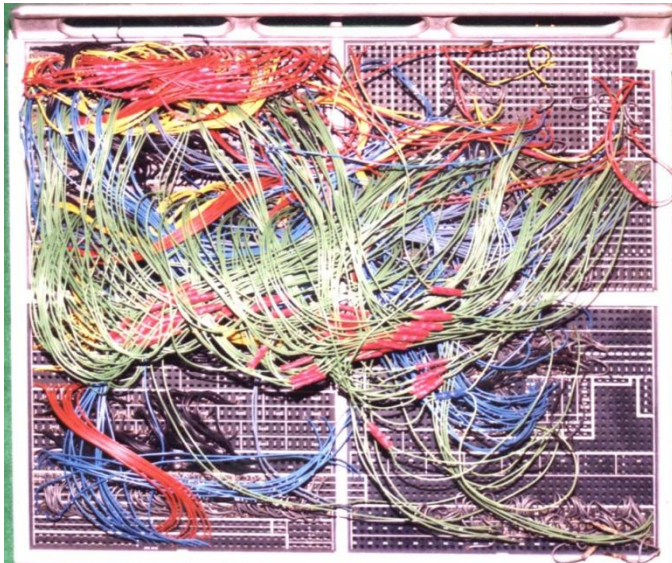
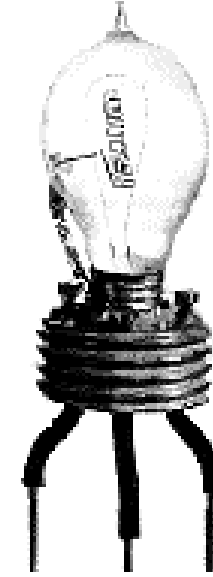


History of Operating Systems

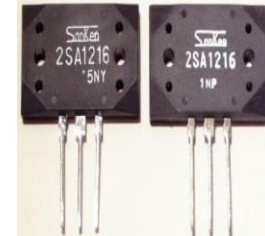
- ❑ First generation 1945 - 1955
 - vacuum tubes, plug boards - *user plugs-in program*



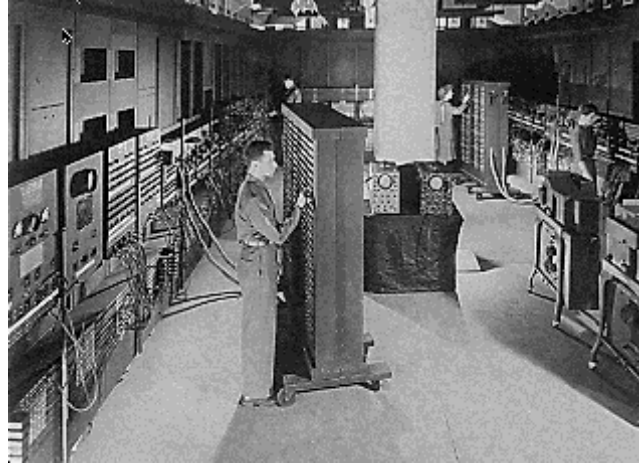
P Per General		STATION NUMBER		DATE		FORTRAN STATEMENT		REMARKS	
00	00	00	00	00	00	00	00	00	00
1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9	9	9
10	10	10	10	10	10	10	10	10	10
11	11	11	11	11	11	11	11	11	11
12	12	12	12	12	12	12	12	12	12
13	13	13	13	13	13	13	13	13	13
14	14	14	14	14	14	14	14	14	14
15	15	15	15	15	15	15	15	15	15
16	16	16	16	16	16	16	16	16	16
17	17	17	17	17	17	17	17	17	17
18	18	18	18	18	18	18	18	18	18
19	19	19	19	19	19	19	19	19	19
20	20	20	20	20	20	20	20	20	20
21	21	21	21	21	21	21	21	21	21
22	22	22	22	22	22	22	22	22	22
23	23	23	23	23	23	23	23	23	23
24	24	24	24	24	24	24	24	24	24
25	25	25	25	25	25	25	25	25	25
26	26	26	26	26	26	26	26	26	26
27	27	27	27	27	27	27	27	27	27
28	28	28	28	28	28	28	28	28	28
29	29	29	29	29	29	29	29	29	29
30	30	30	30	30	30	30	30	30	30
31	31	31	31	31	31	31	31	31	31
32	32	32	32	32	32	32	32	32	32
33	33	33	33	33	33	33	33	33	33
34	34	34	34	34	34	34	34	34	34
35	35	35	35	35	35	35	35	35	35
36	36	36	36	36	36	36	36	36	36
37	37	37	37	37	37	37	37	37	37
38	38	38	38	38	38	38	38	38	38
39	39	39	39	39	39	39	39	39	39
40	40	40	40	40	40	40	40	40	40
41	41	41	41	41	41	41	41	41	41
42	42	42	42	42	42	42	42	42	42
43	43	43	43	43	43	43	43	43	43
44	44	44	44	44	44	44	44	44	44
45	45	45	45	45	45	45	45	45	45
46	46	46	46	46	46	46	46	46	46
47	47	47	47	47	47	47	47	47	47
48	48	48	48	48	48	48	48	48	48
49	49	49	49	49	49	49	49	49	49
50	50	50	50	50	50	50	50	50	50
51	51	51	51	51	51	51	51	51	51
52	52	52	52	52	52	52	52	52	52



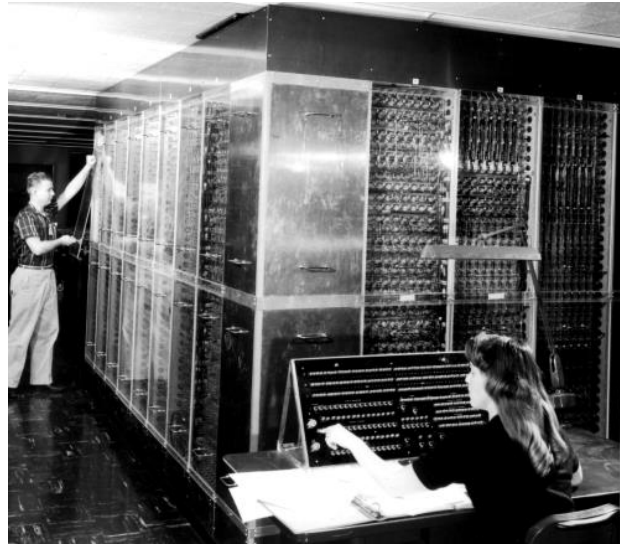
vacuum tube
from the early
1900's



The first computers

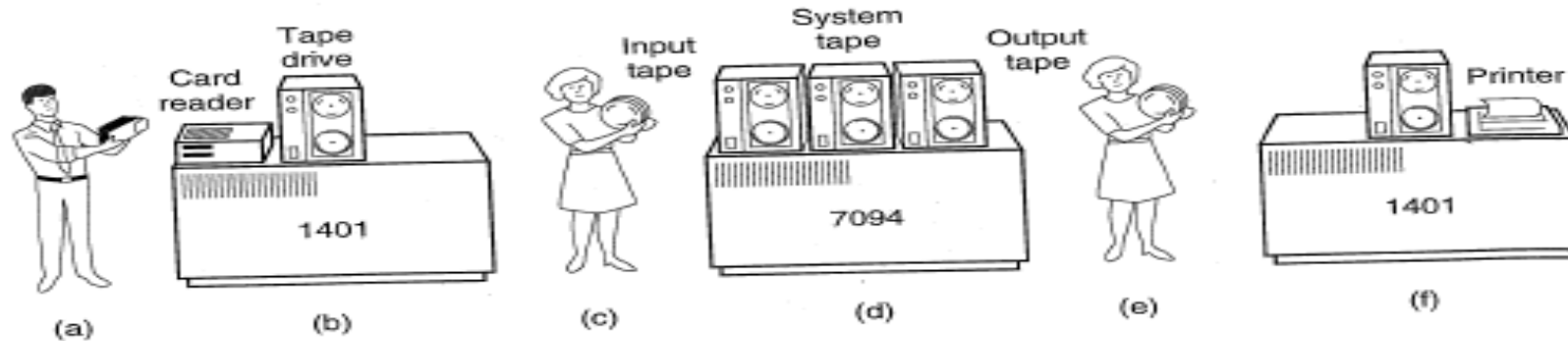


Electronic Numerical Integrator And Computer (ENIAC)



Mathematical Analyzer, Numeric Integrator And Computer(MANIAC)

History of Operating Systems (cont'd)



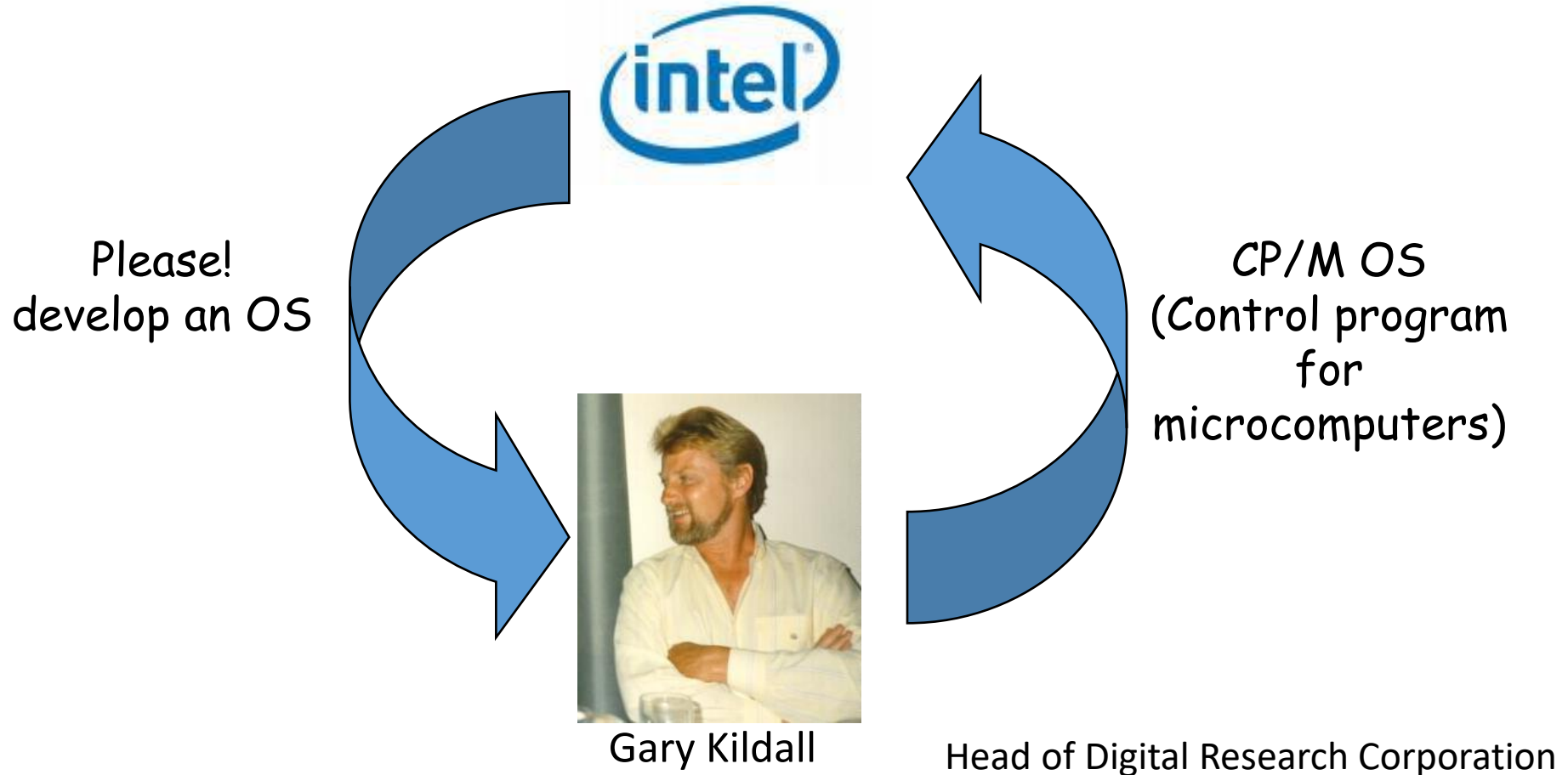
- ❑ Second generation 1955 - 1965
 - transistors, batch systems - *multiple programs on Disk*
- ❑ Third generation 1965 - 1980
 - ICs and multiprogramming - *user interaction (time-sharing)*
- ❑ Fourth generation 1980 - present
 - personal computers - *graphic user-interface*
 - Networks - *file & computing services*
 - Web-computing, *Handheld devices , Cellular phones, Cloud computing...*

Bill Gates



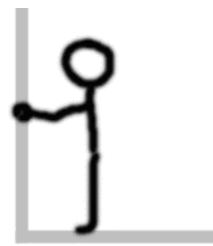
How Bill Gates became rich...

1974: Intel releases the 8080 processor, needs an OS



How Bill Gates became rich...(cont'd)

1974: Intel releases the 8080 processor, needs an OS



CP/M OS

Sure!

Can you grant me
CP/M rights?



Gary Kildall

How Bill Gates became rich...(cont'd)

1980: IBM designs IBM PC, needs an OS

Can you find an
OS for our PC?



IBM

Sorry, too busy!!!!!!



DIGITAL
RESEARCH



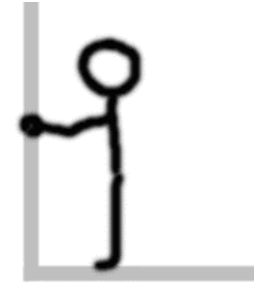
Gary Kildall

Please meet IBM, they need an OS

How Bill Gates became rich...(cont'd)

1980: IBM designs IBM PC, needs an OS

Kildall too busy.
Please develop an
OS!



I'de like to buy the DOS OS



Sure, it's yours for \$75,000

How Bill Gates became rich...(cont'd)

1980: IBM designs IBM PC, needs an OS

May I retain the
rights for MS-
DOS?



Sure, why not!!



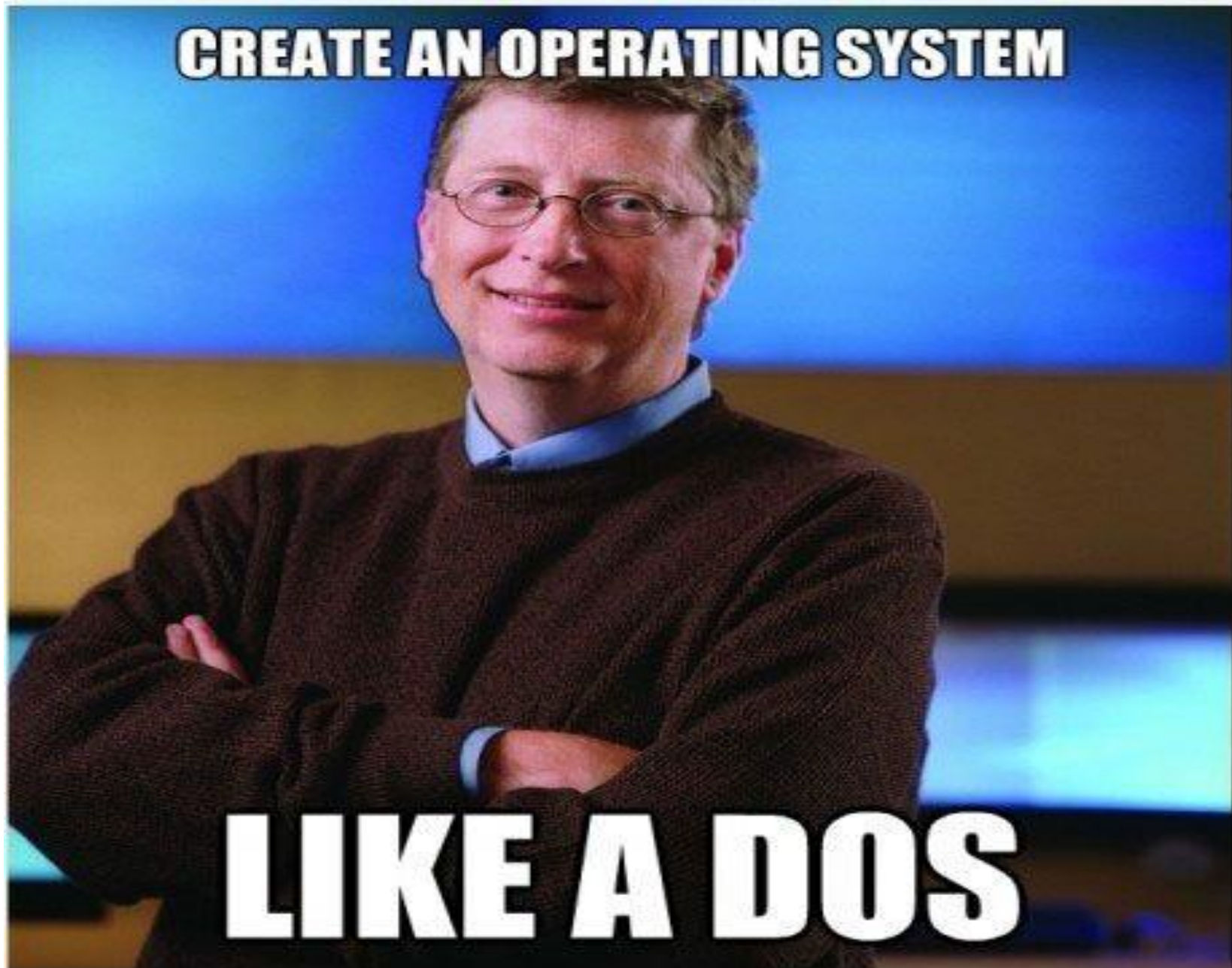
How Bill Gates became rich...(cont'd)



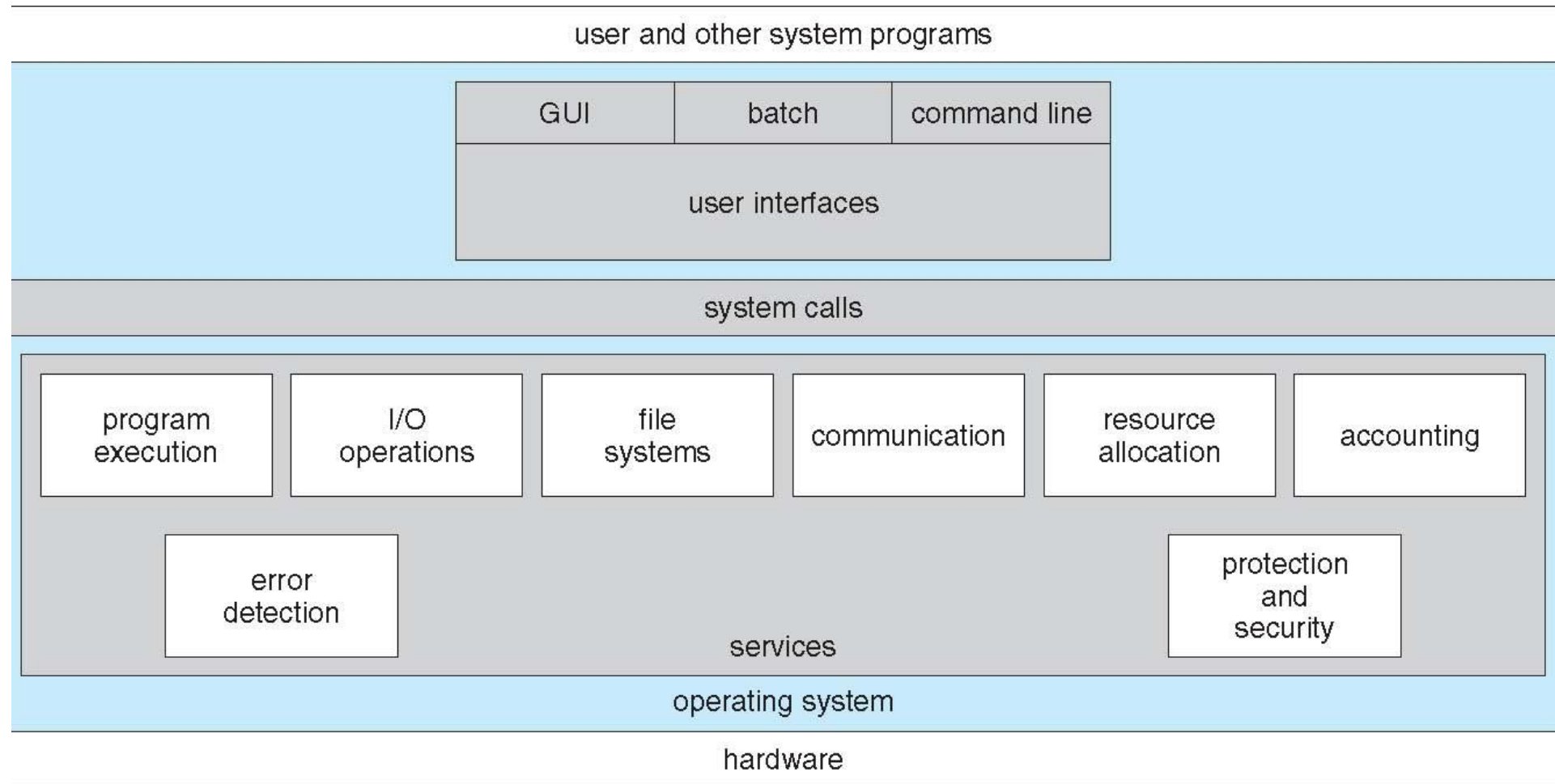
Net worth 97.4 Billion \$



CREATE AN OPERATING SYSTEM



OPERATING SYSTEM SERVICES



OPERATING SYSTEM SERVICES-I

- 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 (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-II

- **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 to ensure correct and consistent computing
 - □ Debugging facilities can greatly enhance the user's and
 - programmer's abilities to efficiently use the system

OPERATING SYSTEM SERVICES-III

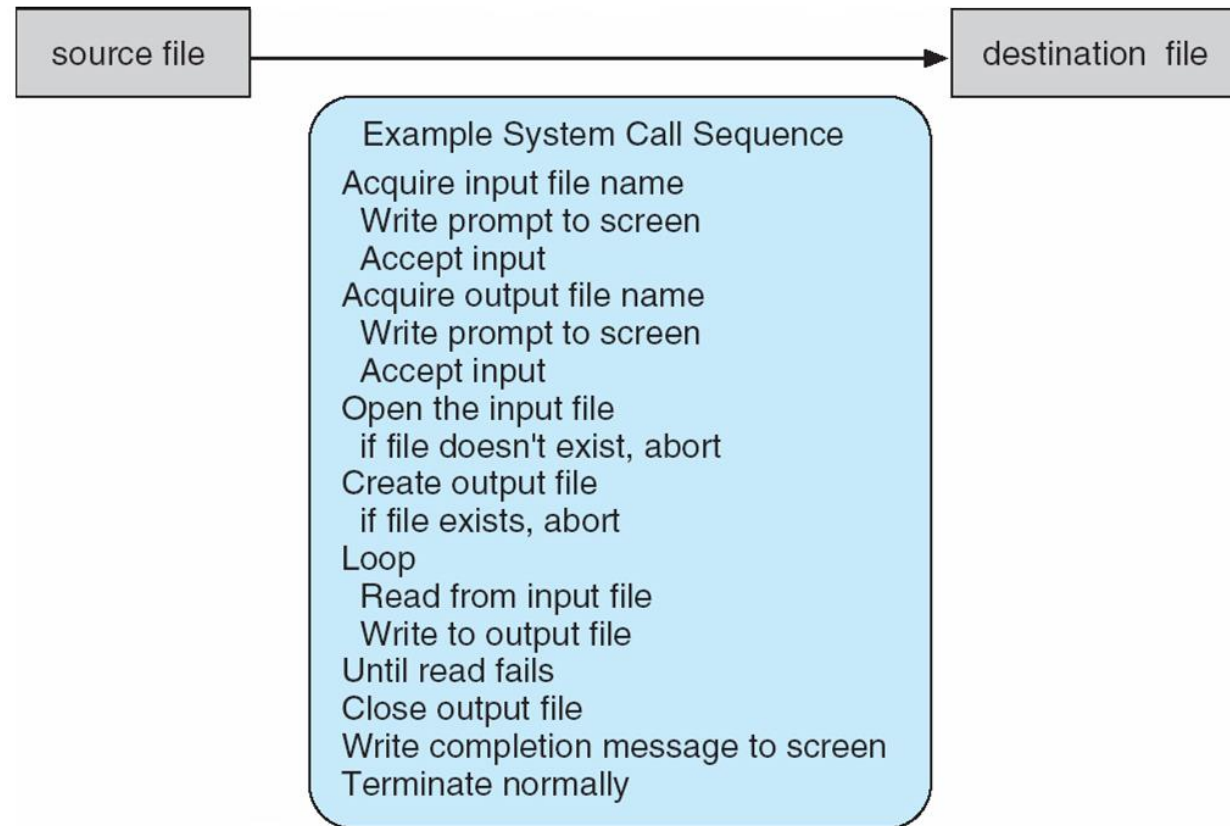
- **Another set** of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
 - **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
 - ☐ Many types of resources -CPU cycles, main memory, 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
 - ☐ **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

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)

Example of System Calls

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



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:

<pre>#include <unistd.h></pre>		
<pre>ssize_t</pre>	<pre>read(int fd, void *buf, size_t count)</pre>	
<div></div>	<div></div>	<div></div>
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:

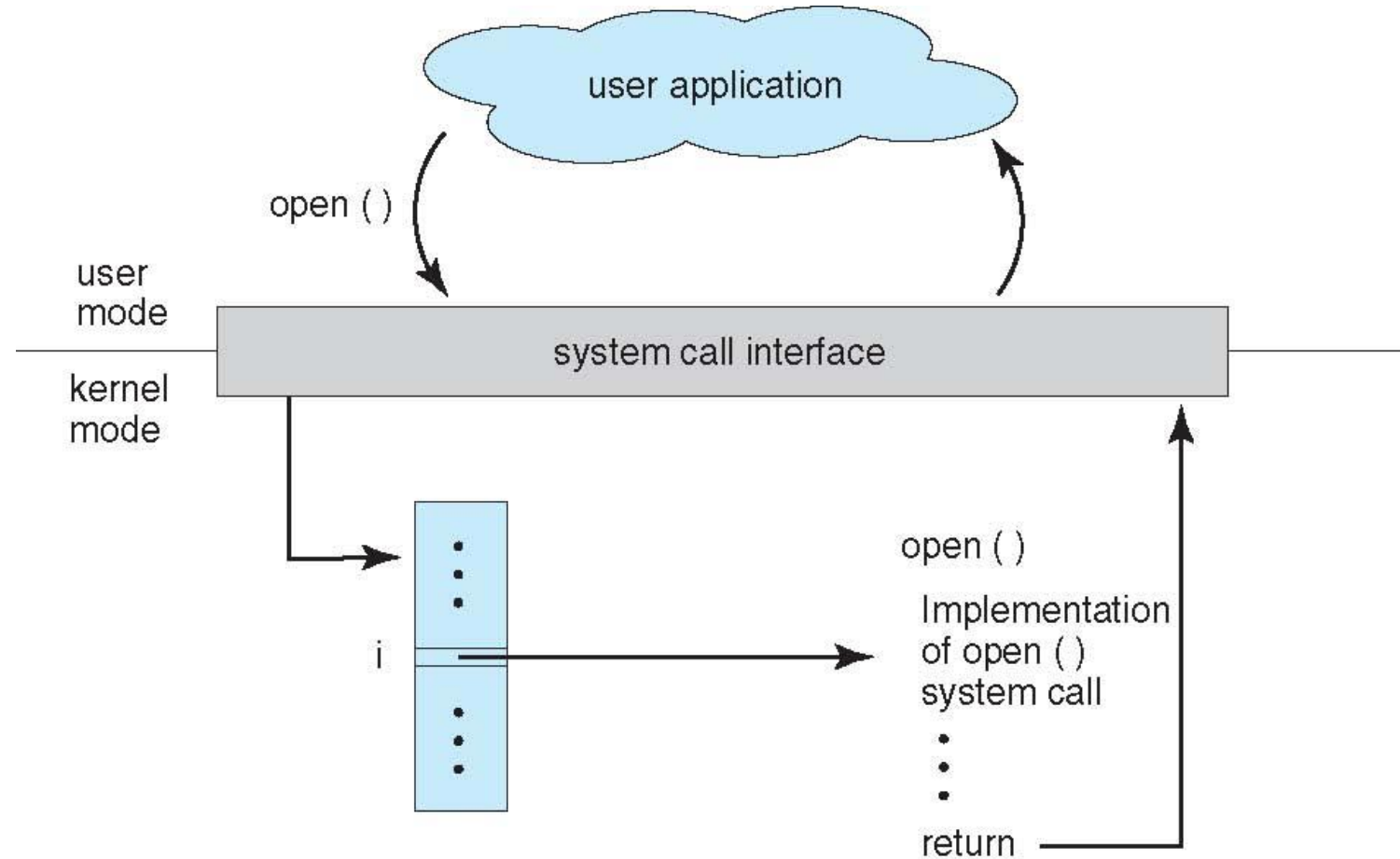
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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
 - Managed by run-time support library (set of functions built into libraries included with compiler)

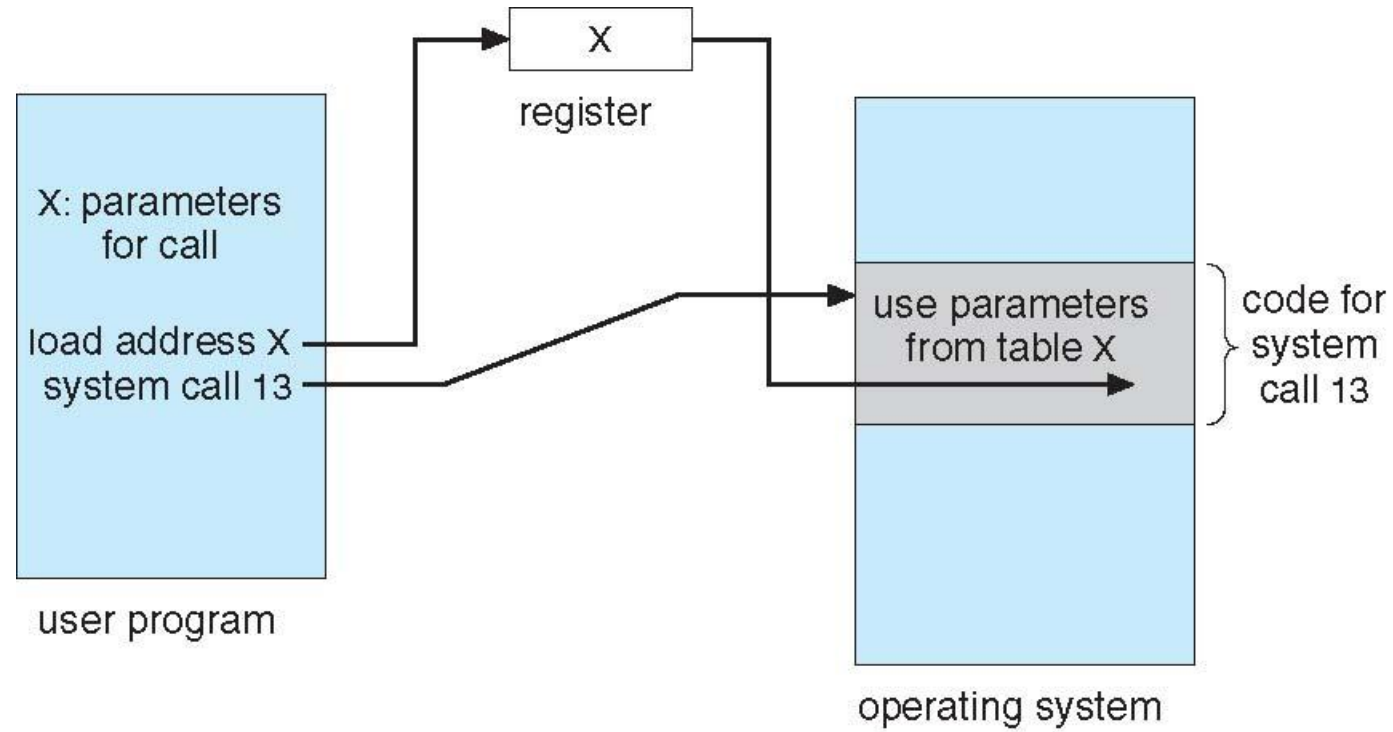
API - System Call - OS Relationship



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 used to pass parameters to the OS
 - Simplest: pass the parameters in registers
 - In some cases, may be more parameters than registers
 - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
 - This approach taken by Linux and Solaris
 - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
 - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



Types of System Calls

- **Process control**
 - ✓ 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** 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.)

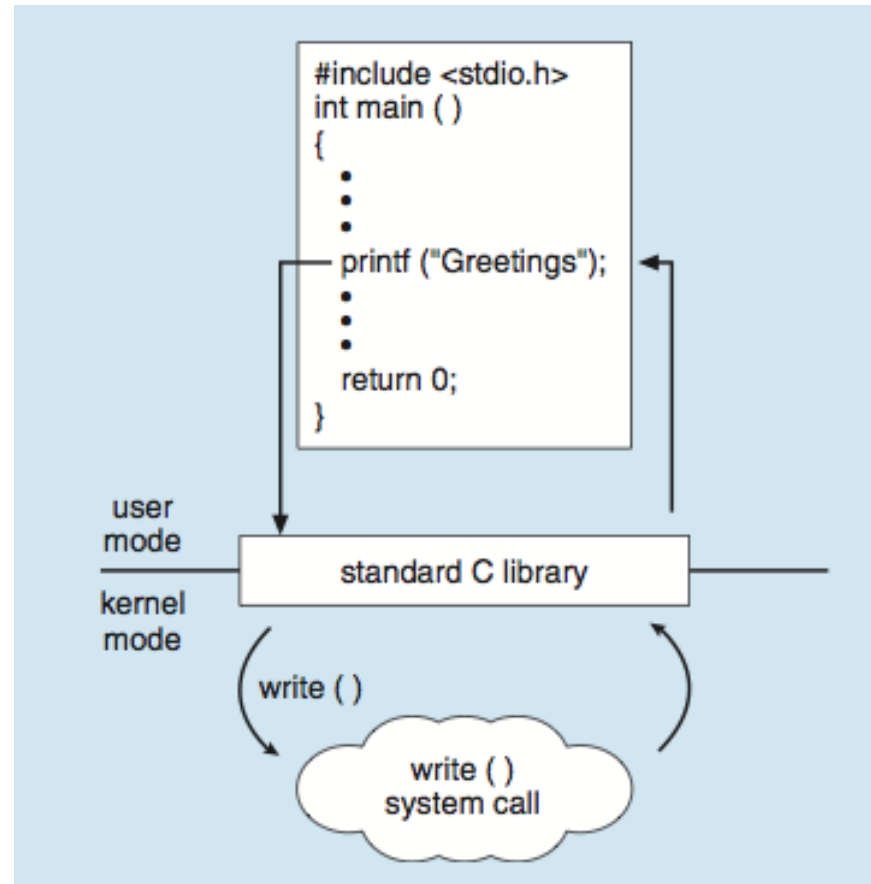
- Protection
 - ✓ Control access to resources
 - ✓ Get and set permissions
 - ✓ Allow and deny user access

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() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

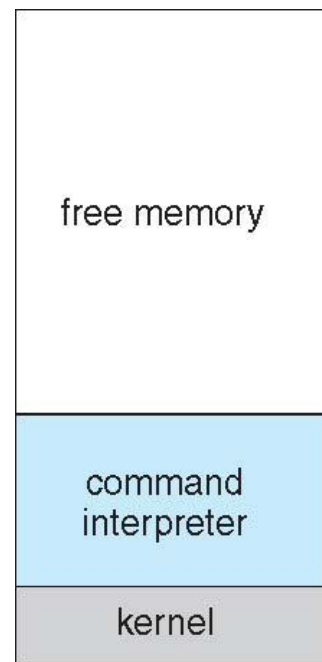
Standard C Library Example

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



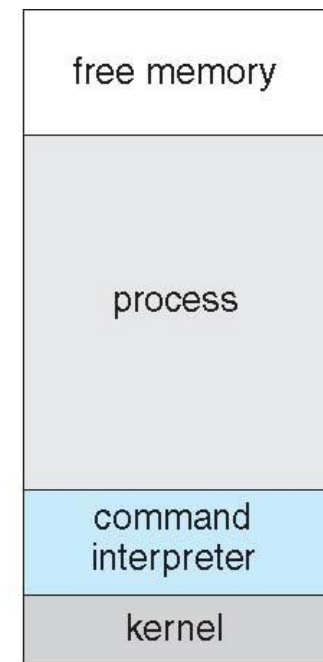
Example: MS-DOS

- 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



(a)

At system startup



(b)

running a program

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

- 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 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**, **subsystems**, **daemons**

- **Application programs**

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

Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- 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

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

Implementation

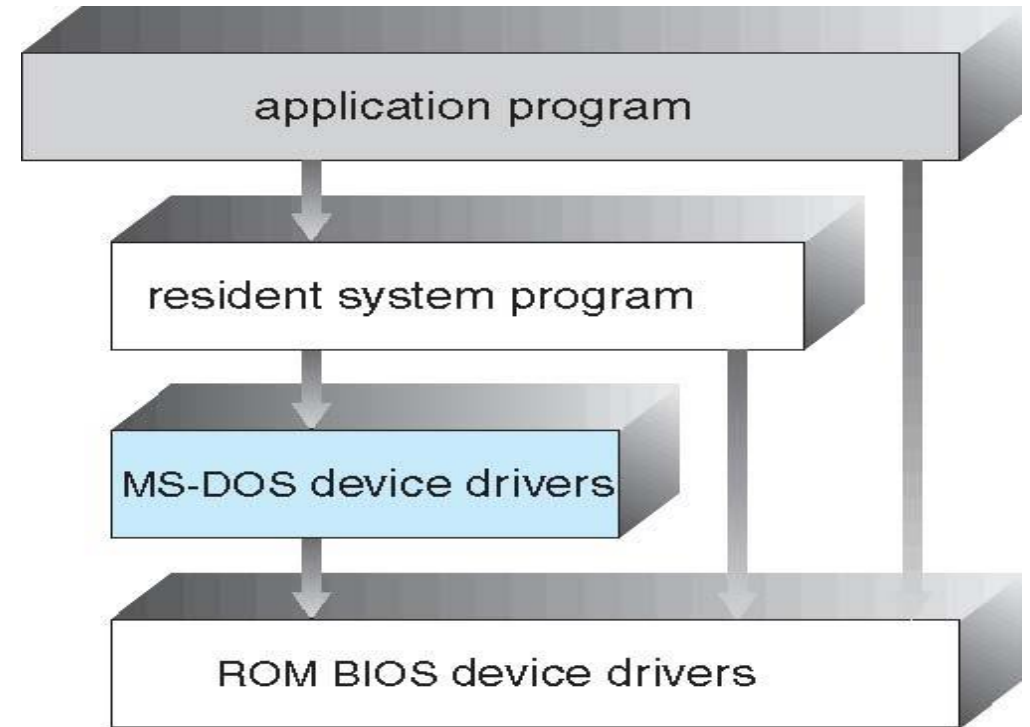
- Much variation
 - Early OS'es in assembly language
 - Then system programming languages like Algol, PL/1
 - Now C, C++
- Actually usually a mix of languages
 - Lowest levels in assembly
 - Main body in C
 - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level 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 very large program
- Various ways to structure ones
 - Simple structure – MS-DOS
 - More complex -- UNIX
 - Layered – an abstraction
 - 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



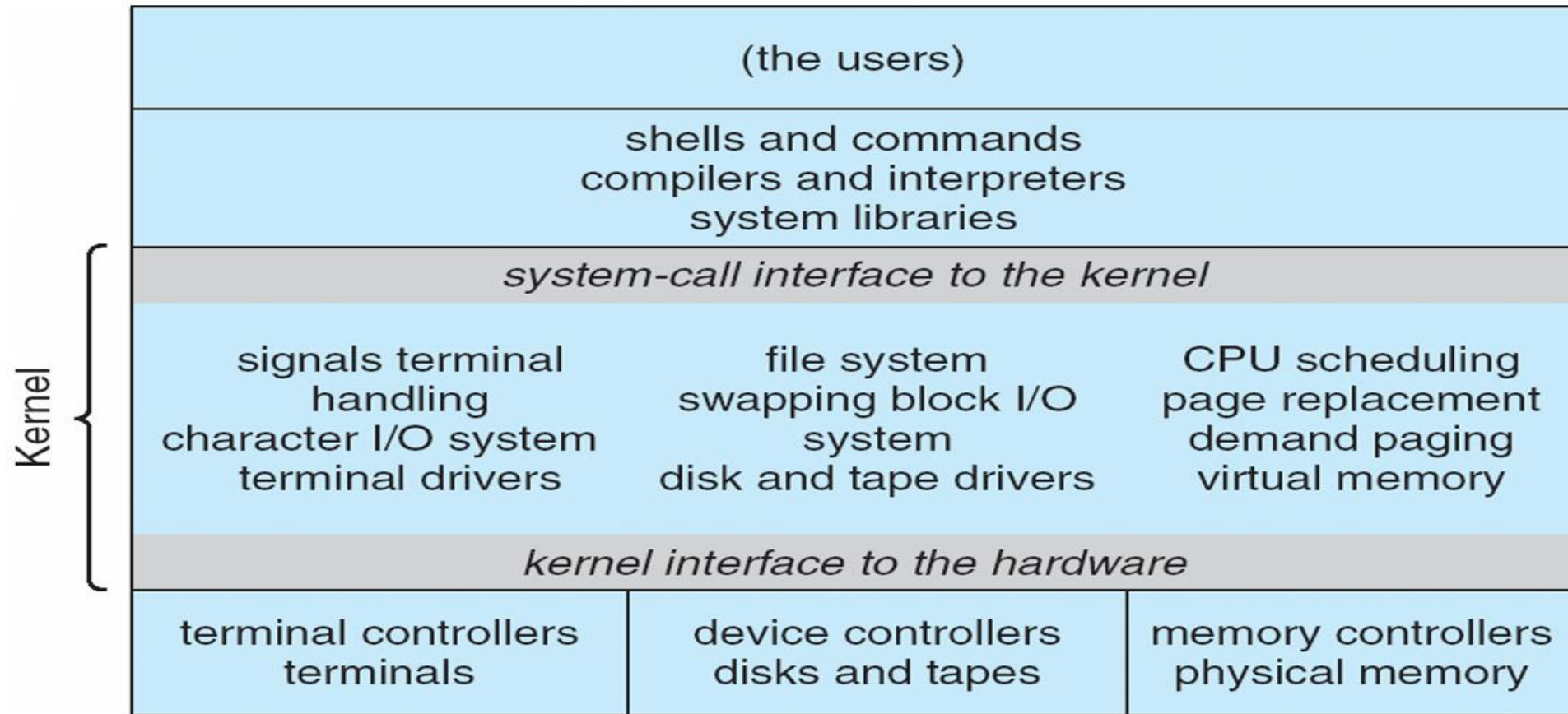
Non Simple Structure -- UNIX

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

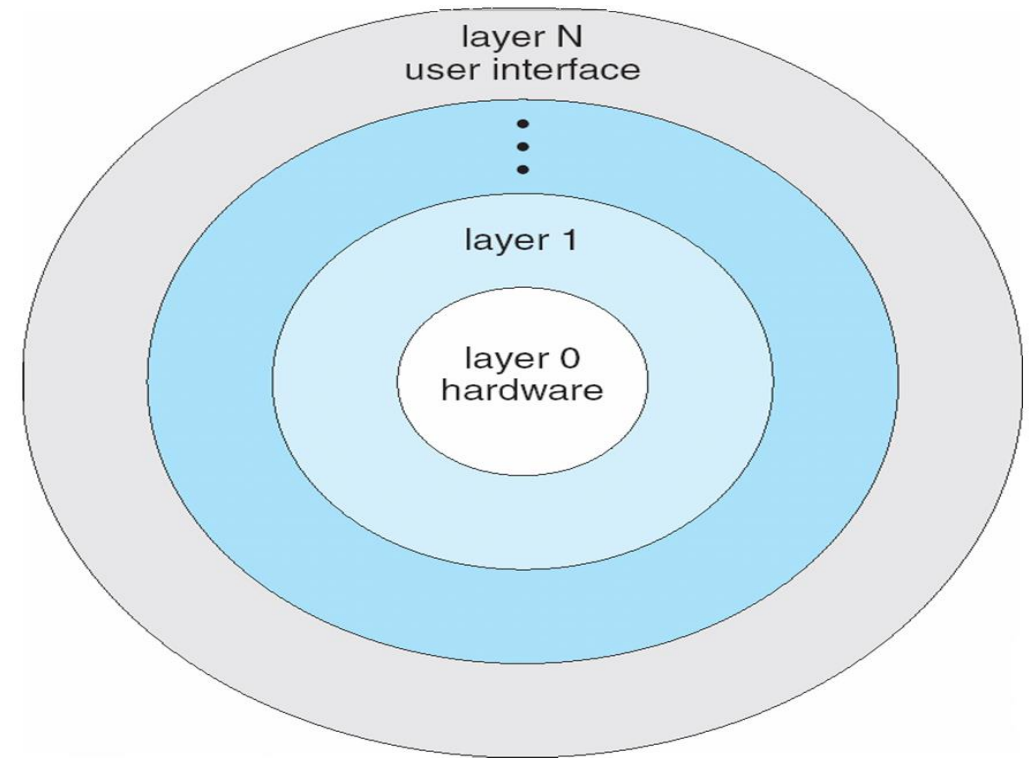
Traditional UNIX System Structure

Beyond simple but not fully layered



Layered Approach

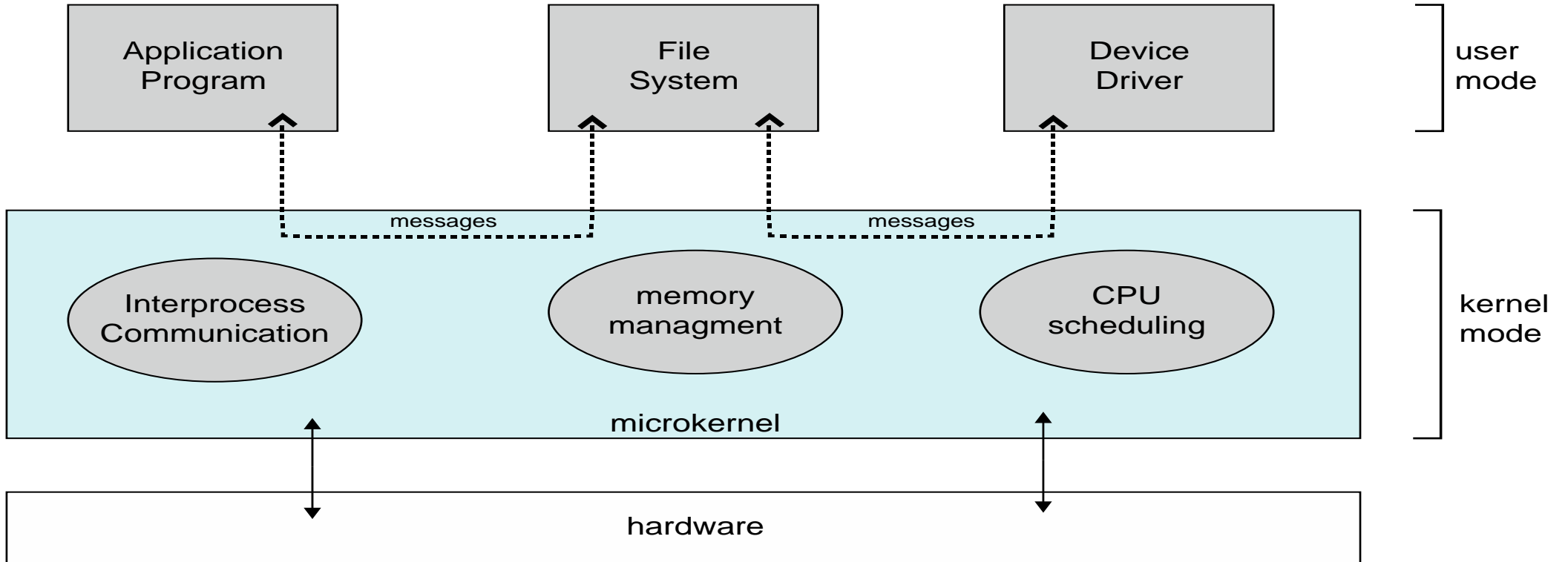
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level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

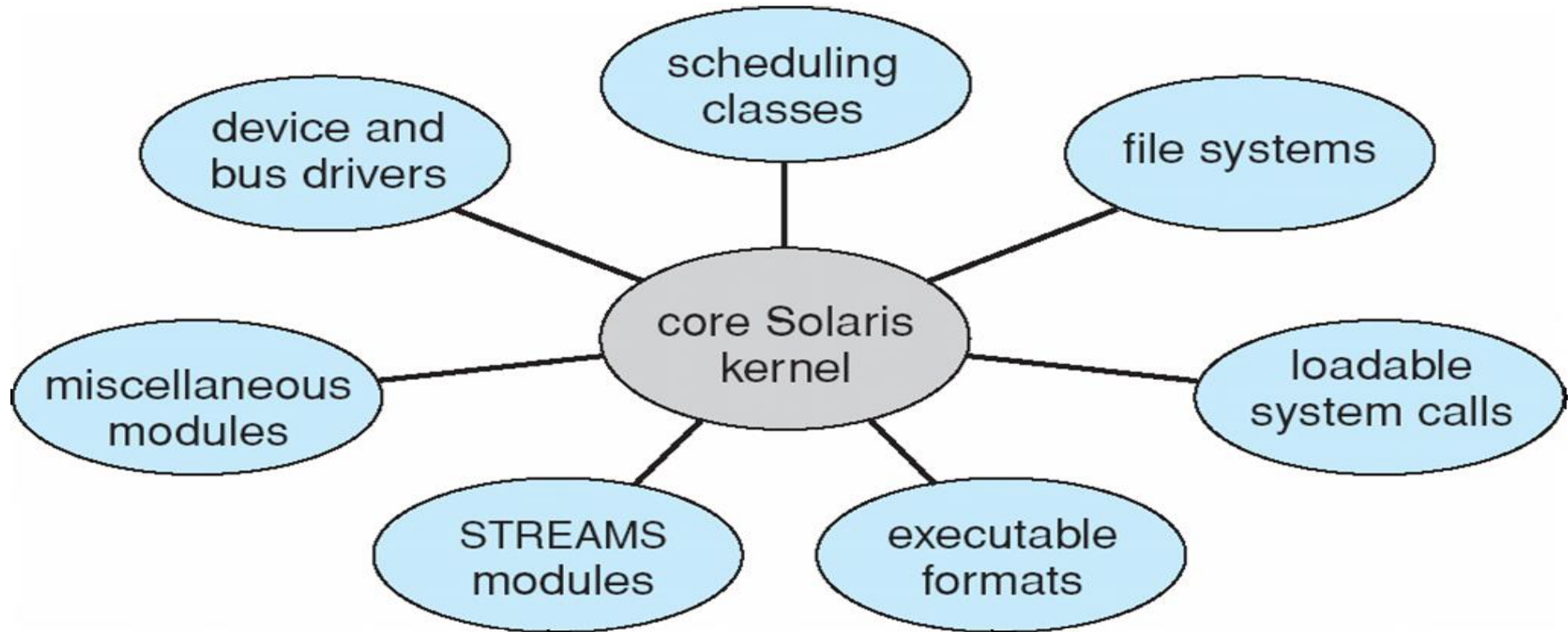
Microkernel System Structure



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

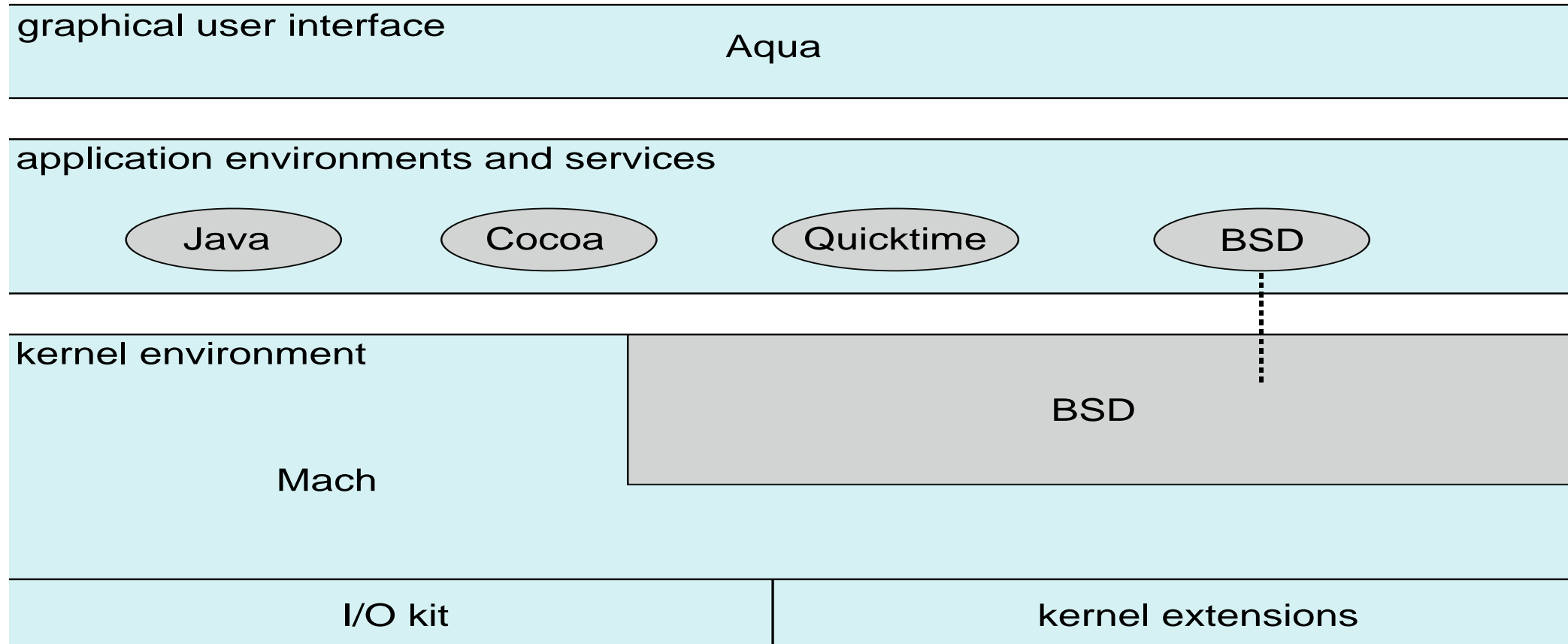
Solaris Modular Approach



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



iOS

- Apple mobile OS for *iPhone*, *iPad*
 - Structured on Mac OS X, added functionality
 - Does not run OS X applications natively
 - 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

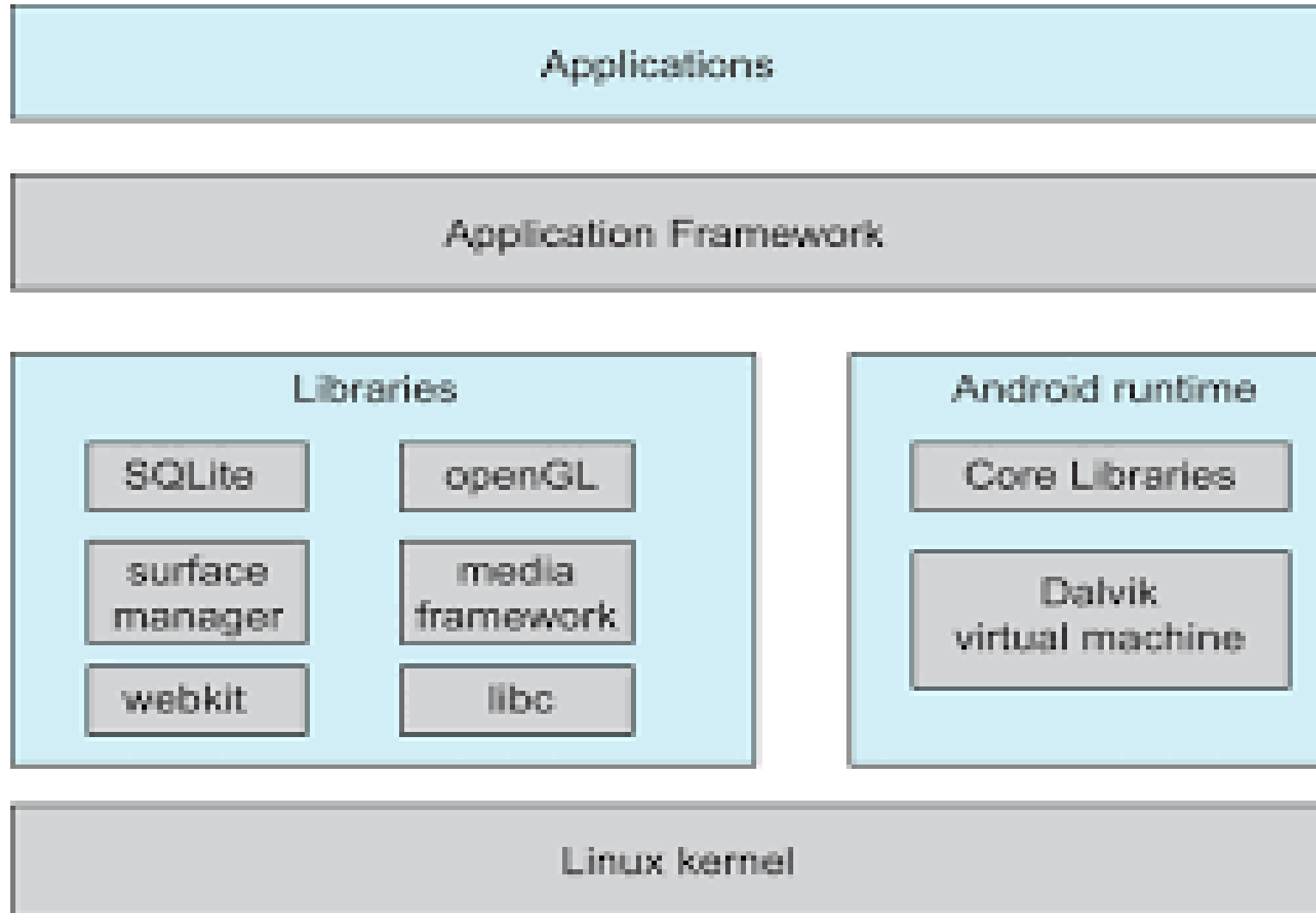
Core Services

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
 - Java class files compiled to Java bytecode then translated to executable then runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

Android Architecture



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**

END OF LECTURE
THANK YOU