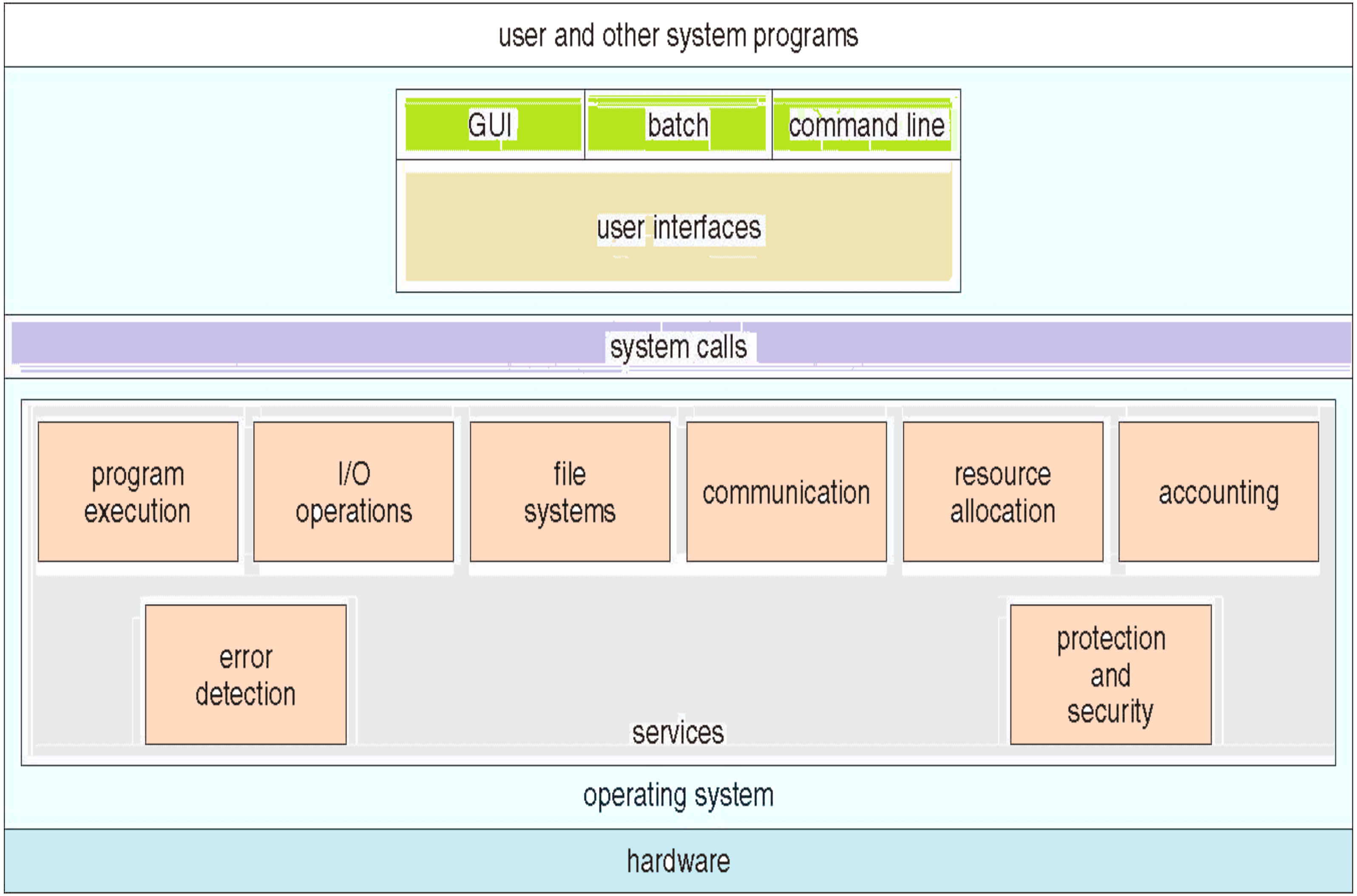
Chapter 2: System Structures

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

A View of OS Services



Operating System Services (1)

- **Operating systems** provide an **environment** for execution of programs and services
- **Operating-system services** provides functions that are helpful to **the user**:
  - **User interface** - Almost all OS 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**
  - **File-system manipulation** - Obviously, **programs** need to read and write files and directories, create and delete them, search them, list file Information, permission management.

Operating System Services (2)

- **Operating-system services** provides functions that are helpful to **the user**:
  - **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 aware of possible errors
    - **Error** 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 (3)

- 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
    - **Some** (e.g. CPU, main memory, and file storage) have special allocation code. **Others** (e.g. I/O devices) have general request and release code
  - **Accounting -** To keep track of which users use **how much** and **what kinds** of computer resources
  - **Protection and security -** The owners of information may want to control use of that information, **Concurrent processes** should not interfere with each other
    - **Protection:** ensuring that all **access** to system resources is **controlled**
    - **Security: require user authentication**, **defend** external I/O devices from **invalid access attempts**
    - If a system is to be protected and secure, precautions must be instituted throughout it. **A chain is only as strong as its weakest link**.

User-OS Interface -- CLI

Command Line Interface (**CLI**) or **command interpreter** allows direct command entry

- Sometimes implemented in **kernel**, sometimes by **systems program**
- Sometimes **multiple flavors** implemented – **Bourne shell**, **C shell**, **Bash**, **Korn shell**
- Primarily **fetches** a **command** from **user** and **executes** it
  - Sometimes commands built-in (内部命令), sometimes just names of programs (外部命令)
  - If the latter, adding **new features** doesn't require **shell modification**

Bash Command Interpreter

```
studm@speech9: /home/studm
[studm@speech9 ~]$
[studm@speech9 ~]$ su root
Password:
[root@speech9 studm]# uname
Linux
[root@speech9 studm]# w
 18:15:02 up 12 days,  1:27,  1 user,  load average: 0.21, 0.09, 0.03
USER     TTY      LOGIN@   IDLE   JCPU   PCPU WHAT
studm    pts/0    18:05    0.00s  0.05s  0.01s sshd: studm [priv]
[root@speech9 studm]#
[root@speech9 studm]# host speech9.csie.ntust.edu.tw
speech9.csie.ntust.edu.tw has address 140.118.175.19
[root@speech9 studm]#
[root@speech9 studm]# traceroute -n 140.118.125.29
traceroute to 140.118.125.29 (140.118.125.29), 30 hops max, 40 byte packets
 1  140.118.125.254  0.890 ms  1.143 ms  1.416 ms
 2  140.118.125.28  0.490 ms  0.496 ms  0.490 ms
 3  140.118.125.29  0.817 ms  0.814 ms  0.807 ms
[root@speech9 studm]#
[root@speech9 studm]# ps
  PID TTY          TIME CMD
23168 pts/0    00:00:00 su
23170 pts/0    00:00:00 bash
23241 pts/0    00:00:00 ps
[root@speech9 studm]#
[root@speech9 studm]# ps awx | grep smbd
 3223 ?        Ss     0:01 smbd -D
 3241 ?        S      0:00 smbd -D
15539 ?        S      0:14 smbd -D
18850 ?        S      0:00 smbd -D
[root@speech9 studm]#
[root@speech9 studm]#
```

User-OS Interface -- GUI

- User-friendly **graphic user interface** **(**desktop metaphor)
  - Usually **mouse**, **keyboard**, and **monitor**
  - **Icons** (in **window**) represent **files**, **programs**, **actions**, etc
  - **Various mouse buttons** over **objects** in the interface cause **various actions** (e.g. provide information, execute function, open folder)
  - Invented at **Xerox PARC, 1973**

- Many systems now include both **CLI** and **GUI** interfaces
  - **Microsoft Windows** is GUI with CLI “command” shell
  - **Apple Mac OS X** as 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** not possible or not desired
  - Actions and selection based on **gestures**
  - **Virtual keyboard** for text entry

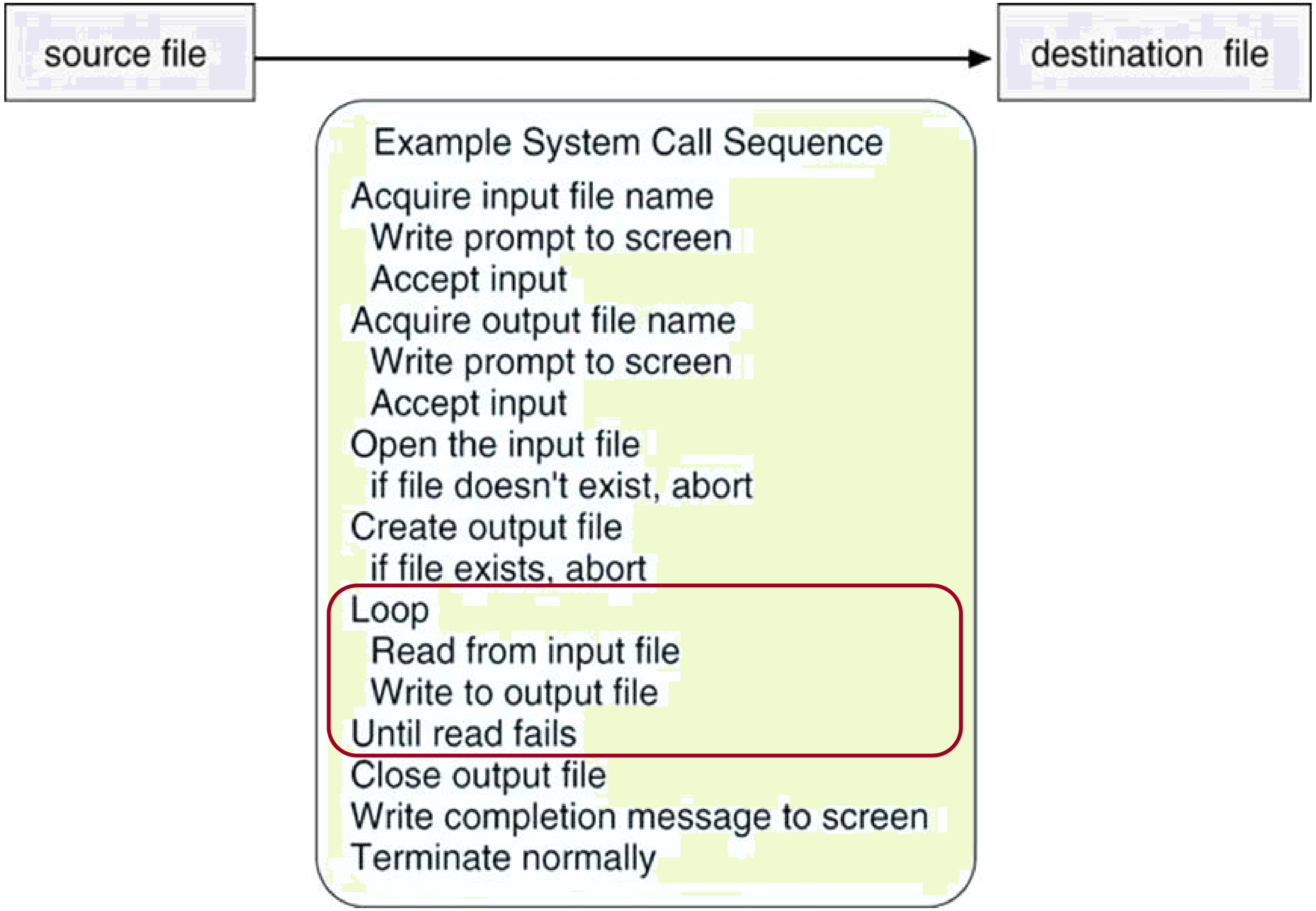
System Calls

- **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 Program Interface** (**API**) rather than directly using **system calls**
- Three most common **APIs** are
  - **Win32 API** for **MS Windows**,
  - **POSIX API** for POSIX-based systems (virtually all versions of **UNIX**, **Linux**, and **Mac OS X**)
  - **Java API** for the **Java virtual machine** (**JVM**)
- Why use **APIs** rather than system calls?

(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

- Consider the **read( )** function that is available in **UNIX** and **Linux** systems

```
#include <unistd.h>

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

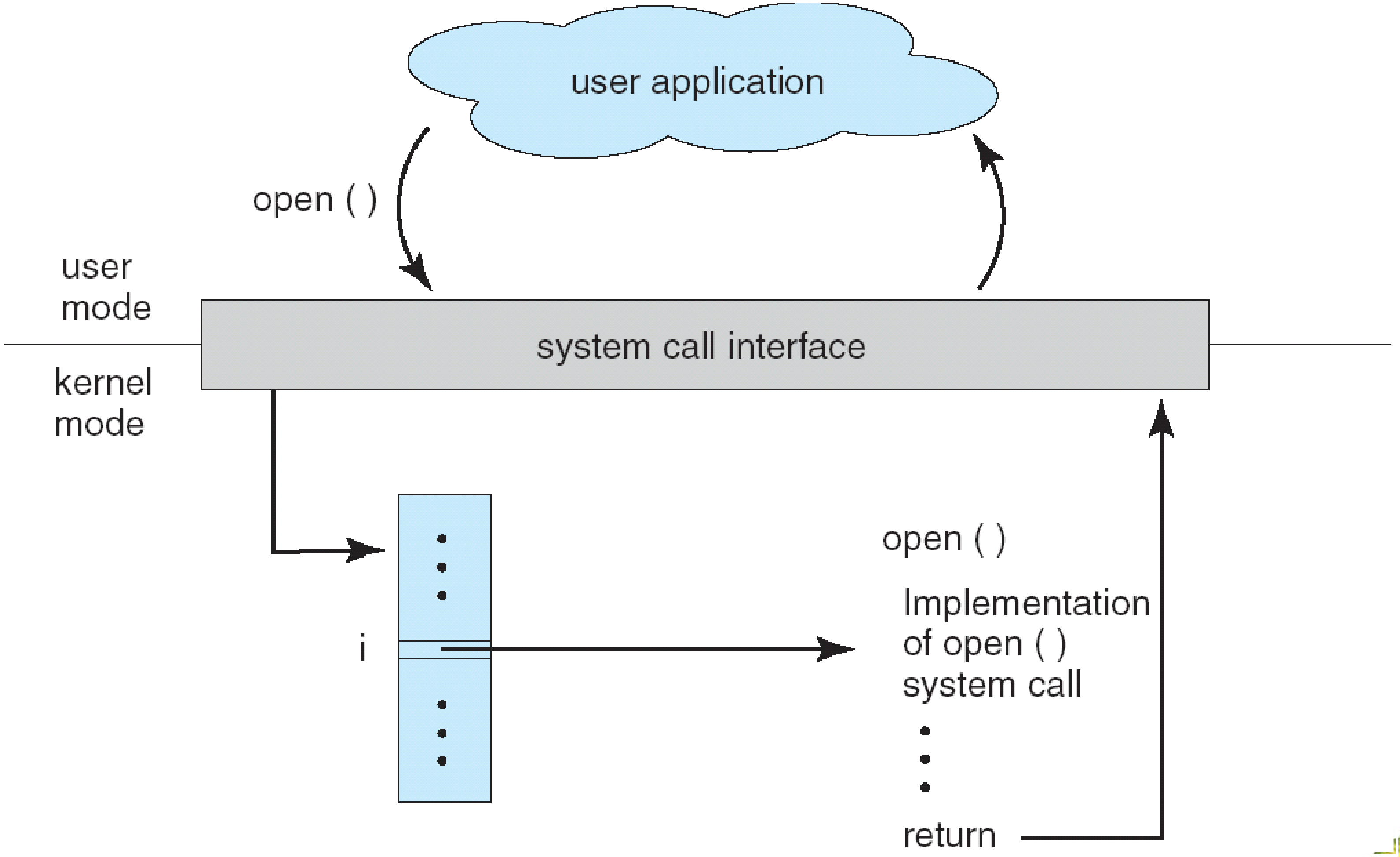
return value	function name	parameters
`ssize_t`	`read`	`(int fd, void *buf, size_t count)`

- The parameters passed to **read ( )**
  - **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. If end of file, return 0. If an error occurs, return -1.

System Call Implementation

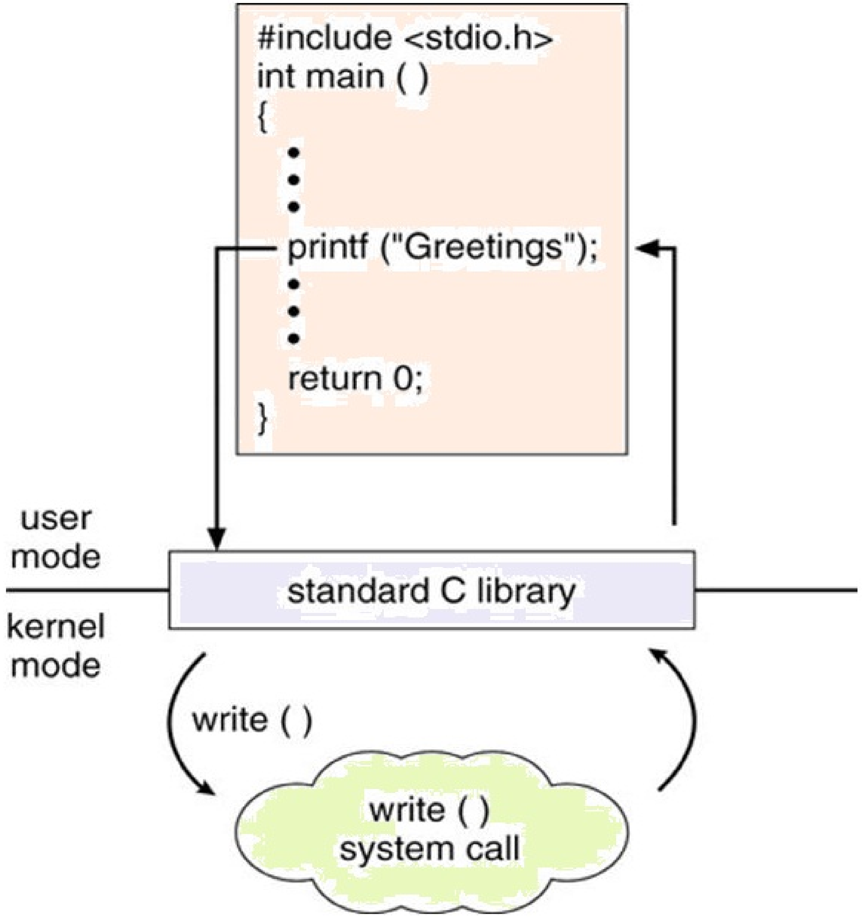
- Typically, a **number** is associated with each **system call**
  - **System-call interface** maintains a **table indexed** according to these numbers
- The **system call interface**
  - invokes **intended system call** in **OS kernel**
  - returns **status** of the system call and any **return values**
- **How the system calls are implemented?**
  - The **caller** need know nothing about it
  - Just obey **API** and understand what **OS** will do as a result call
  - hidden from programmer by **API** (most details of **OS interface**)
    - set of functions built into **libraries** included with **compiler**

API – System Call – OS Relationship



Standard C Library Example

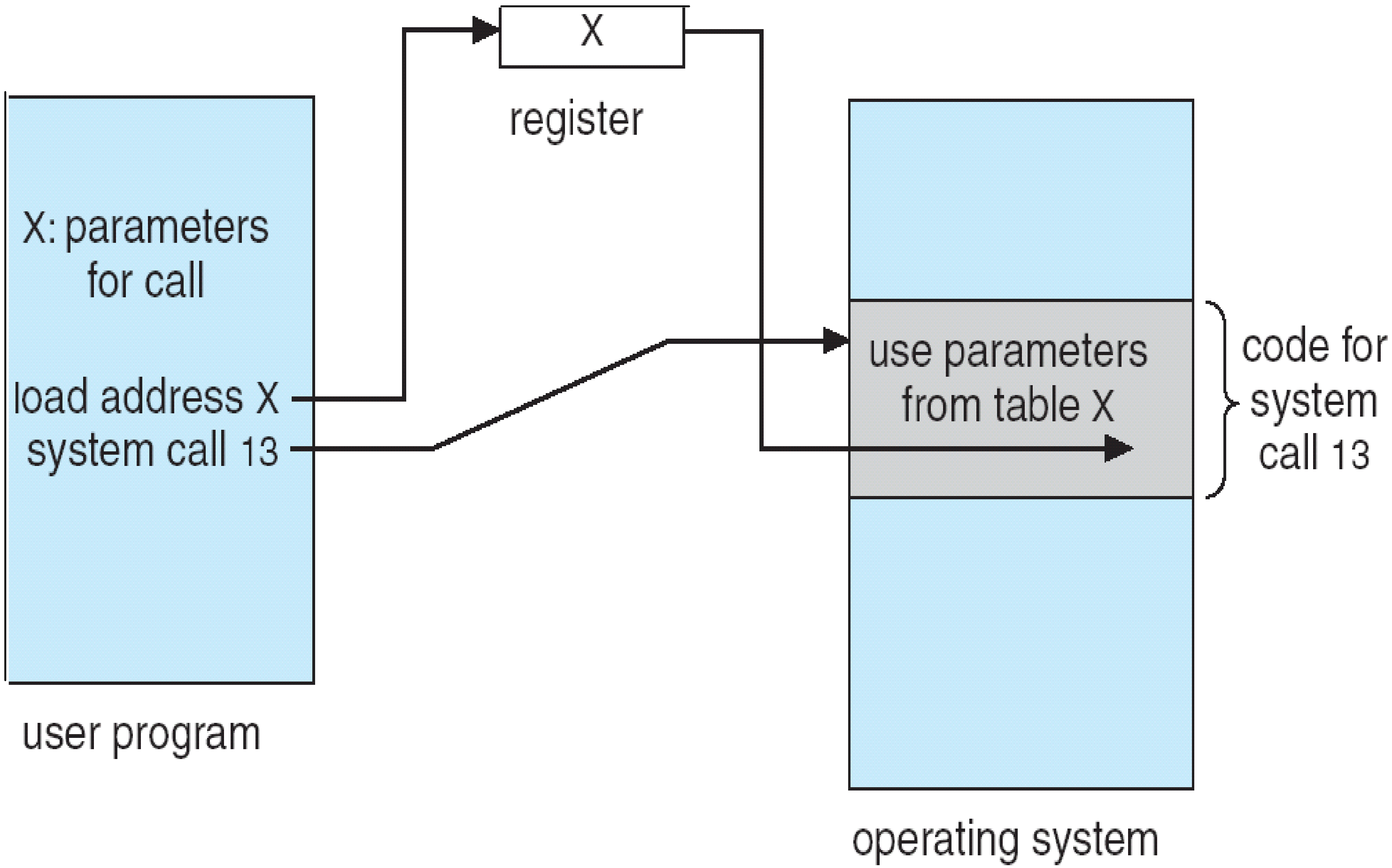
- **C program** invoking **printf()** library call, which calls **write() system call**



System Call Parameter Passing

- Often, more information is required than simply **identity of 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

Parameter Passing via Table



Classes of System Calls

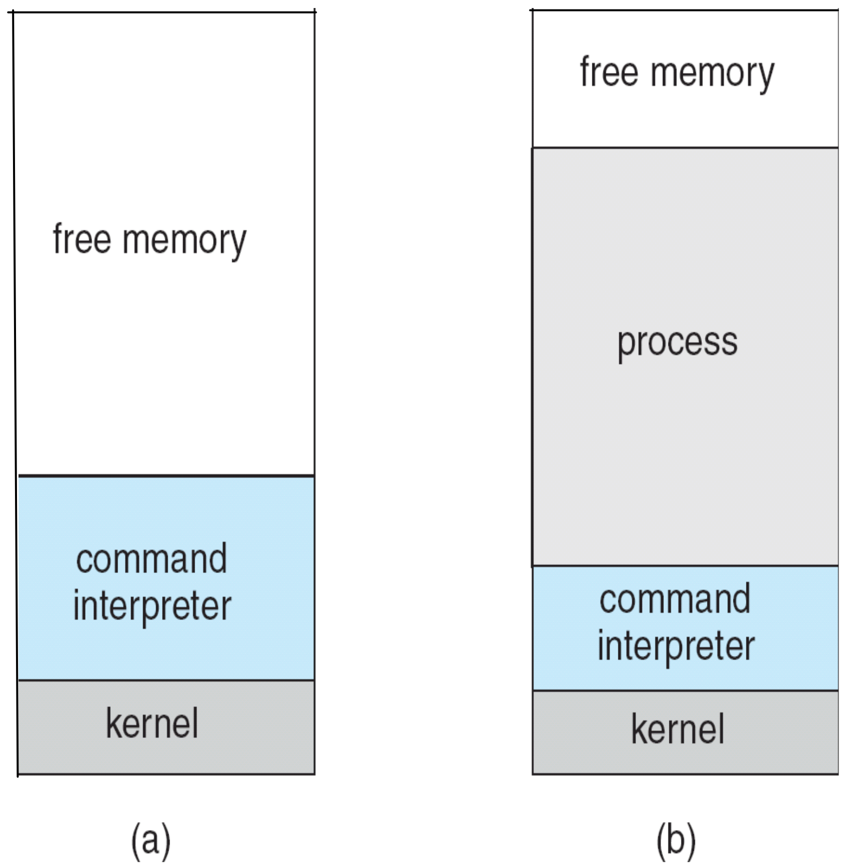
- **Process control**
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
- **File management**
  - create file, delete file
  - **open**, **close** file
  - read, write, reposition device management
- **Device management**
  - request device, release device
  - read, write, reposition
  - **get** device attributes, **set** device attributes
- **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**
  - **Message passing model:** send, receive messages
  - **Shared-memory model:** create and gain access to memory regions
- **Protections**
  - **Control** access to resources
  - **Get** and **set permissions**
  - Allow and deny user access

Examples of Windows & Unix System Call

	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

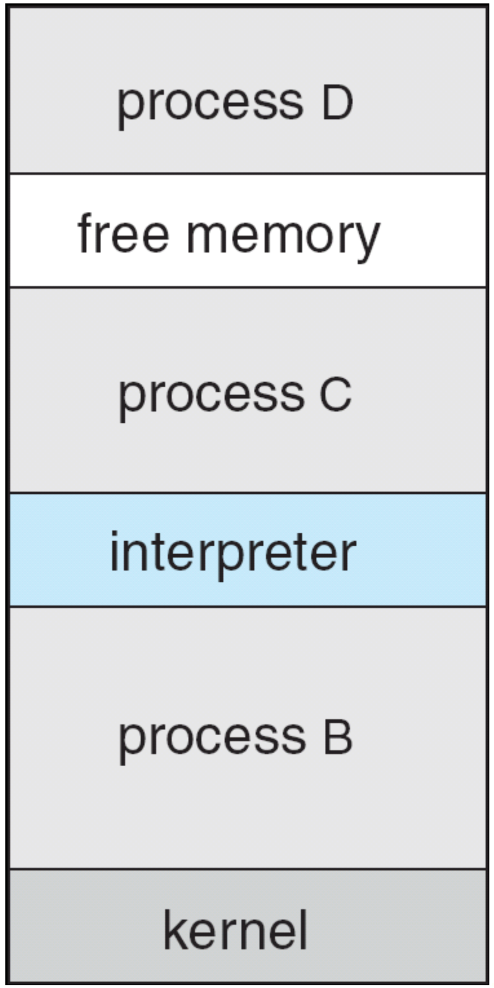
- **Single tasking**
- **Single memory space**
- **Shell** invoked when **system booted**
  - **Loads program** into **memory**, overwriting all **but** the **kernel**
  - Program exit -> shell reloaded



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

Example: FreeBSD

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



OS Design and Implementation (1)

- **Internal structure** of different Operating Systems can vary widely
- Start by defining **goals** and **specifications**
- **Affected** by choice of
  - **Hardware**,
  - **Type of system: batch, time sharing, multiuser, real time**
  - ***User* goals** and ***System* goals**
    - **User goals** – OS should be **convenient to use**, **easy to learn**, **reliable**, **safe**, and **fast**
    - **System goals** – OS should be easy to **design**, **implement**, and **maintain**, as well as **flexible, reliable, error-free**, and **efficient**

OS Design and Implementation (2)

- **Important principle** to separate

  Policy: What will be done? (e.g. time slice width)

  Mechanism: How to do it? (e.g. timer interrupt)

- **Mechanisms** determine how to do something
- **Policies** decide what will be done
- It allows maximum flexibility if policy decisions are to be changed later

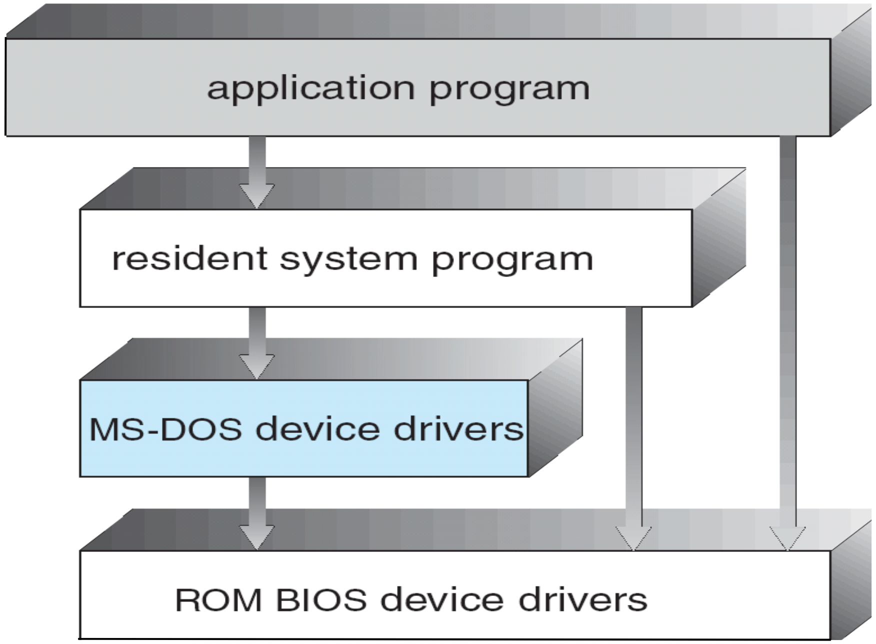
OS Design and Implementation (3)

- Much variation in language
  - 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**

Operating System Structure

- **Simple Structure**
  - **MS-DOS** written to provide the **most functionality** in the **least space**
  - **Not divided** into **modules**

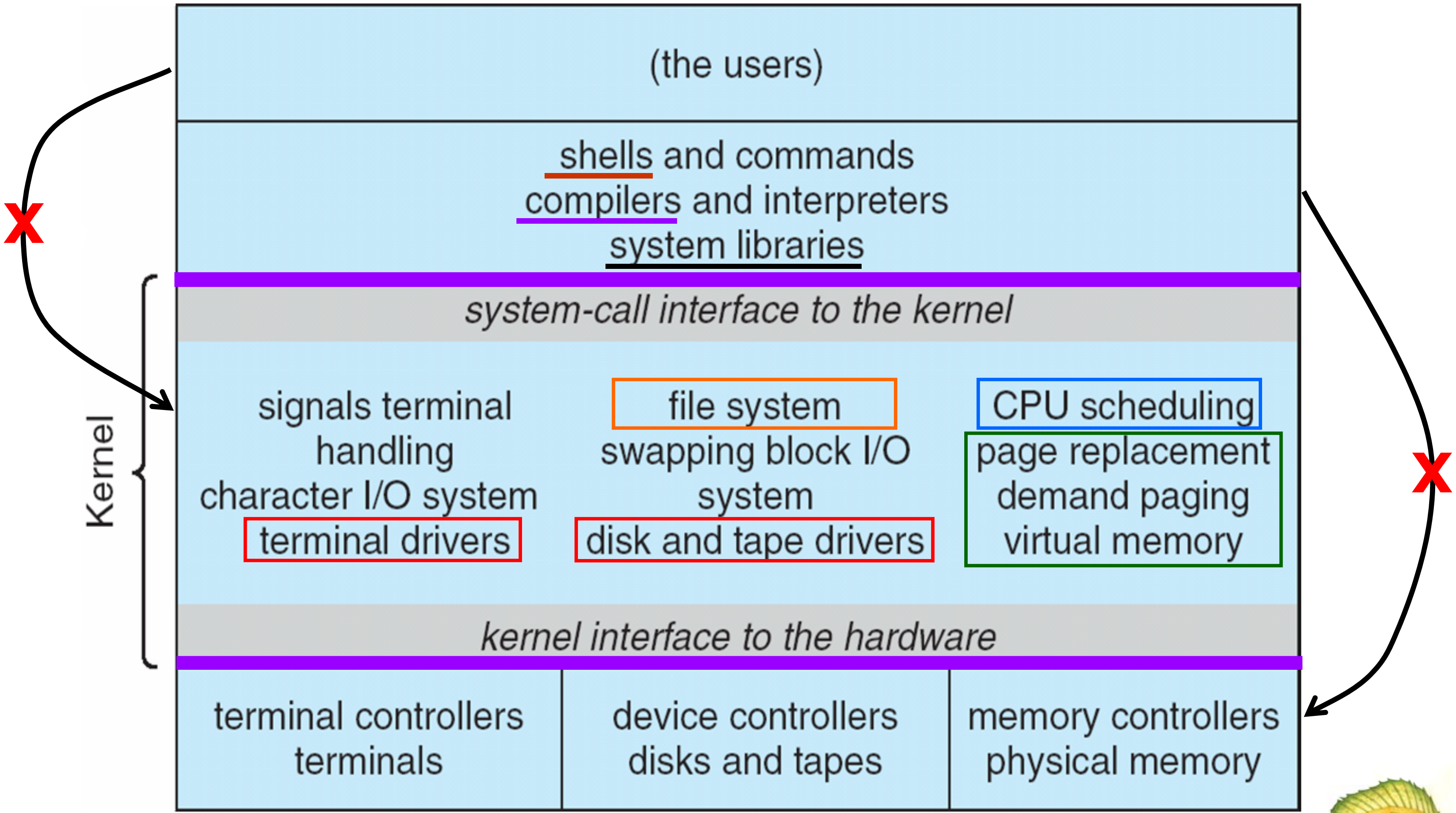
  - Although **MS-DOS** has some **structure**, its **interfaces** and **levels** of functionality are **not well separated** (application program can call ROM BIOS)



Intel 8088 CPU provides no **dual modes**

Traditional UNIX System Structure

- **Limited Structuring** (Beyond simple but not fully layered)



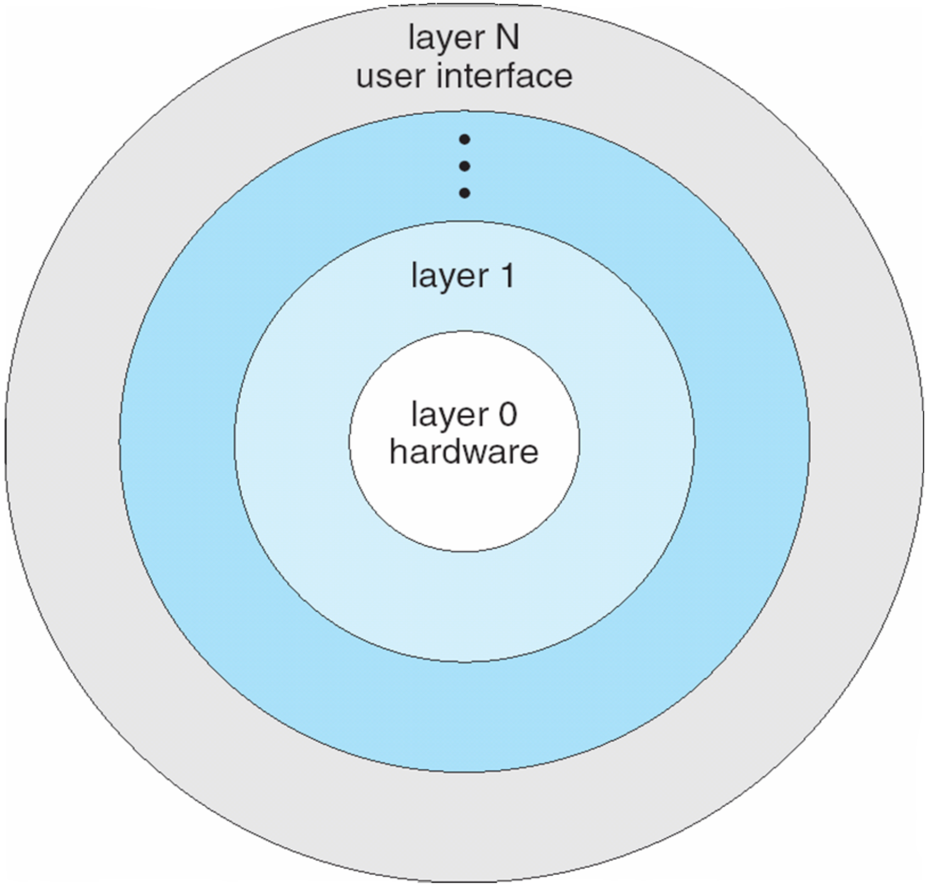
UNIX Structure

- **UNIX** – the original UNIX operating system had **limited structuring**.

- The **UNIX** OS consists of two separable parts
  - **Systems programs**
    - ***shells, compilers, system libraries***
  - **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

Layered Approach

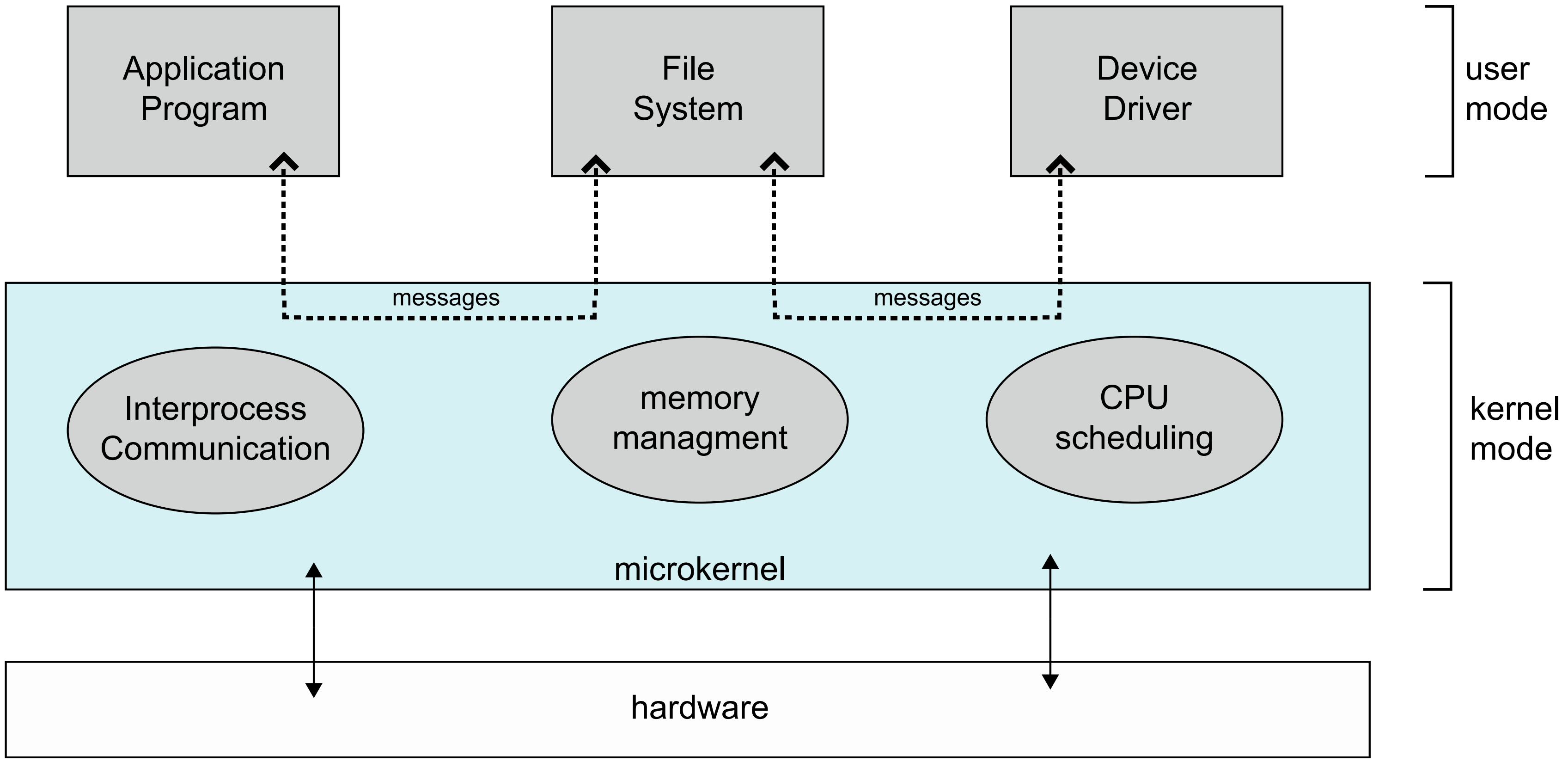
- **OS** is divided into a number of **layers** (levels),
  - each built on top of **lower layers**
  - **the bottom layer** (layer 0), is the **hardware**;
  - **the highest** (layer N) is the **user interface**
- Modularity: layers are selected such that each uses functions and services of only lower-level layers



Microkernel System Structure

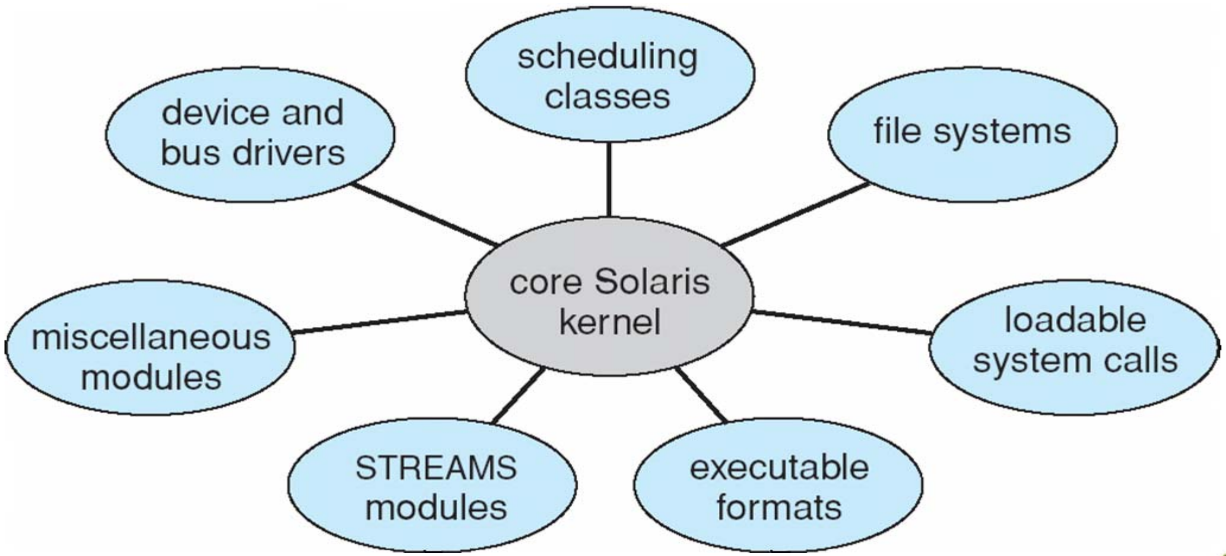
- **Moves as much** from the ***kernel*** into ***user*** **space**
  - e.g. CMU Mach, 1980.
- **Communication** takes place between **user modules**
  - using **message passing**
- **Benefits**:
  - Easier to **extend** a microkernel
  - Easier to **port** the operating system to **new architectures**
  - More **reliable** and **secure** (less code is running in kernel mode)
- **Detriments**:
  - **Performance overhead** of user space to kernel space communication
  - For example, **Windows NT** (under ver. 4) delivers **lower performance** than **Windows 95**

Microkernel System Structure



Modules: Modular Kernel

- Most **modern OS** implement **loadable kernel modules**
  - Each core component is separate, e.g. **scheduling classes**, **file systems**, **device and bus drivers**
  - Each talks to the others over **known interfaces**
  - Each is **loadable** as needed within the kernel
  - E.g. Linux, Solaris, Mac OS X, Windows



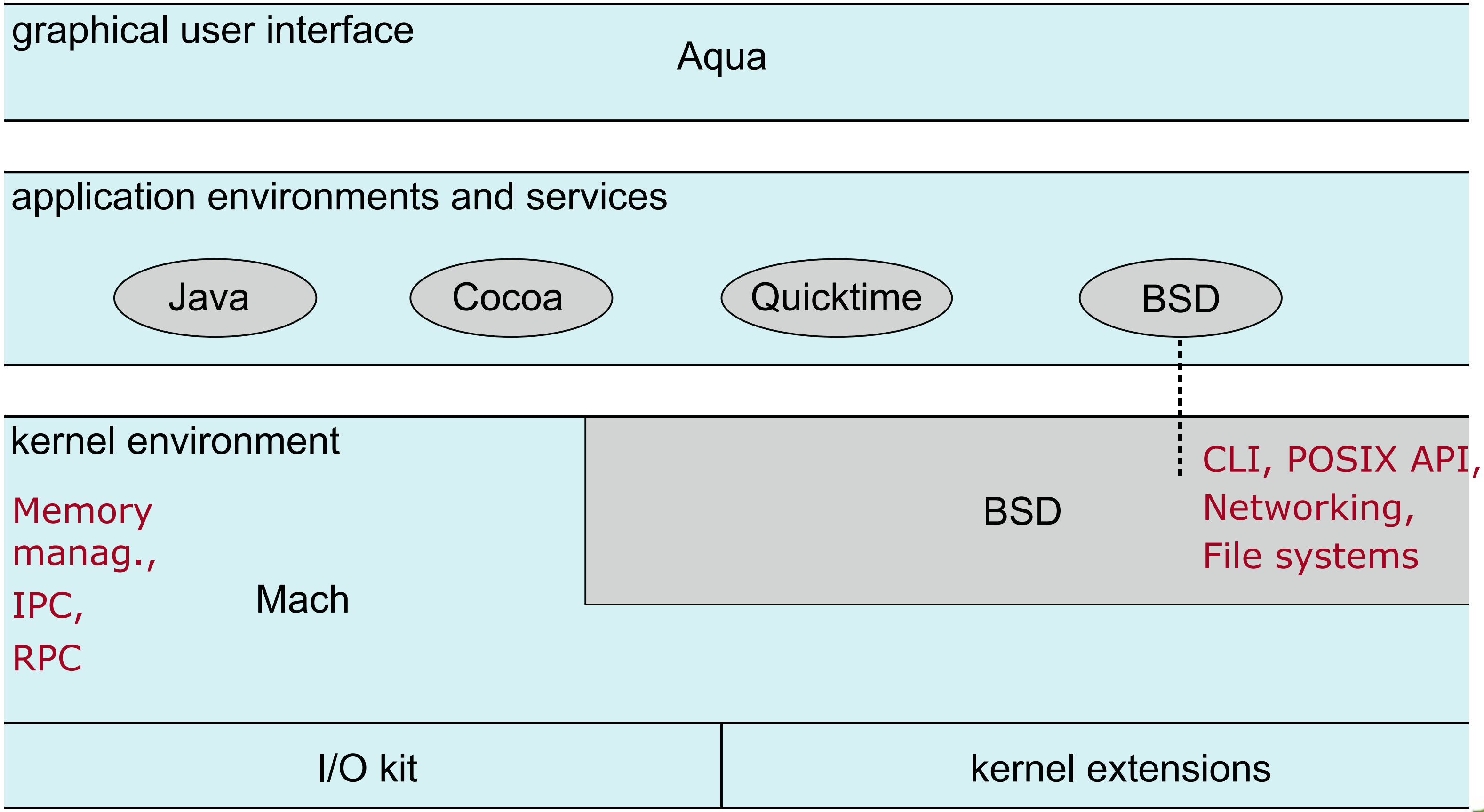
Hybrid Systems

- Most **modern operating systems** 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***). Also support dynamically **loadable modules**

- 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

- **Hybrid** structure; **Layered** structure
- Cocoa specifies API for **Objective-C language** (for writing applications)



iOS

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

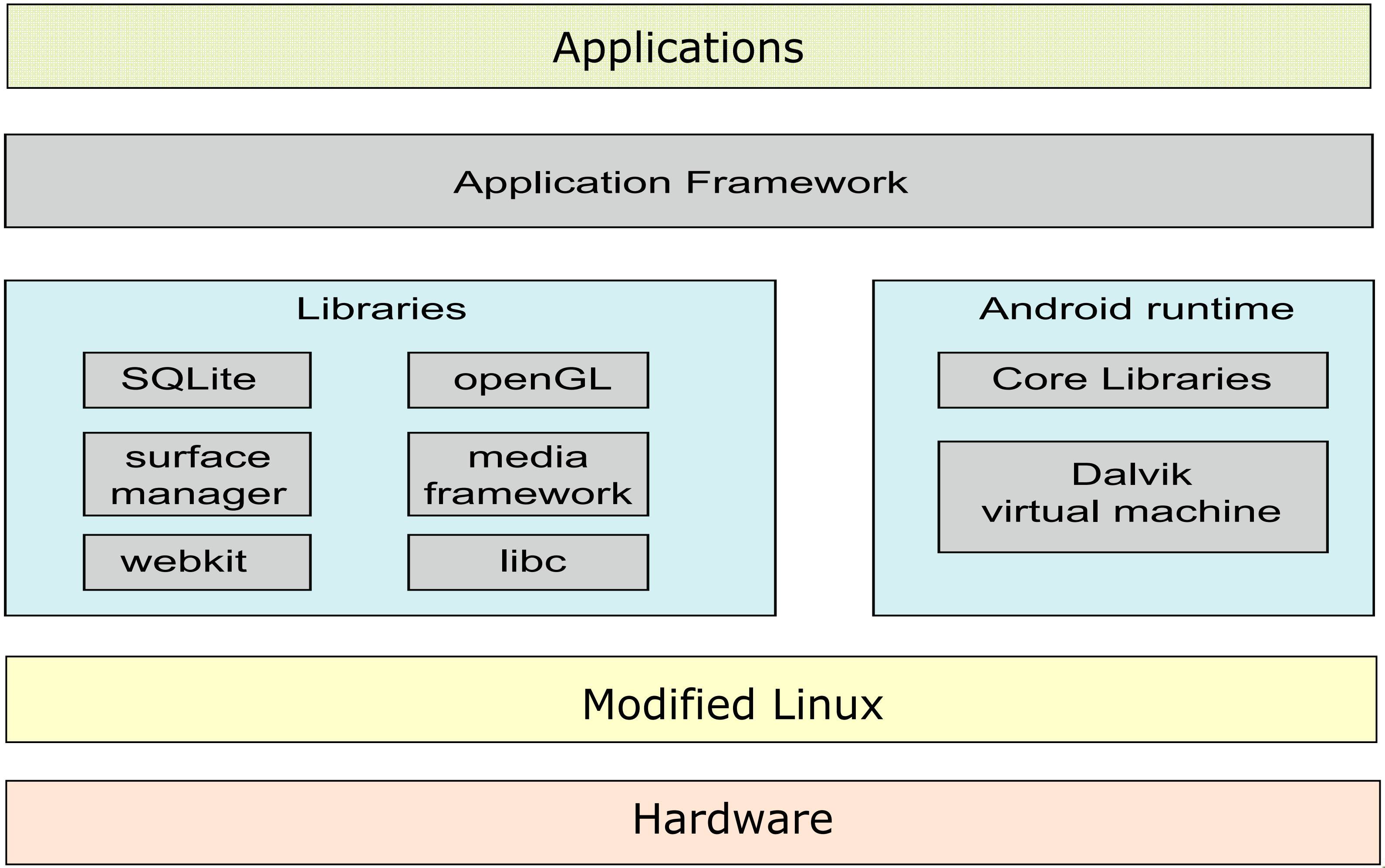
Layered structure

Cocoa Touch
Media Services
Core Services
Core OS

Android

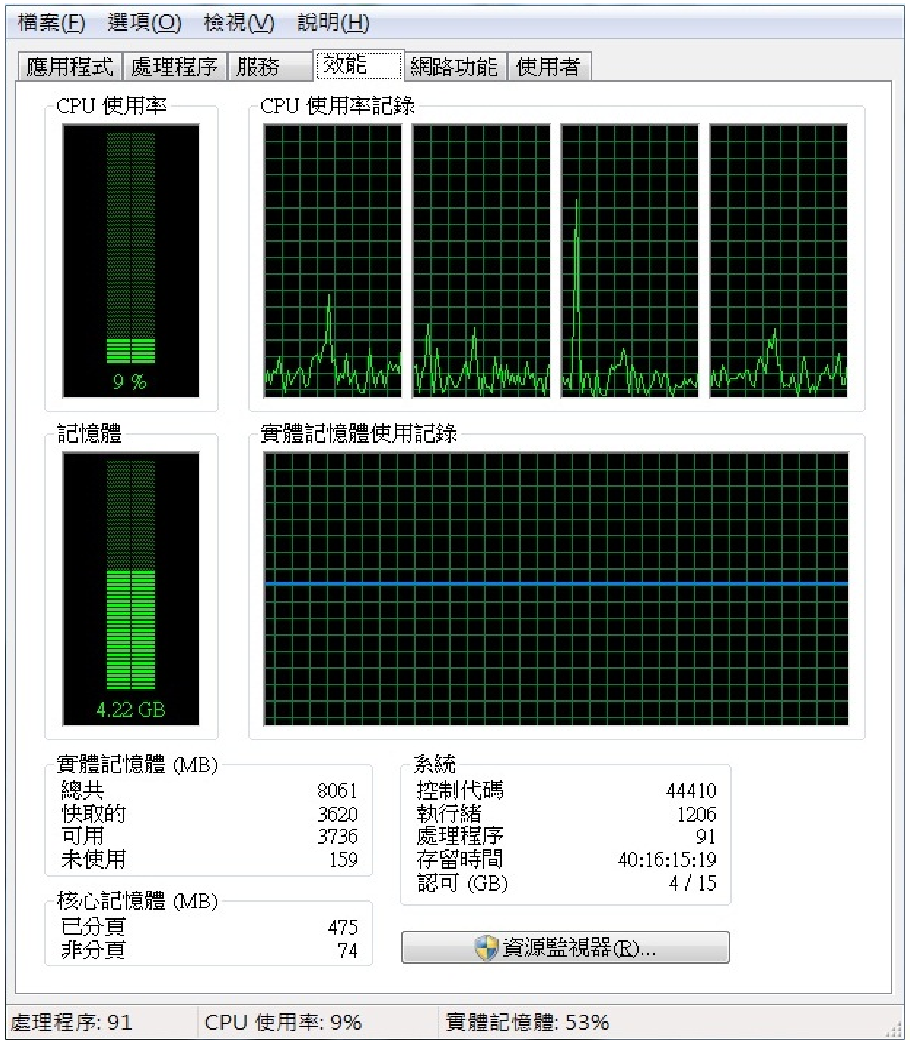
- Developed by **Open Handset Alliance** (mostly Google)
  - Open Source
  - Similar stack to **IOS**
- Based on **Linux kernel** but modified
  - Provides ***process***, ***memory***, ***device-driver*** management
  - Adds ***power management***
- **Runtime environment** includes **core set of libraries** and **Dalvik virtual machine**
  - Apps developed in **Java** plus **Android API** (Google designed)
  - Java class files compiled to Java bytecode then translated to executable that runs on **Dalvik VM**
  - Libraries include frameworks for *web browser* (webkit), *database* (SQLite), *multimedia*, smaller *libc*

Android Architecture



Performance Tuning

- **Improve performance** by removing **bottlenecks**
- **OS** must provide means
  - to compute and display **measures** of **system behavior**
  - For example, "**top**" program or "**Windows Task Manager**"



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 program** obtains information concerning the specific **configuration** of the **hardware system**
  - ***CPU type***, ***memory size***, ***device types***
- **Compile time generation**: A system **administrator** can use it to modify a copy of the source code of OS, and then compile the **source code**
- **Execution time generation**: The selection of **OS modules** occurs at **execution time**

System Boot

- **Booting** – starting a computer by **loading the kernel**
- When power initialized, execution starts at a fixed memory location
  - **Firmware ROM** used to hold initial boot code
- **OS** must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader (program)**, 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