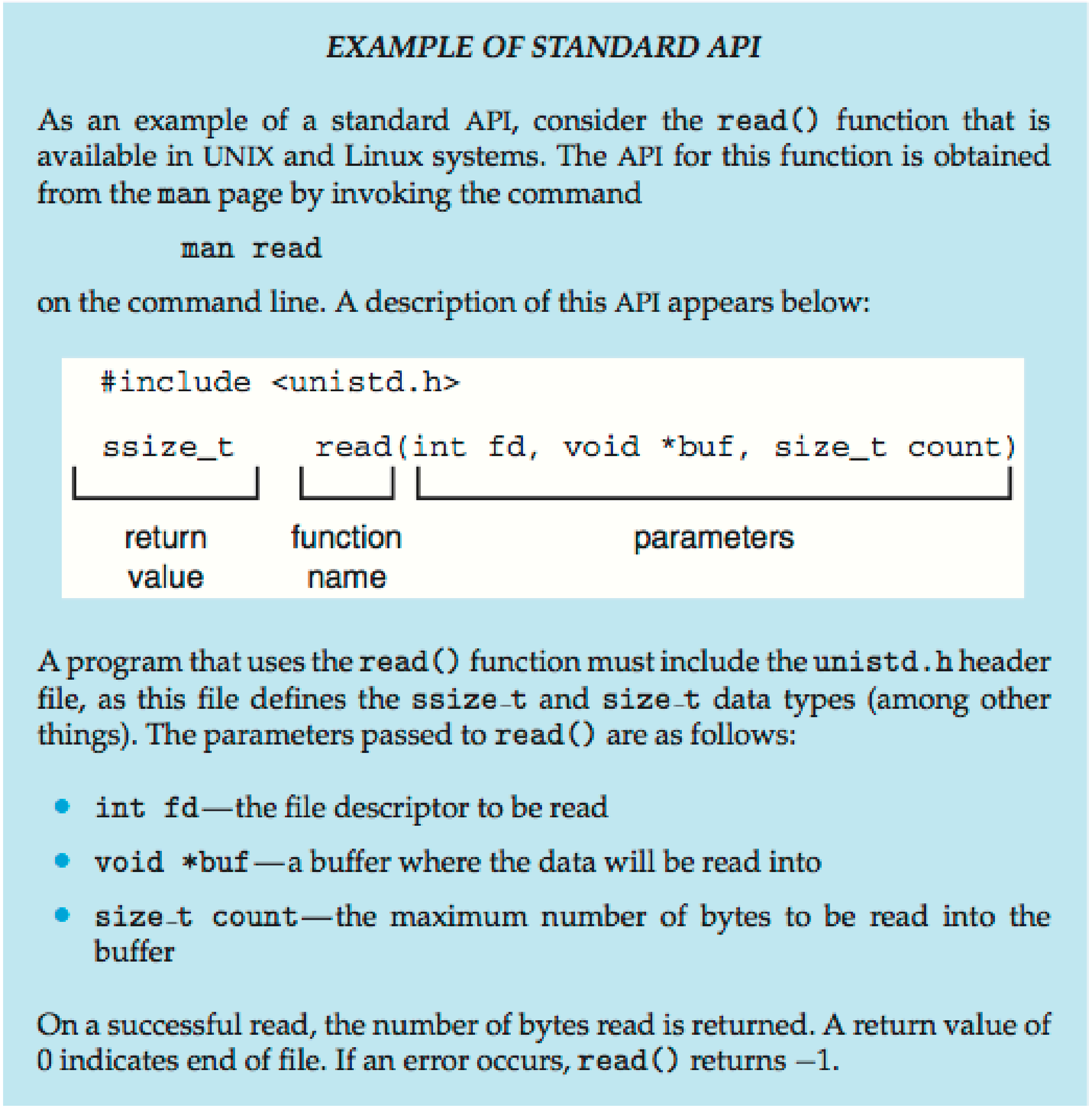


Example System Call Sequence

```
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
```

Use of APIs simplifies these steps!

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>
```

$$\underbrace{\texttt{ssize\_t}}_{\text{return value}}\quad\underbrace{\texttt{read}}_{\text{function name}}\underbrace{\texttt{(int fd, void *buf, size\_t count)}}_{\text{parameters}}$$

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:

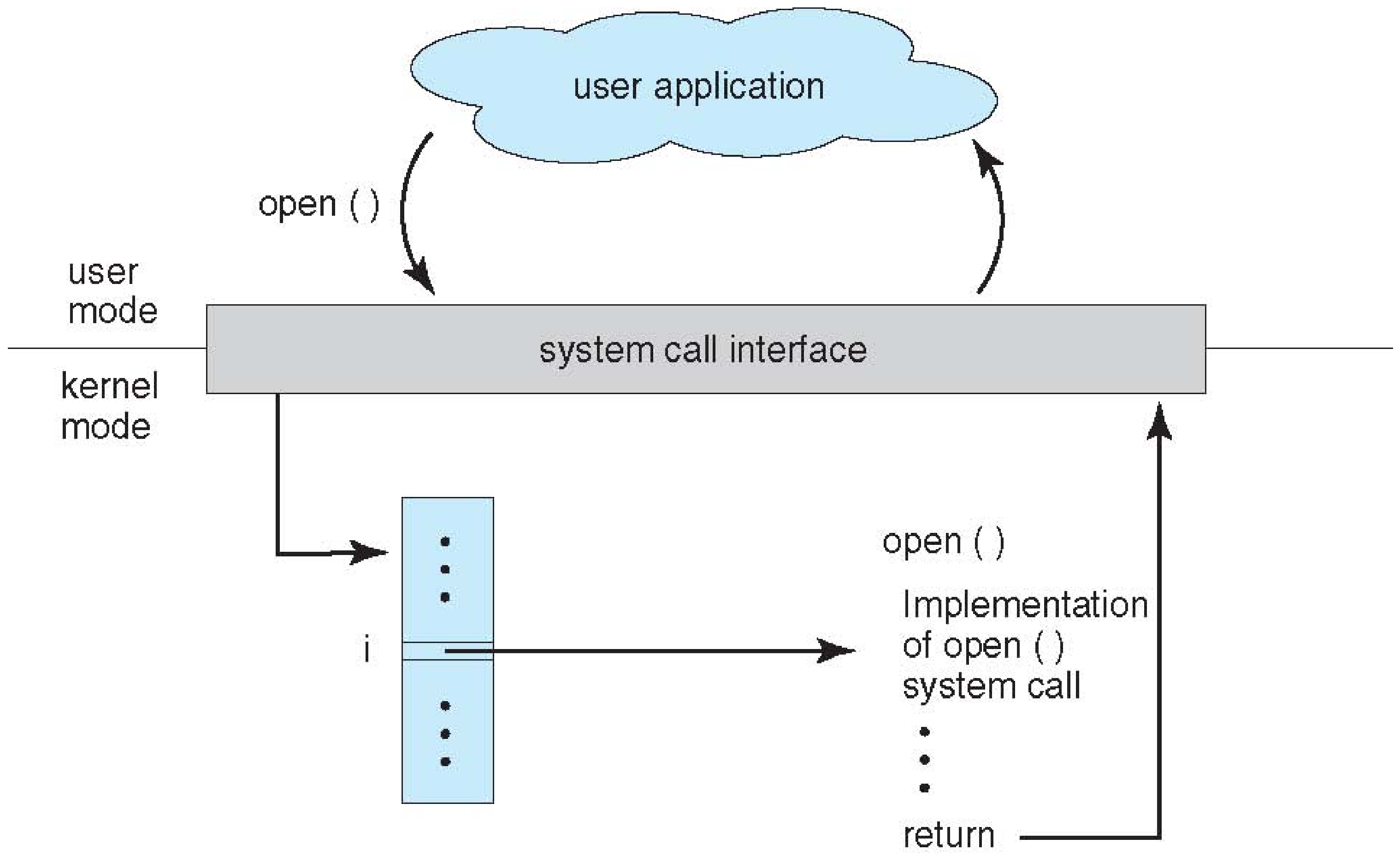
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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.

System Call Implementation

- Typically, a unique number is associated with each system call.
  - **System-call interface** maintains a table indexed according to these numbers.
- The system call interface invokes intended system call in OS kernel, and returns status of the system call and any return values.
- The caller does not need to know 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 the API.
    - Managed by run-time support library (set of functions built into libraries included with compiler).

API – System Call – OS Relationship

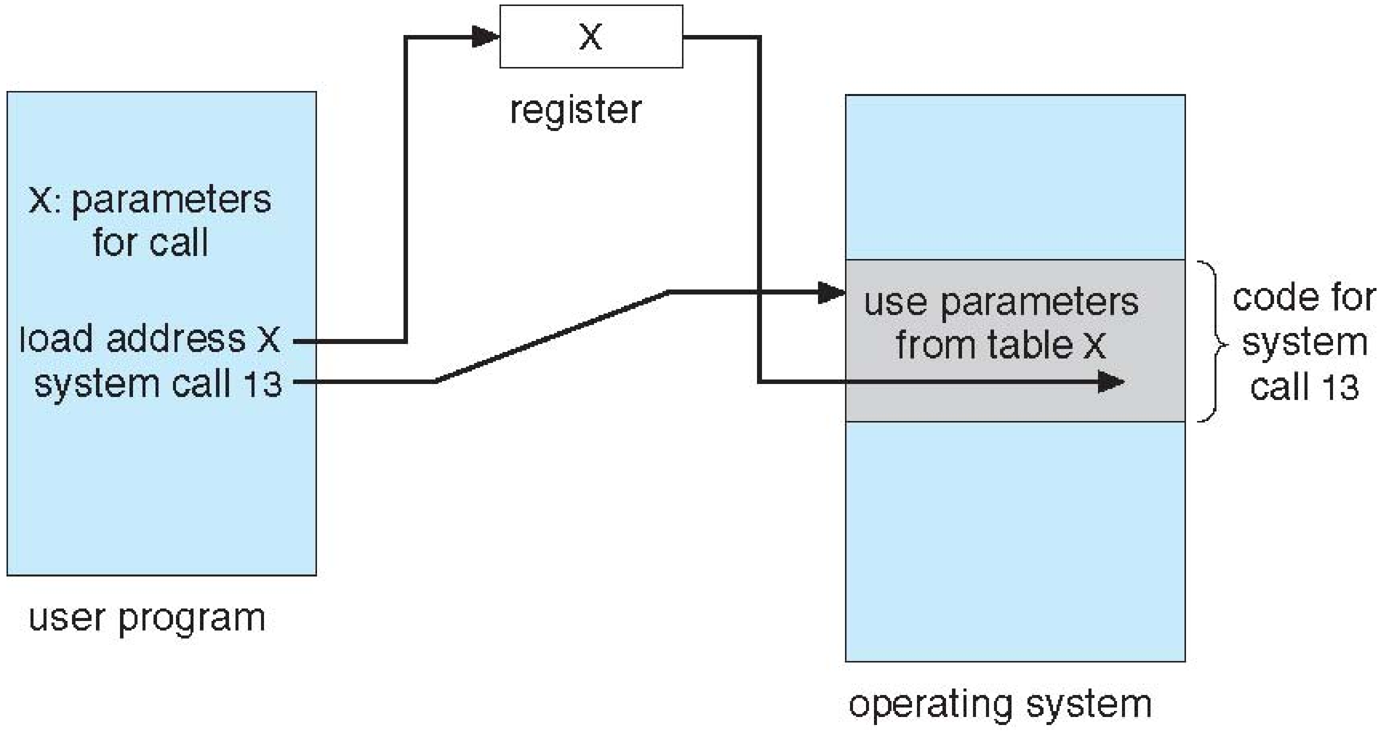


System Call Parameter Passing

- 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 are used to pass parameters to the OS:
  1. **Pass the parameters in registers.**
     - In some cases, may be more parameters than registers
  2. **Parameters are 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
  3. **Parameters are placed, or pushed, onto a 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
  - end, abort
  - load, execute
  - create process, terminate process
  - 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

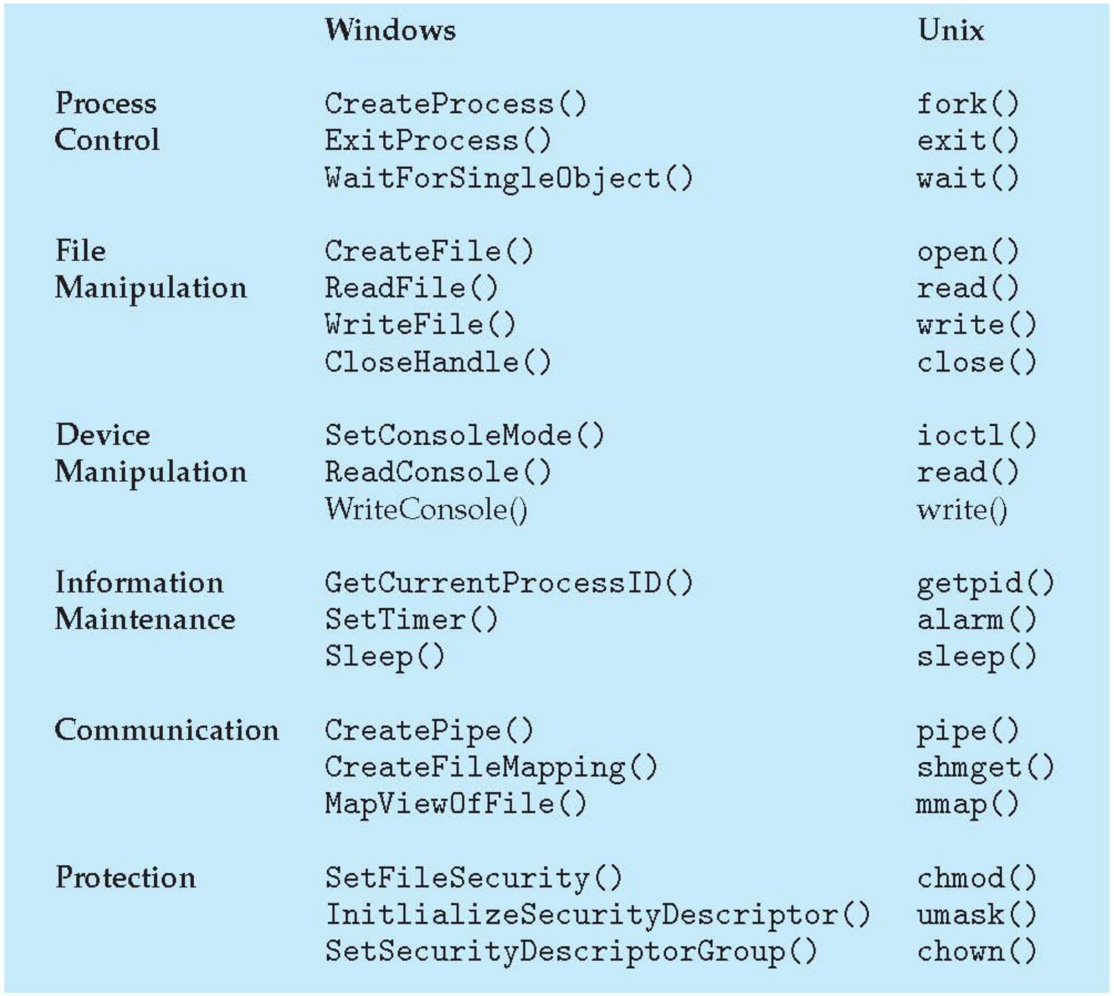
Types of System Calls (Cont.)

- 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**
    - 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

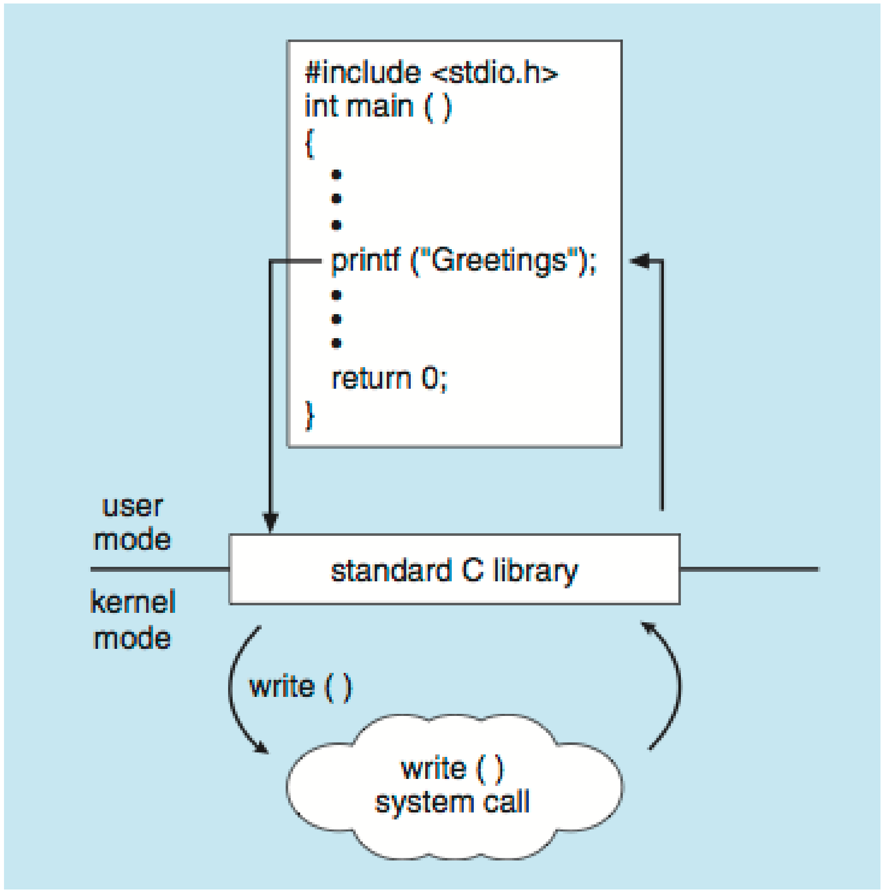
Examples of Windows and Unix System Calls



	Windows	Unix
Process Control	CreateProcess()	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

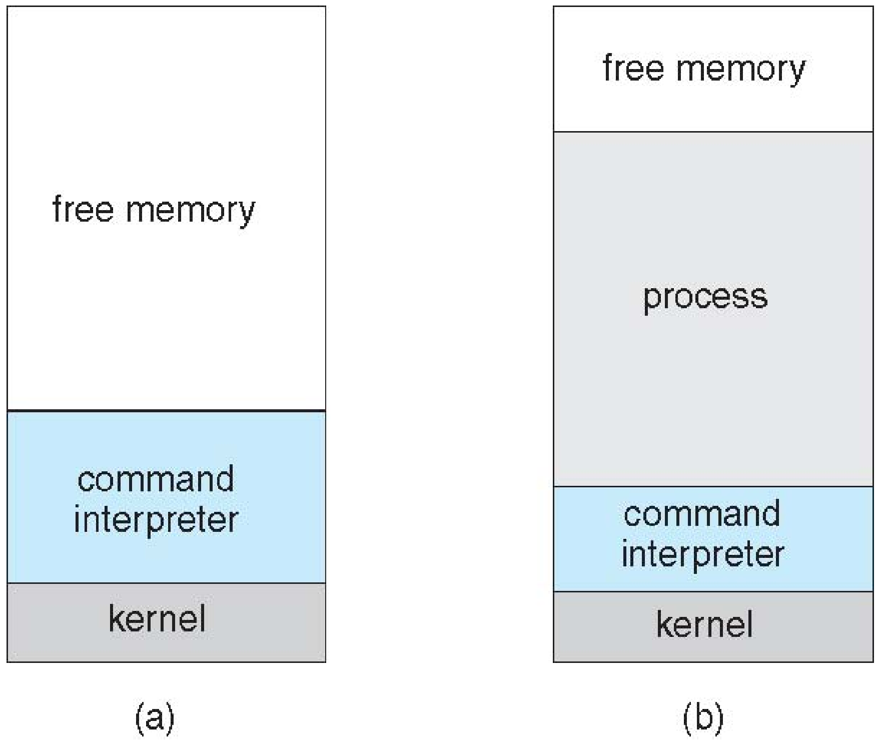
- C program invoking printf() library call, which calls write() system call



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

Example: MS-DOS

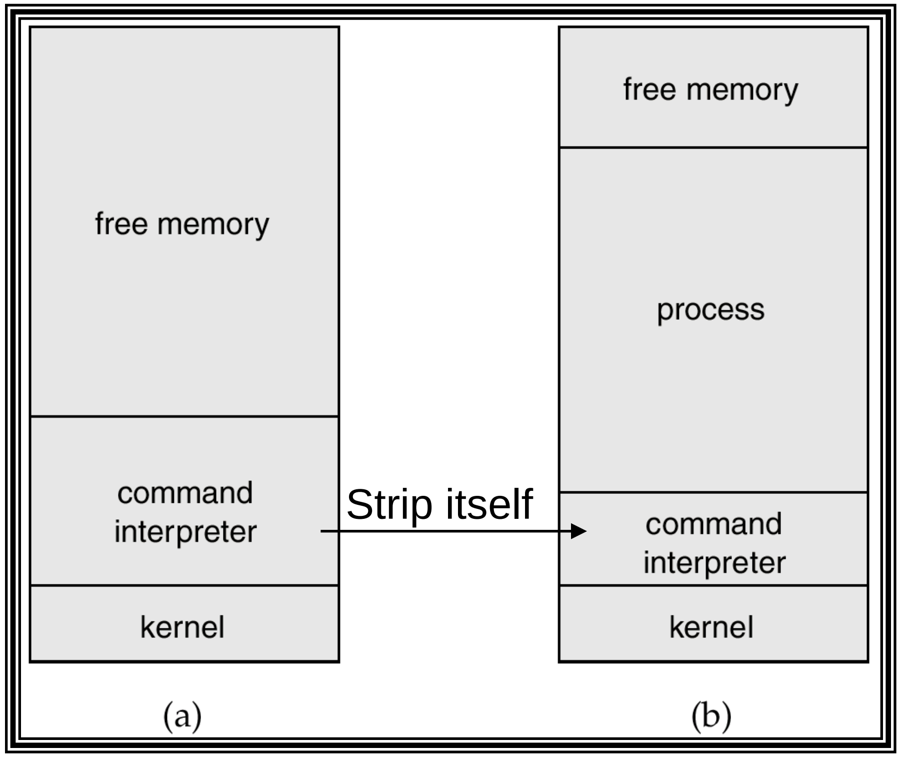
- Single-tasking
- Shell invoked when system booted
- Simple method to run program
  - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded



(a) At system startup (b) running a program

More on MS-DOS

- Load and execute programs
- **Single-tasking system**
- Control given to program
- Control returns to interpreter

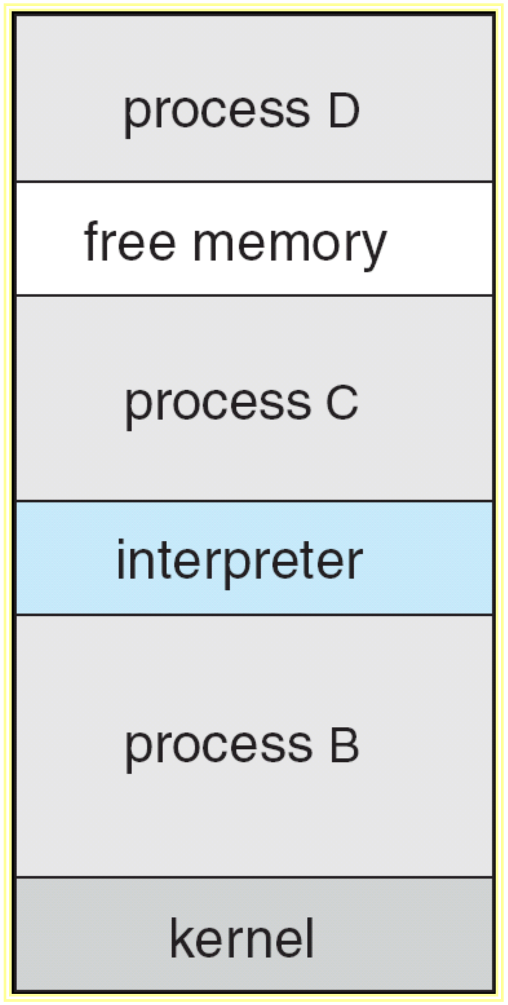


At System Start-up

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 of 0 – no error or > 0 – error code



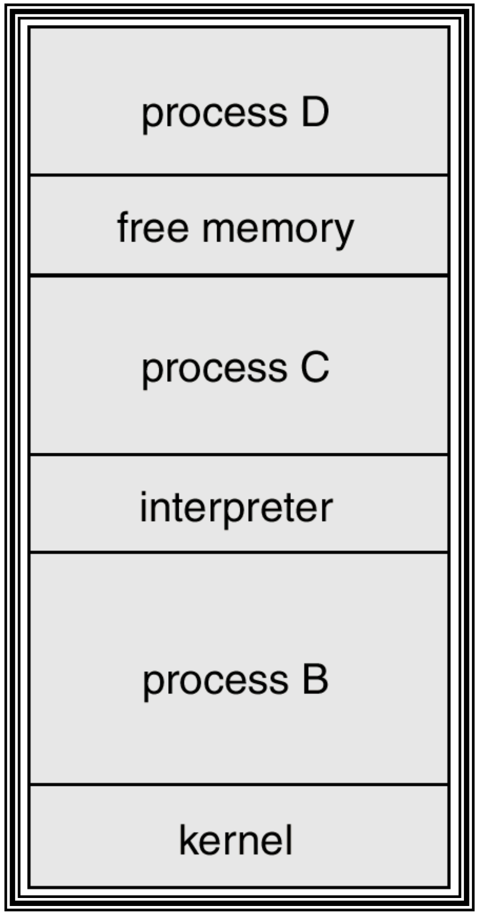
Process Control in UNIX

- **Multitasking system** (with multiple processes):
  - Command interpreter
  - Background processes:

    $ ./print_file &
    (shell executes a fork() systems call)
  - Can use ps to display status of processes:

    $ ps –al

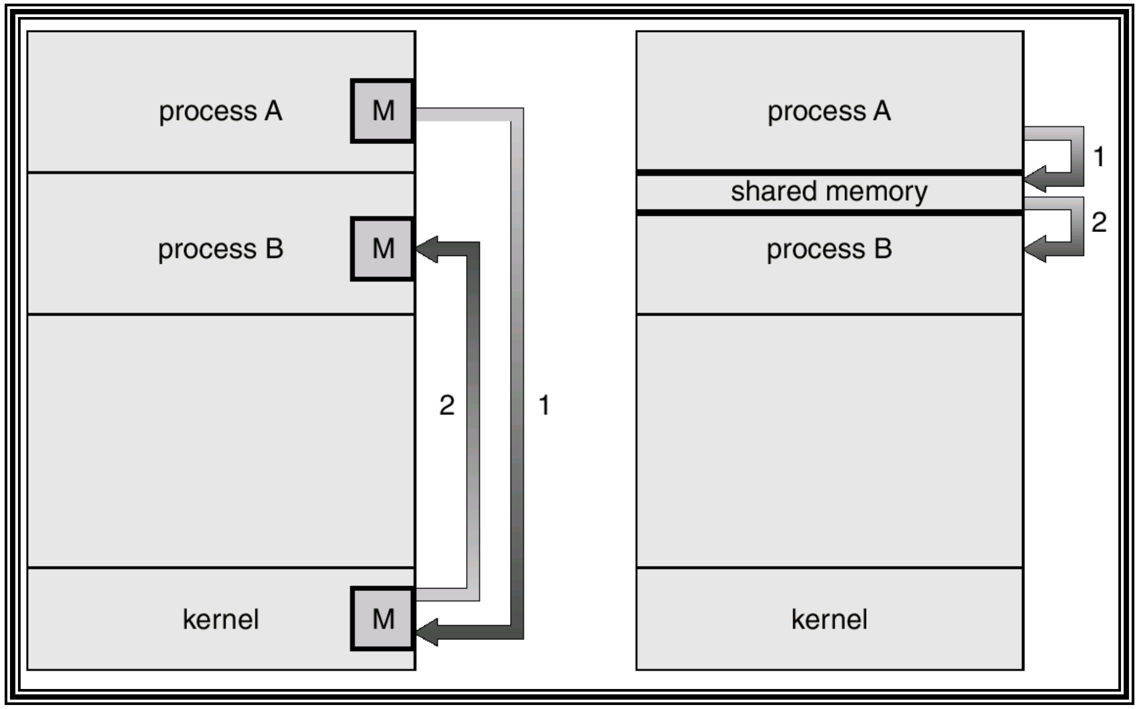
    $ ps -el



- Systems calls:
  - fork(), exec(), exit(), wait(), sleep(), etc

Communication

- Communication may take place using either message passing or shared memory.



Message Passing   Shared Memory

msgget(), msgsnd(), msgrecv()   shmget(), shmctl()

More on Command Interpreters

Many interpreters:
JCL Interpreter; control card interpreter; command-line Interpreter; shell

Unix Shell commands:

```
date
history
crypt <infile >outfile
df (file systems)
du (disk utilization)
last|more
cat a b c | sort > /dev/lp     OR     cat a b c | sort -Plp
```

System Programs

- 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 modification
  - 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 (System Utilities)

- Provide a convenient environment for program development and execution.
  - Some of them are simply user interfaces to system calls; others are considerably more complex.
- **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 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.
- **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**, **subsystems**, **daemons**.
- **Some Application programs**
  - Web browsers, editors, database systems, statistical packages, games (?), ….
    - Don’t pertain to the system.
    - Run by users.
    - Not typically considered part of OS.
    - Launched by command line, mouse click, finger poke.

Operating System Design and Implementation

- Design and Implementation of OS is not an “unsolvable problem”; but some approaches have proven successful.
- Internal structure of different Operating Systems can vary widely.
- Start 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.

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.

- Specifying and designing OS is highly creative task of **software engineering.**

Implementation

- Much variation
  - Early OS's 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

- Design and Implementation of OS is not an “unsolvable problem”; but “only” some approaches have proven successful.
- Internal structures of different OSs can vary widely.
- Start by defining goals and specifications.
- Affected by choice of hardware, type of system.
- User goals and System goals
  - User goals: OS--convenient to use, easy to learn, reliable, safe, and fast.
  - System goals: OS--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?

- 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.

- Specifying and designing OS is highly creative task of **software engineering.**

Implementation

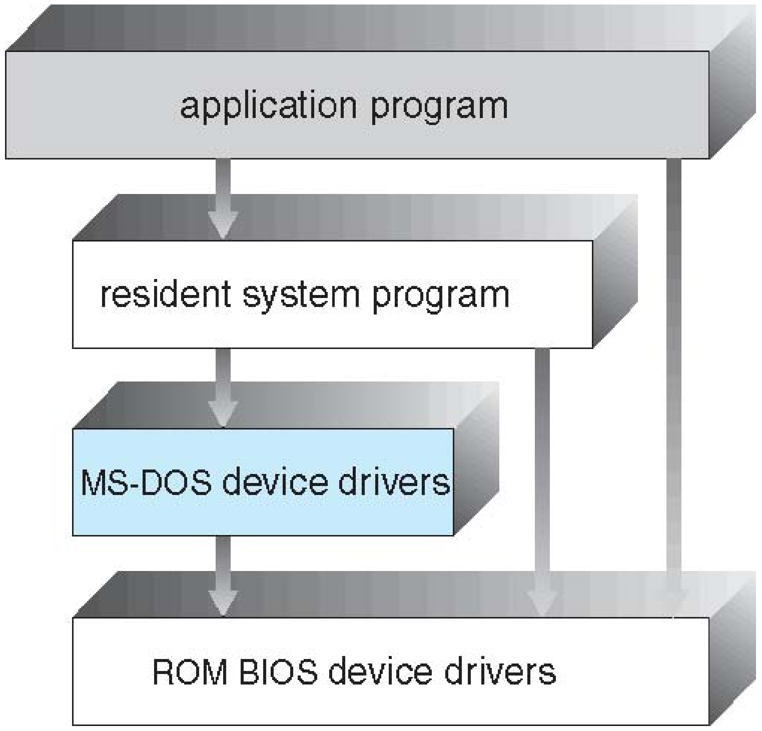
- Much variation
  - Early OS's in assembly language.
  - Then system programming languages like Algol, PL/1.
  - Now C, C++
- Actually usually a mix of languages.
  - Lowest levels in assembly.
  - Main body in C.
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts.
- 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 Structure

- General-purpose OS is a very large program.
- Various ways to structure one as follows.

Simple Structure

- 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.

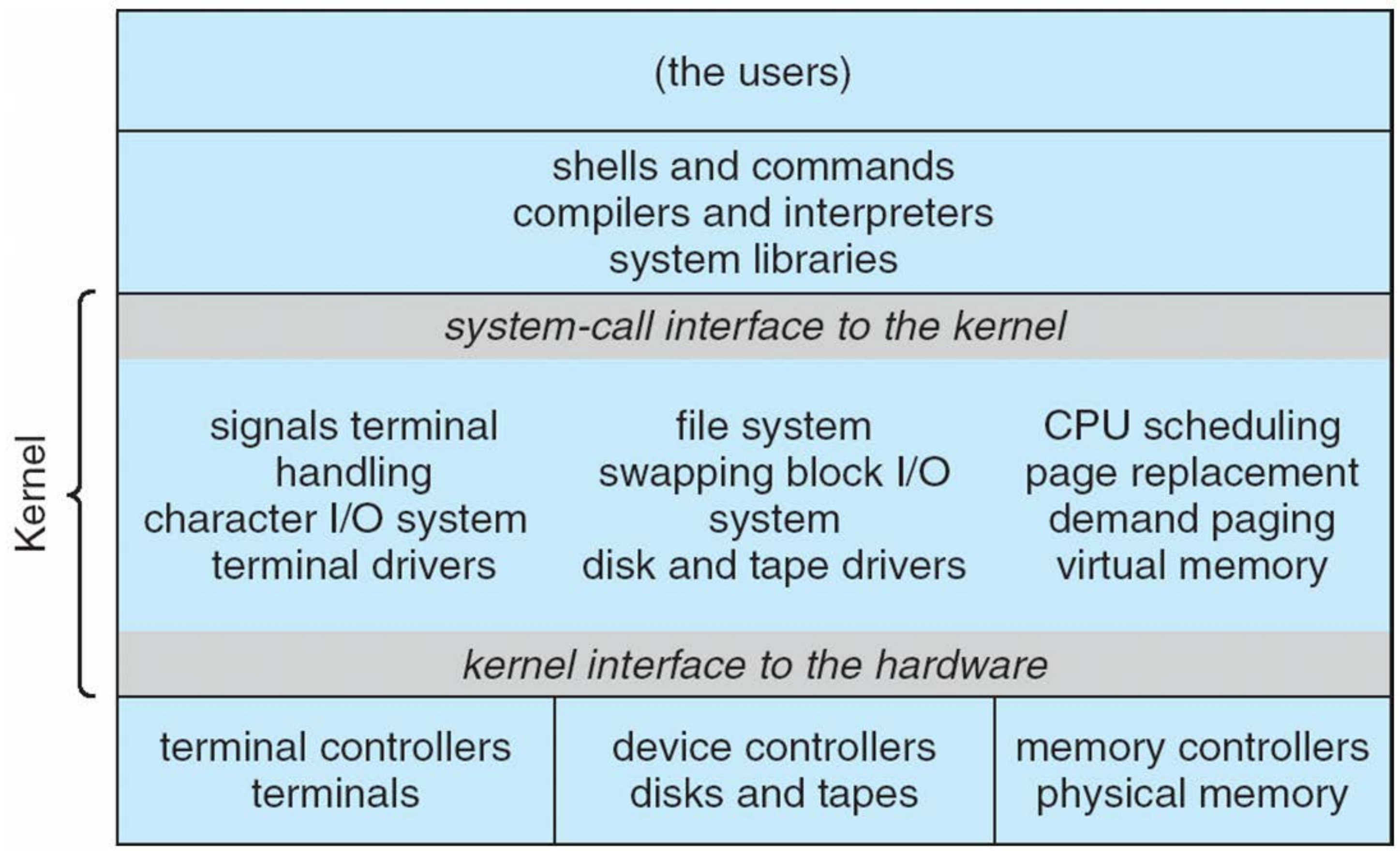


UNIX

- 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

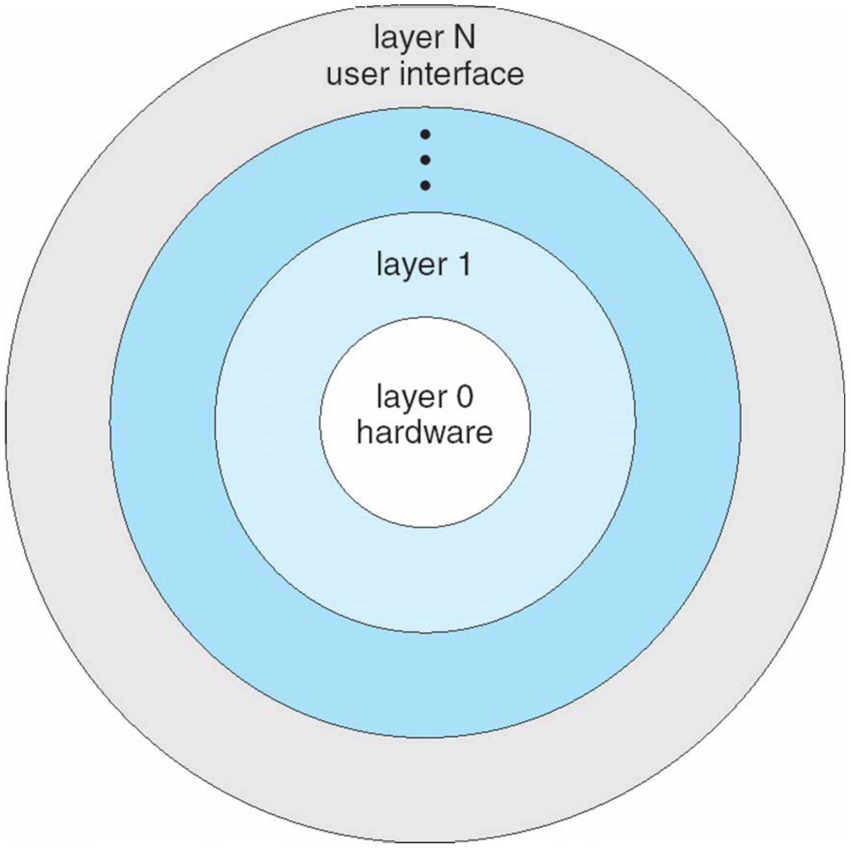
Beyond simple, but not fully layered.



Layered (but not modular) 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.

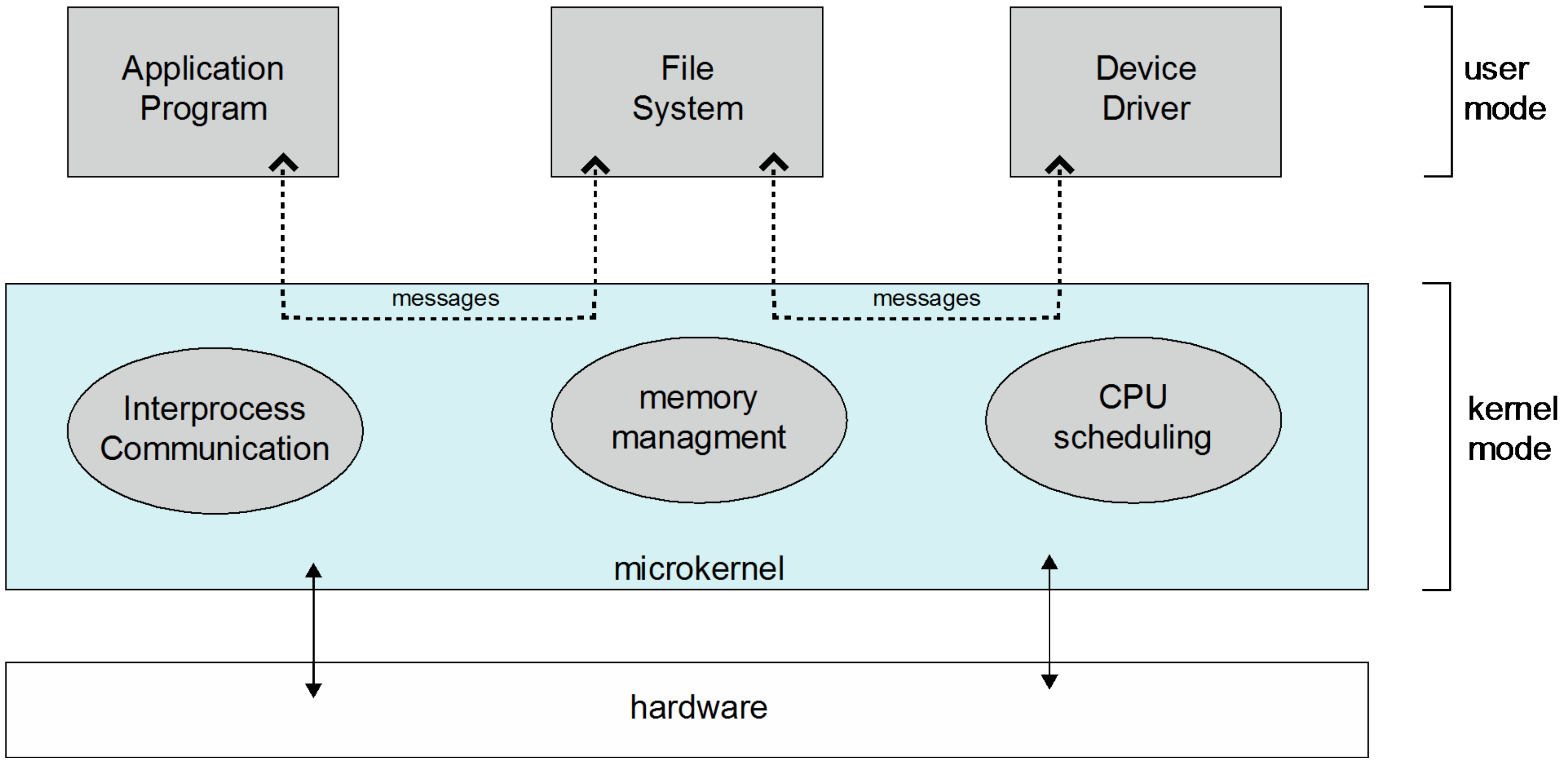
- With modularity, layers are selected such that each layer uses functions (operations) and services of only lower-level layers.



Microkernel System Structure

- Idea: Move code from the kernel into user space as much as possible
- Example: **Mach OS**
  - Mac OS X kernel (**Darwin**) partly based on Mach.
- Communication between user modules is via **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

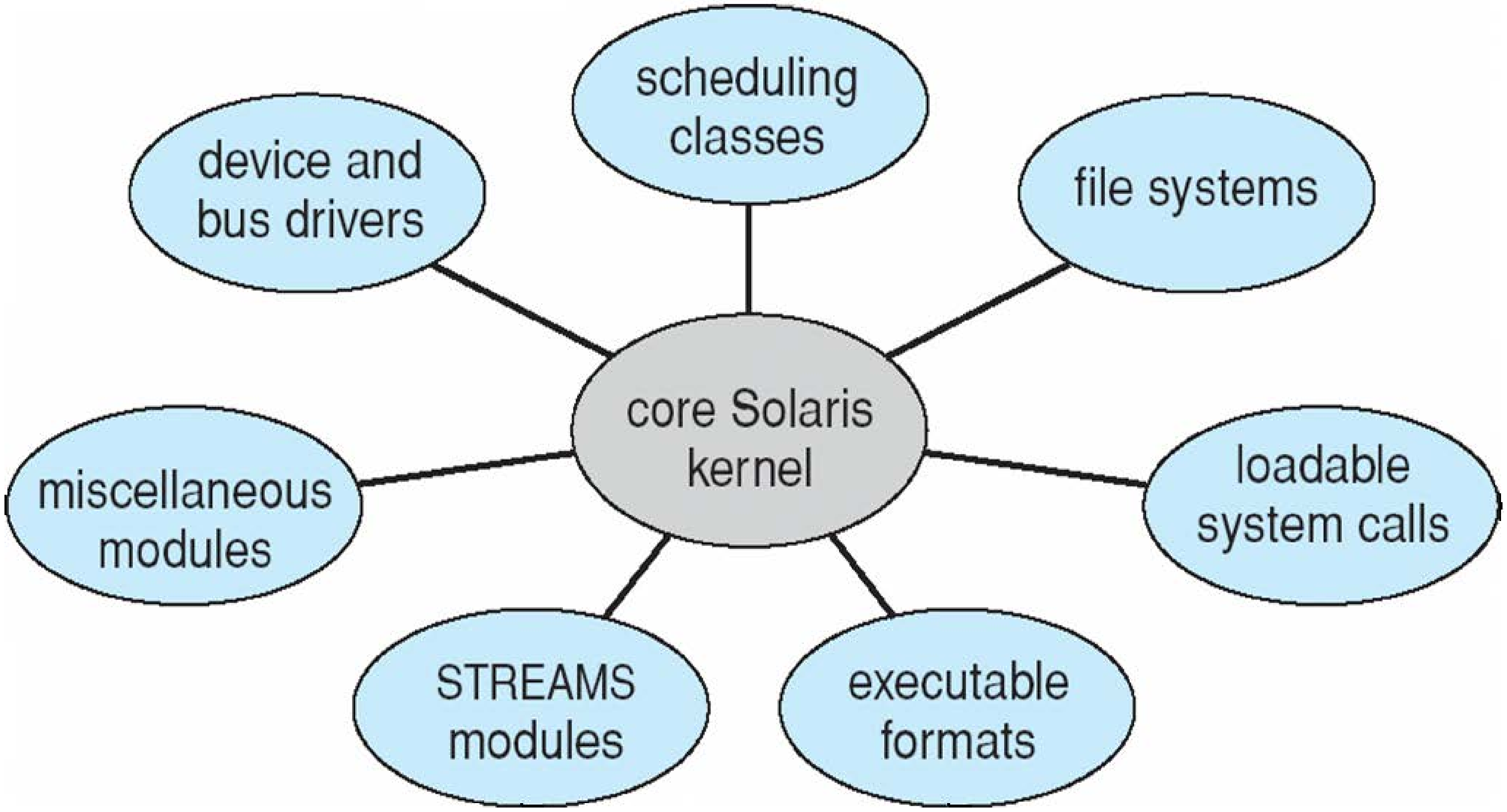


Modular Approach

- Most modern OSs implement **loadable kernel modules.**
  - Object-oriented approach is used.
  - Each kernel module is separate.
  - Each kernel module talks to the others over known interfaces.
  - Each kernel module is loadable as needed within the kernel.

- Overall, similar to layers, but provides more flexibility.
  - Examples: Linux, Solaris, etc.

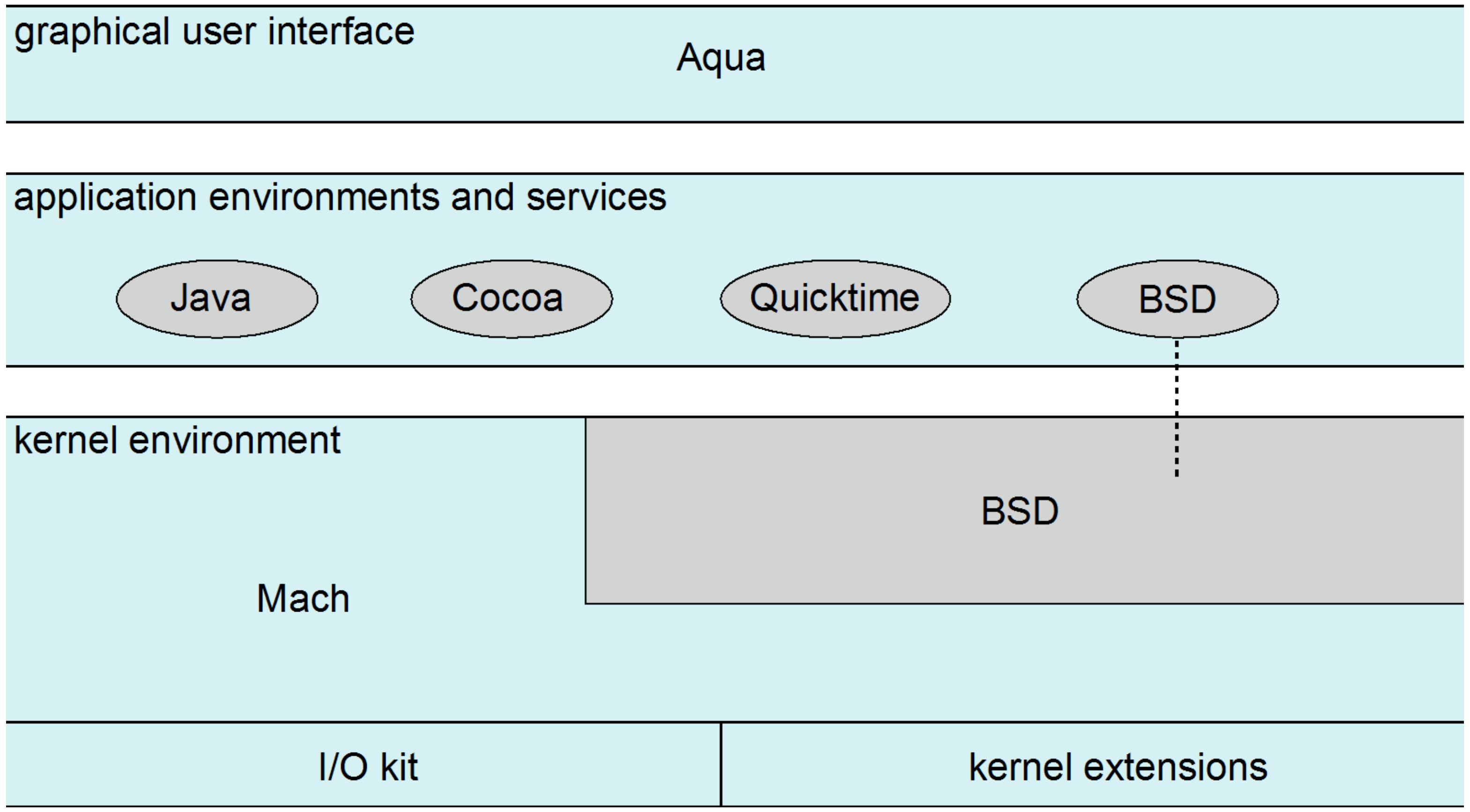
SUN Solaris Modular Approach



Hybrid Systems

- Most modern OSs do NOT use one pure model.
  - **Hybrid** combines multiple approaches to address performance, security, and usability needs.
  - **Linux and Solaris kernels** are in kernel address space; so monolithic. But, modular for dynamic loading of functionality.
  - **Windows mostly monolithic**. But, uses the microkernel approach for different subsystem ***personalities.***
- Apple (Mac) OS X (El Capitan: version 10.11) is hybrid, layered, **Aqua** UI plus **Cocoa** programming environment.
  - Kernel consists of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**).

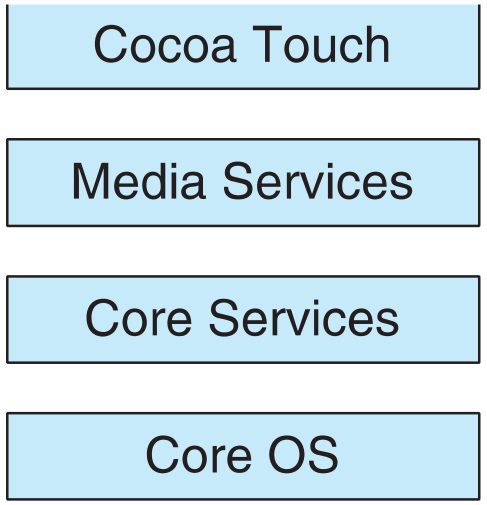
(Mac) OS X Structure



iOS 9 (as of 2015)

Apple mobile OS for **iPhone** and **iPad**

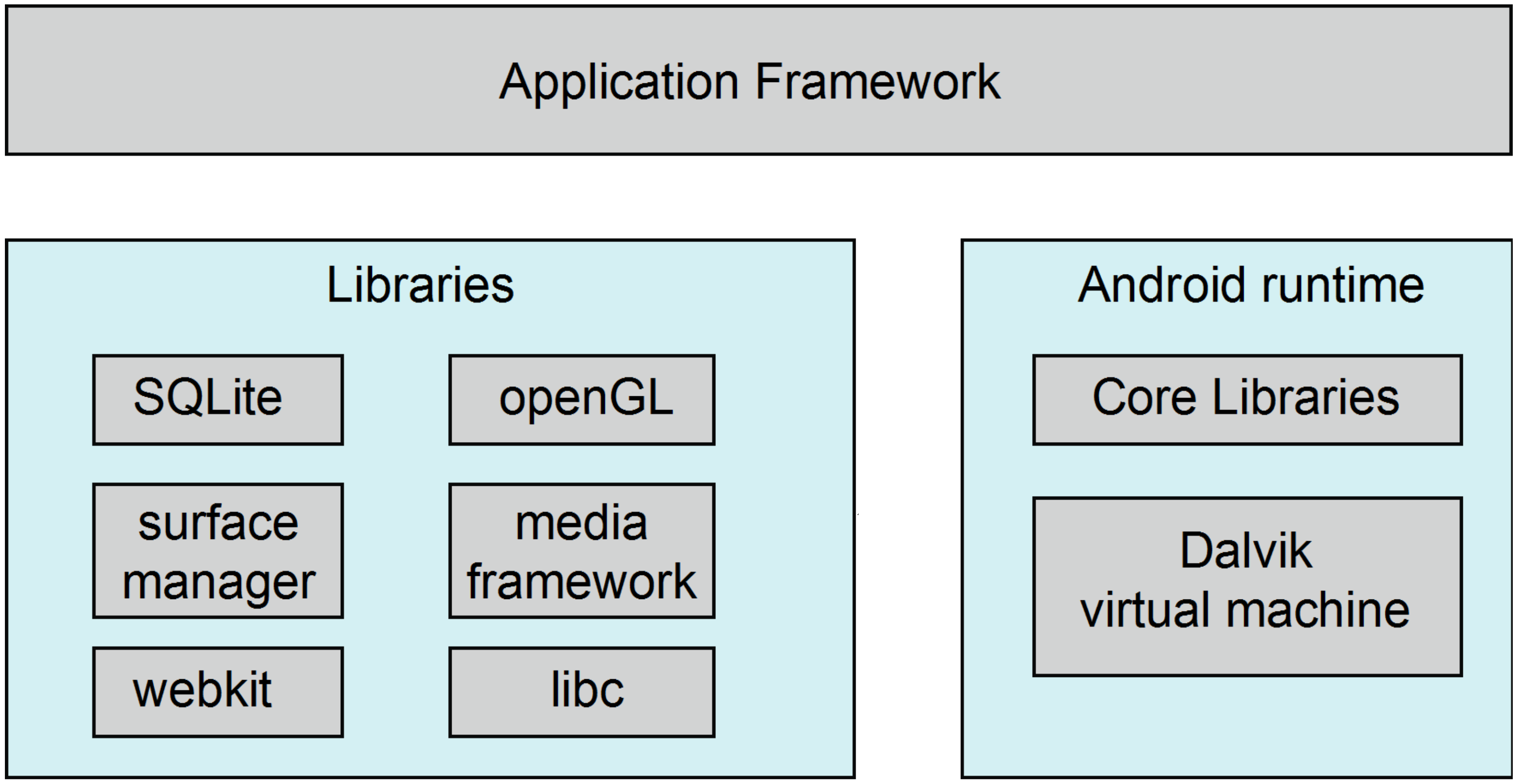
- Structured on Mac OS X, with added functionality.
- Does **not** run OS X applications natively.
  - Runs on different CPU architecture--ARM-based 64-bit multi-core system (Apple A8) on a chip versus Intel of OS X.
- **Cocoa Touch** Objective-C API for developing apps.
- **Media services** layer for graphics, audio, video.
- **Core services** provides cloud computing, databases.
- Core operating system is based on 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.
  - Apps developed in Java plus Android API via Android Studio.
    - Java class files compiled to Java bytecode, then translated to executable that runs in Dalvik VM.
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc (standard c library).

Android Architecture



Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs.**

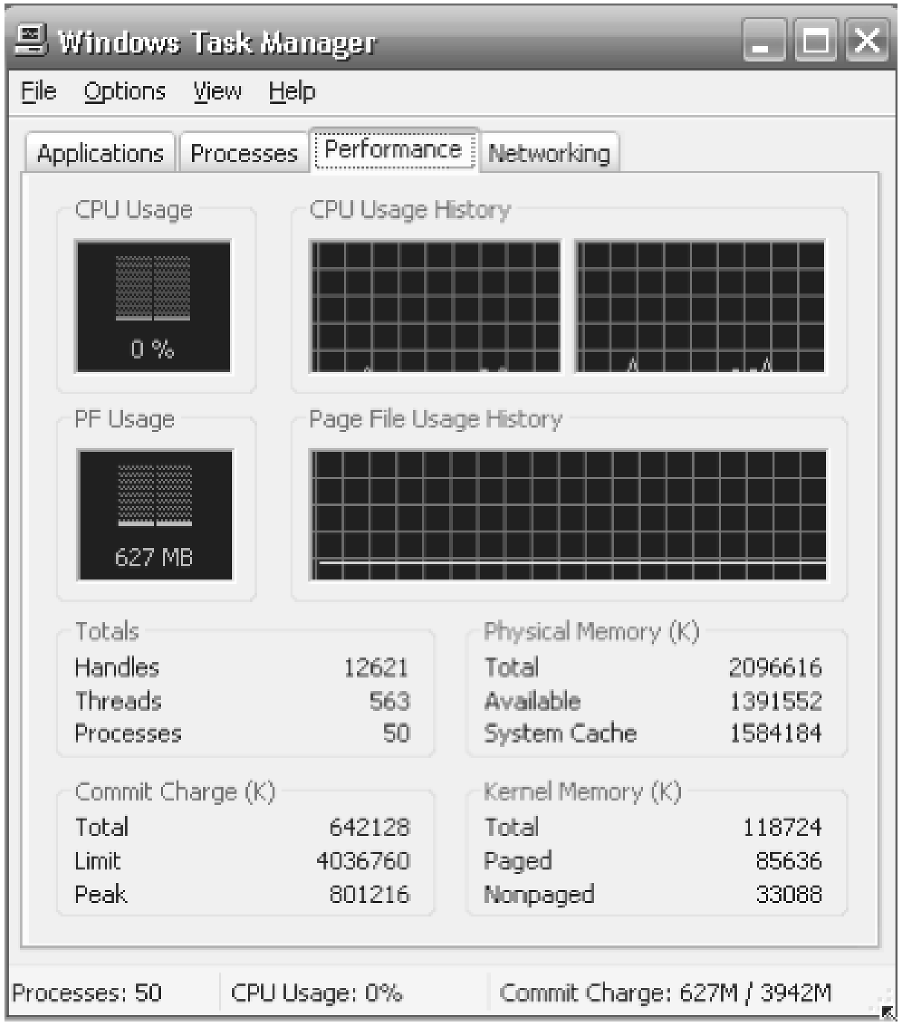
- OS's generate **log files** containing error information.

- 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."

OS Performance Tuning

- Improve performance by removing bottlenecks.
- OS must provide means of computing and displaying measures of system behavior.



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

Debugging: DTrace

- DTrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems.
- **Probes** fire when code is executed within a **provider**, capturing state data and sending it to **consumers** of those probes.
- Example of following XEventsQueued system call move from libc library to kernel and back.

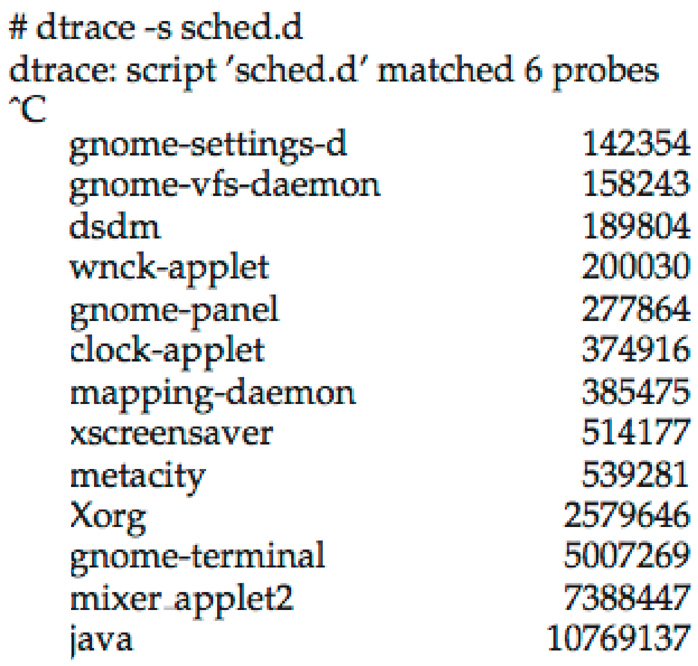
```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                    U
  0   -> _XEventsQueued                 U
  0     -> _X11TransBytesReadable       U
  0     <- _X11TransBytesReadable       U
  0     -> _X11TransSocketBytesReadable 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

- 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;
}
```



```
# 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

- OSs are designed to run on any of a class of machines; the system must be configured for each specific computer site.
- SYSGEN 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, 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.**