Chapter 2: Operating Syst em Structures

Contents

- Operating System Services

- User Operating System Interface

- System Calls
- Types of System Calls

- System Programs

- Operating System Design and Implementation
- Operating System Structure

- System Boot

2.1 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** **(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** - 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.

Operating System Services (Cont.)

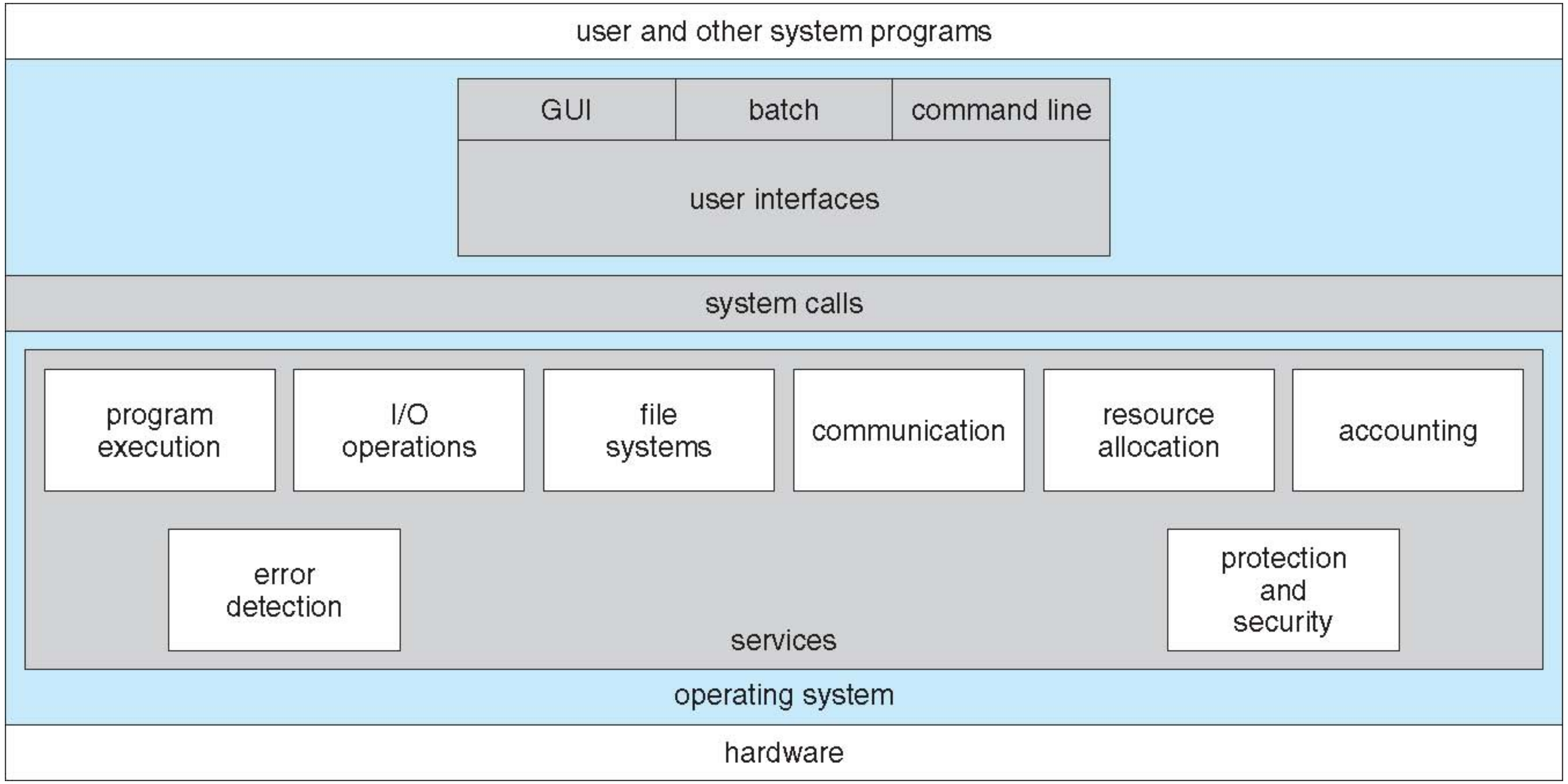
- **Communications** – Processes may exchange information, on the sa me computer or between computers over a network
  - Communications may be via shared memory or through message passing (pac kets moved by the OS)
- **Error detection** – OS needs to be constantly aware of possible error s
  - May occur 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 abilitie s to efficiently use the system

Operating System Services (Cont.)

Another set of OS functions exists for ensuring the efficient operation of the system itsel f via resource sharing

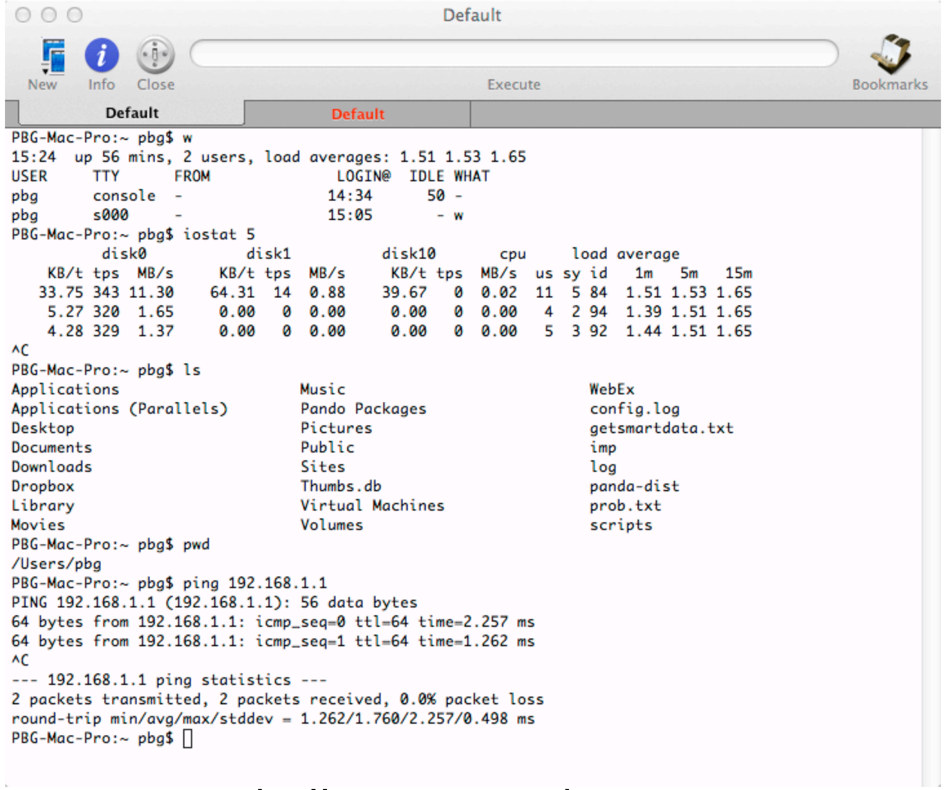
- **Resource allocation -** When multiple users or multiple jobs running concurrently, res ources must be allocated to each of them
  - Many types of resources - Some (such as CPU cycles, main memory, and file stora ge) may have special allocation code, others (such as I/O devices) may have gener al request and release code
- **Accounting -** To keep track of which users use how much and what kinds of compute r resources
- **Protection and security -** The owners of information stored in a multiuser or network ed computer system may want to control use of that information, concurrent processe s 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 def ending external I/O devices from invalid access attempts

A View of Operating System Services



2.2 User Operating System Interface - CLI

- CLI or **command interpreter** allows direct command entry



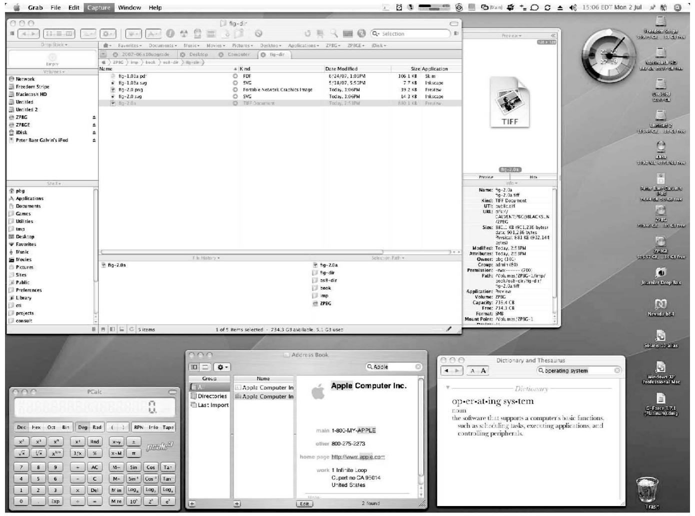
Bourne Shell Command Interpreter

- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
  - Sometimes commands built-in, sometimes just names of programs
  - If the latter, adding new features doesn't require shell modification

User Operating System Interface - GUI

- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC

The Mac OS X GUI



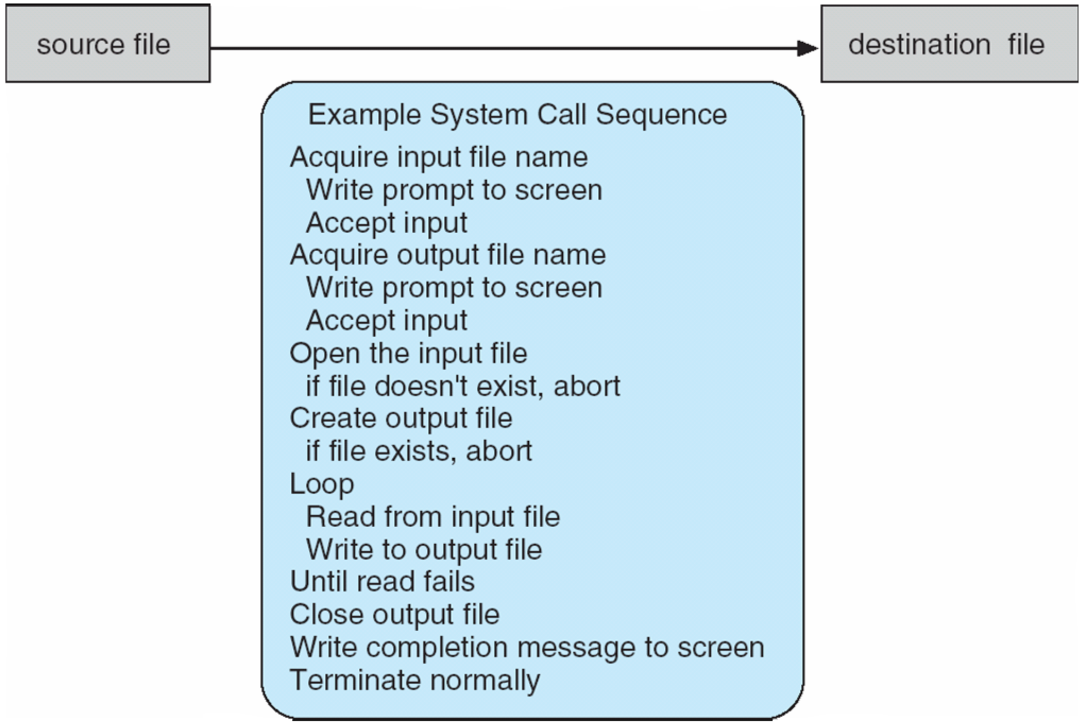
Touchscreen interface

User Operating System Interface - GUI

- 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 underneat h and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GN OME)

2.3 System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- System call sequence to copy the contents of one file to another file



System Calls (Cont'd)

- 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?

Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

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

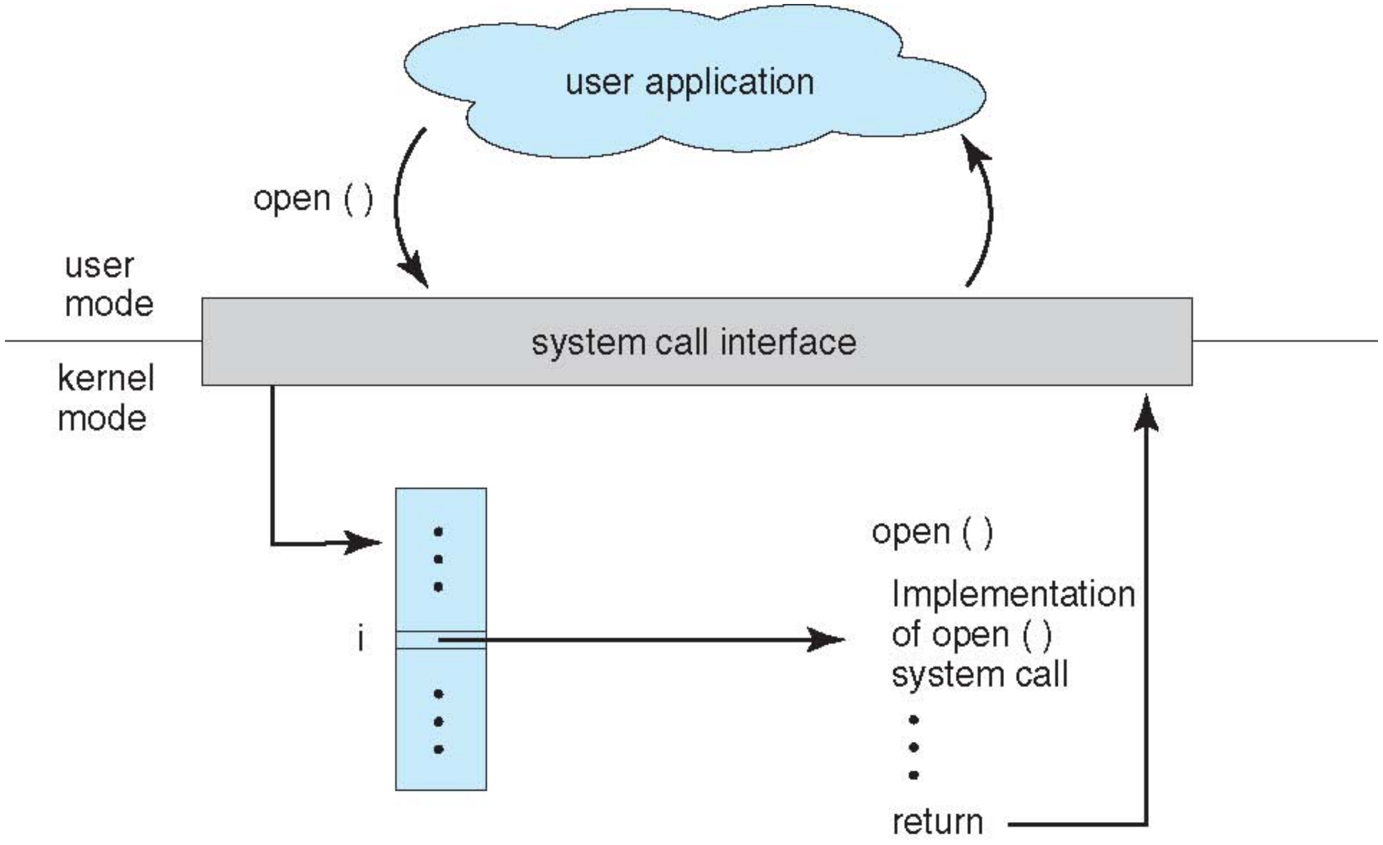
return value	function name	parameters
`ssize_t`	`read`	`(int fd, void *buf, size_t count)`

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

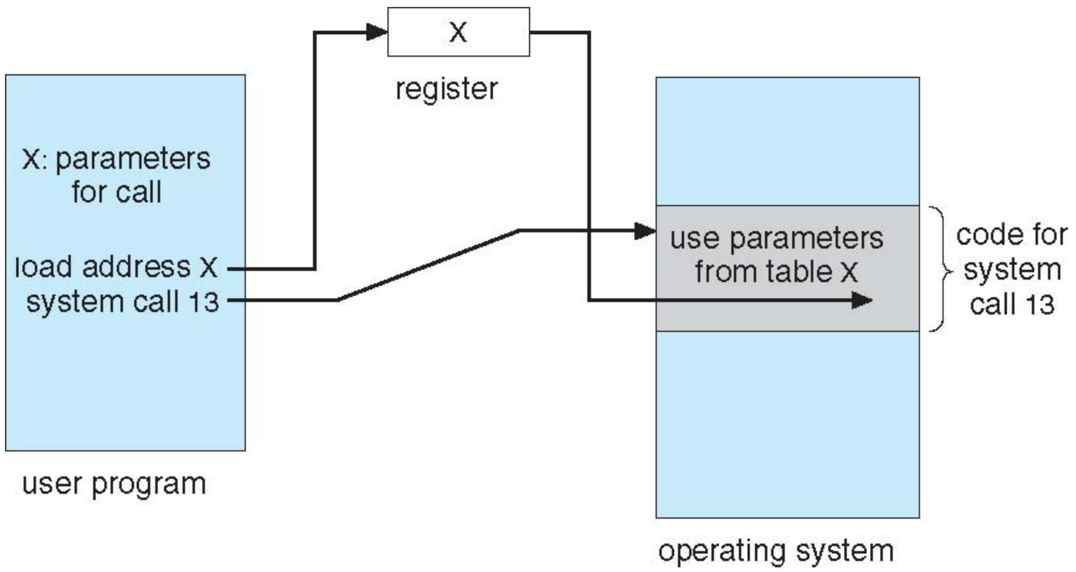
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

System Call Implementation



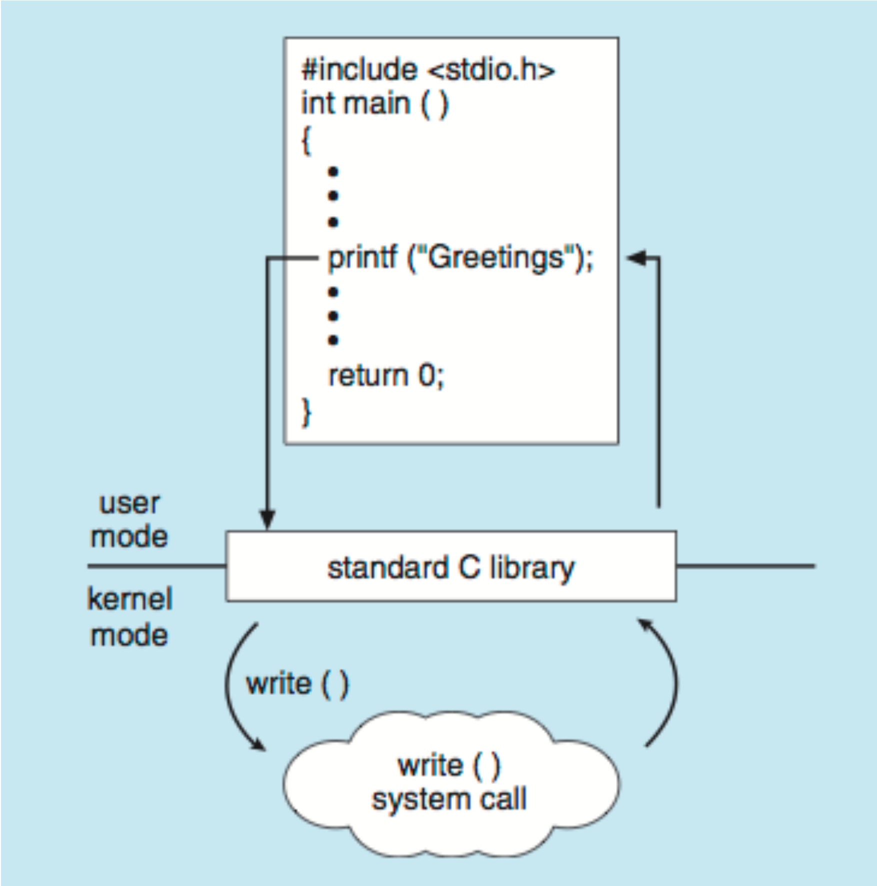
System Call Parameter Passing

- 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



Standard C Library Example

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

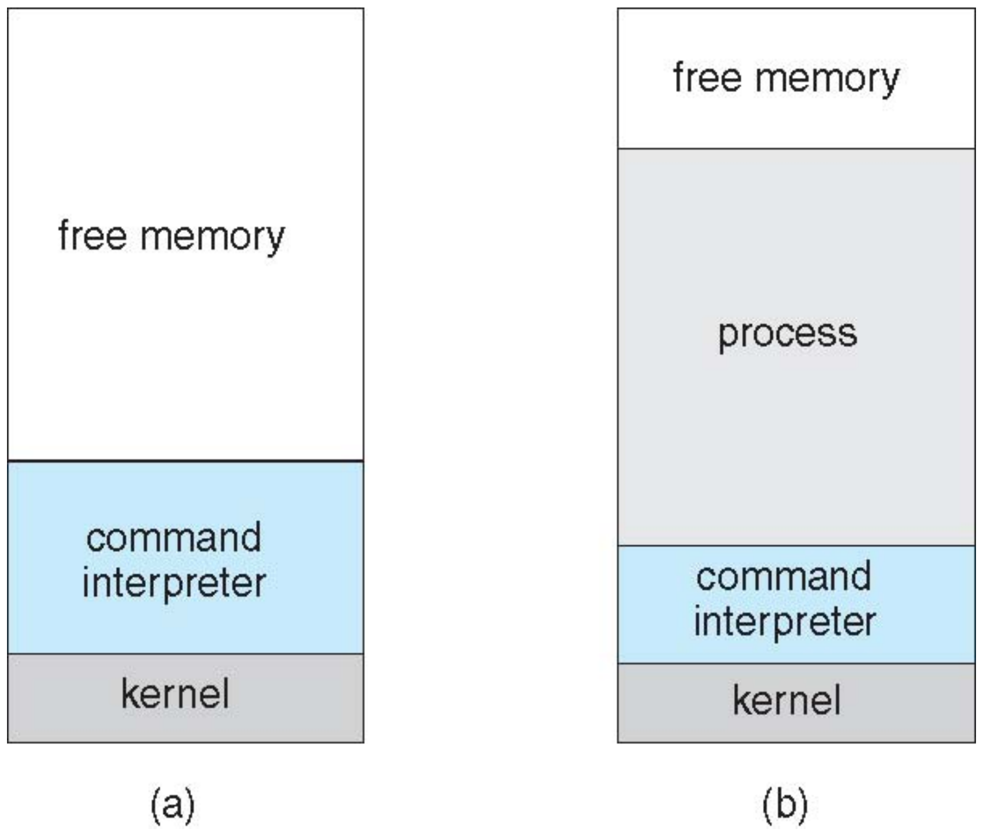


2.4 Types of System Calls

- Process control
  - end, abort
  - create process, 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

Example: MS-DOS

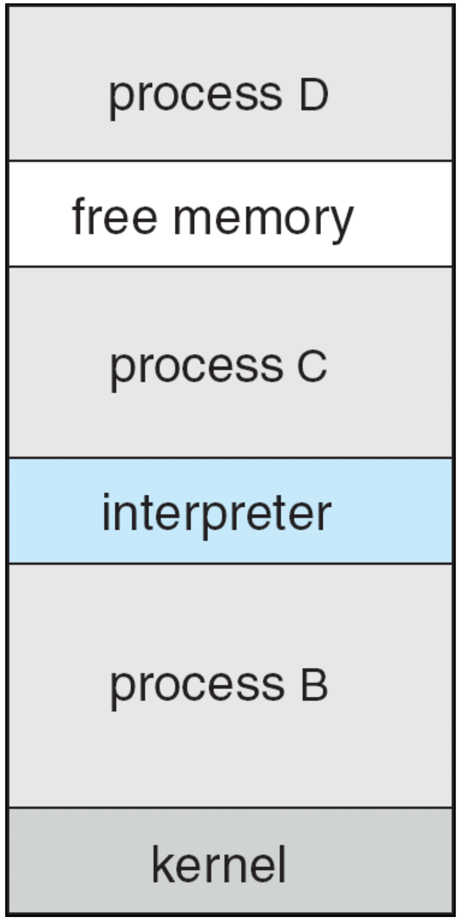


(a) At system startup (b) running a program

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

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



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 **proce ss 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()

2.5 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, printi ng
  - 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

2.6 Operating System Design and Implementation

- Design and Implementation of OS not "solvable", but some approaches have proven successful
- 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
- Important principle to separate

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

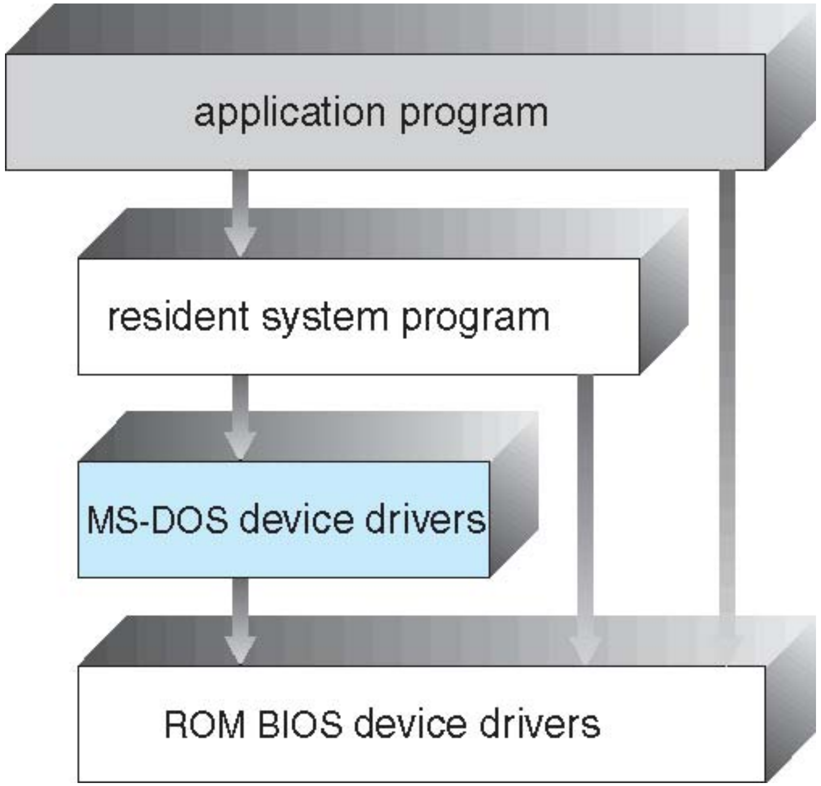
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

Implementation

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

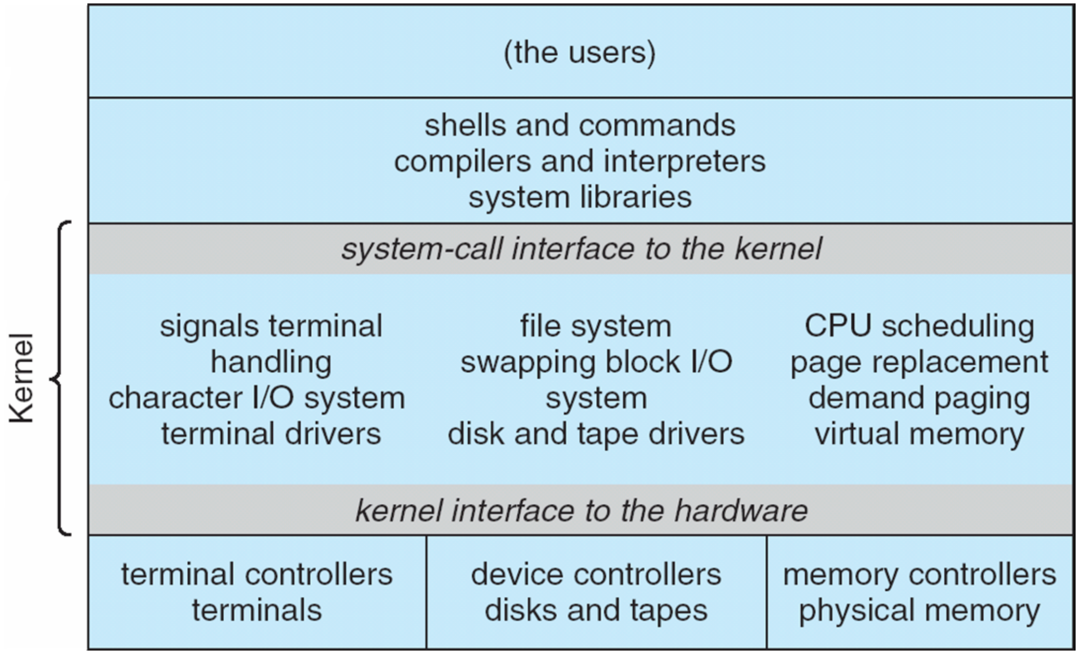
2.7 Operating System Structure

- General-purpose OS is very l arge program
- Various ways to structure on e as follows
- Simple structure - MS-DOS
  - written to provide the most fu nctionality in the least space
  - Not divided into modules
  - Although MS-DOS has some s tructure, its interfaces and leve ls of functionality are not well separated



UNIX

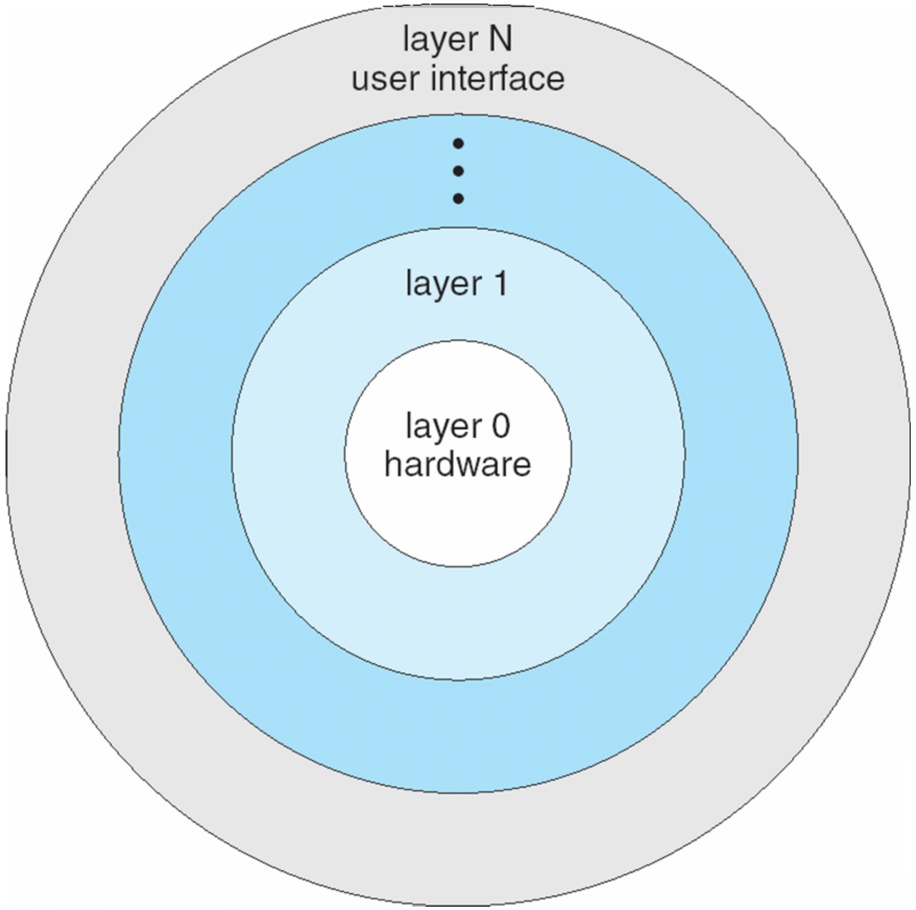
- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level



Layered Approach

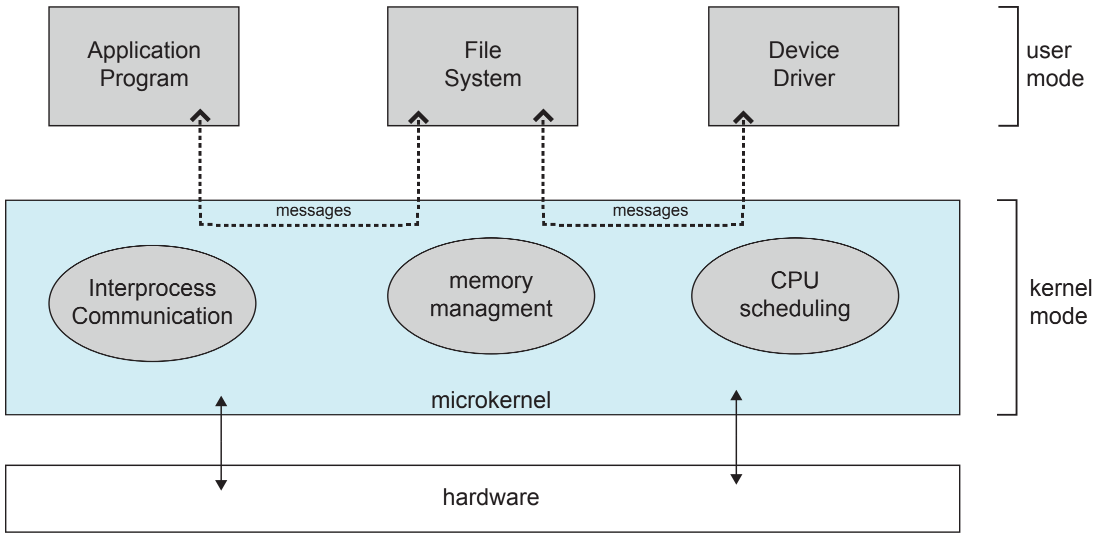
- The operating system is divided i nto a number of layers (levels), ea ch built on top of lower layers. T he bottom layer (layer 0), is the h ardware; the highest (layer N) is t he user interface.

- With modularity, layers are selecte d such that each uses functions (operations) and services of only l ower-level layers

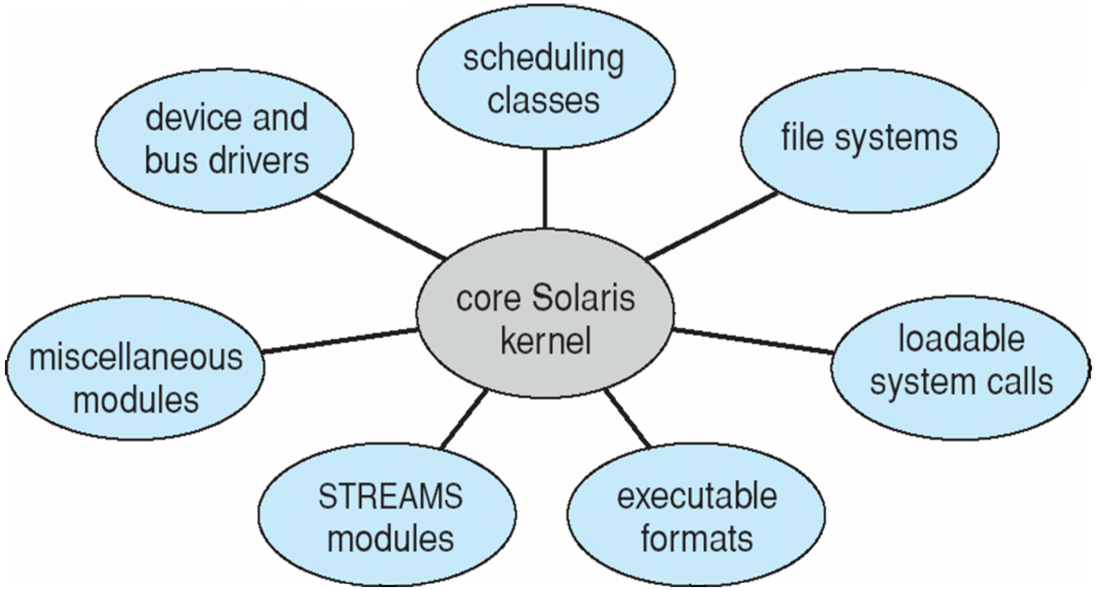


Microkernel System Structure

- 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



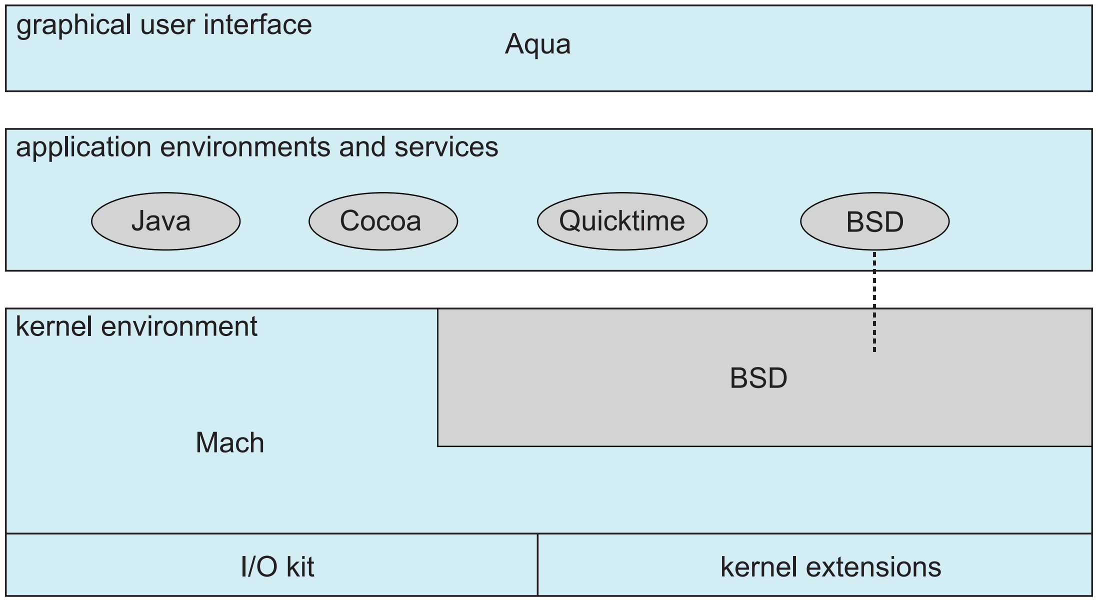
Modules



- Most 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

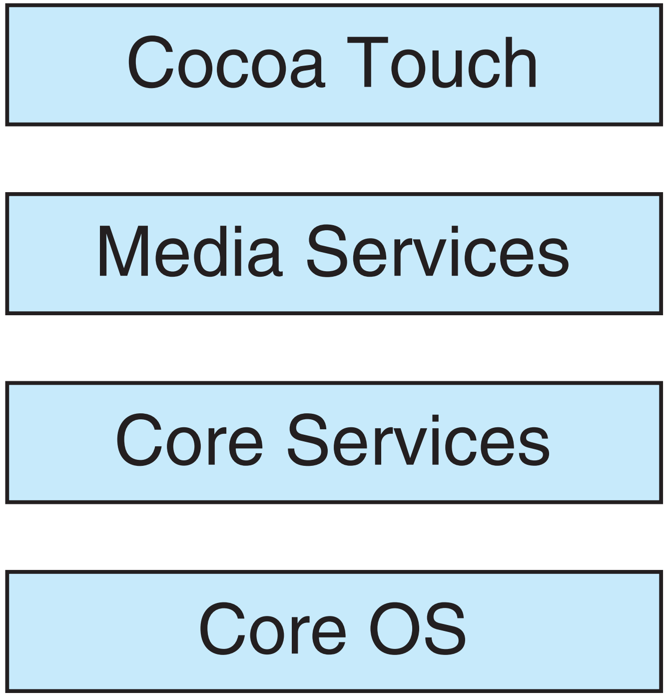
Hybrid Systems

- Most modern operating systems actually not one pure model
  - Hybrid to address performance, security, usability needs
  - Linux and Solaris kernels in kernel addres s space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microk ernel
- Apple Mac OS X hybrid, layered, **Aqua** U I plus **Cocoa** programming environment
  - Below is kernel consisting of Mach micro kernel and BSD Unix parts, plus I/O kit a nd dynamically loadable modules (called **kernel extensions**)



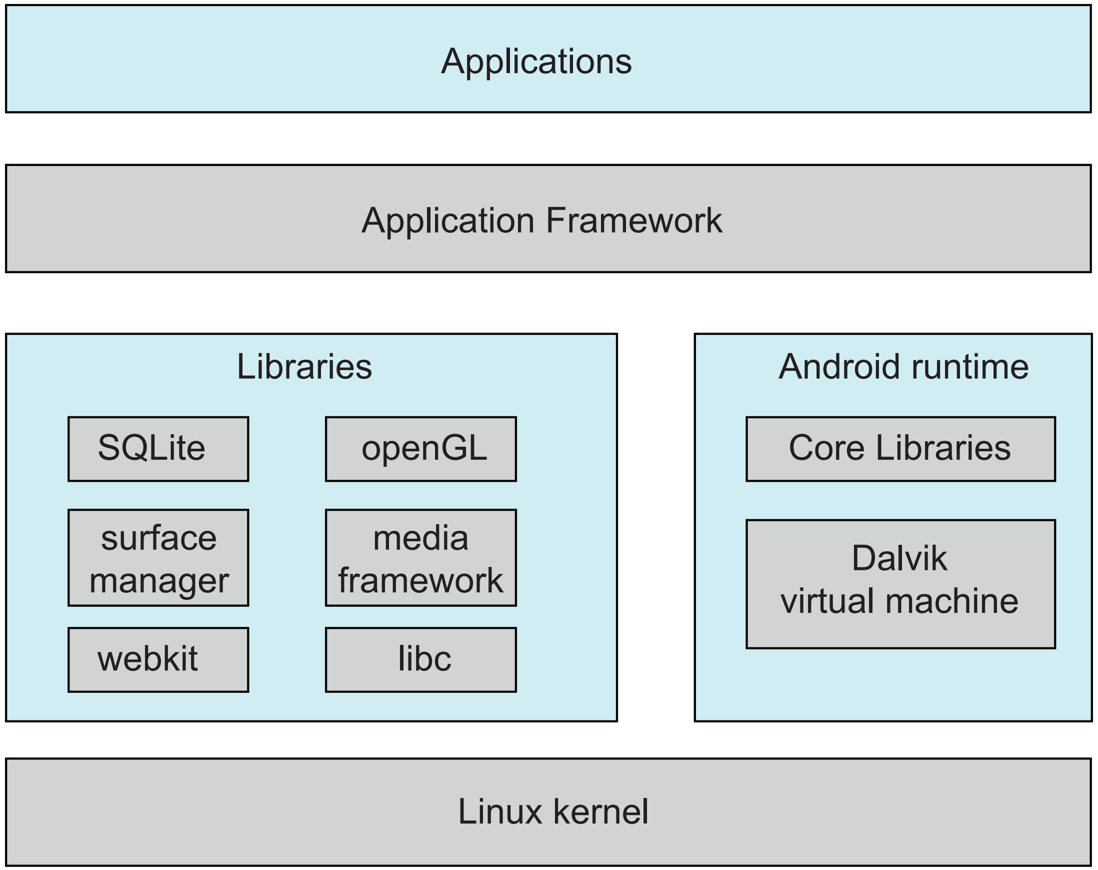
iOS

- Apple mobile OS for ***iPhone***, ***iPad***
- Structured on Mac OS X, added functionality. Does not run OS X applications natively
- **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
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
    - Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc



2.10 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**

Separating Mechanism and Policy

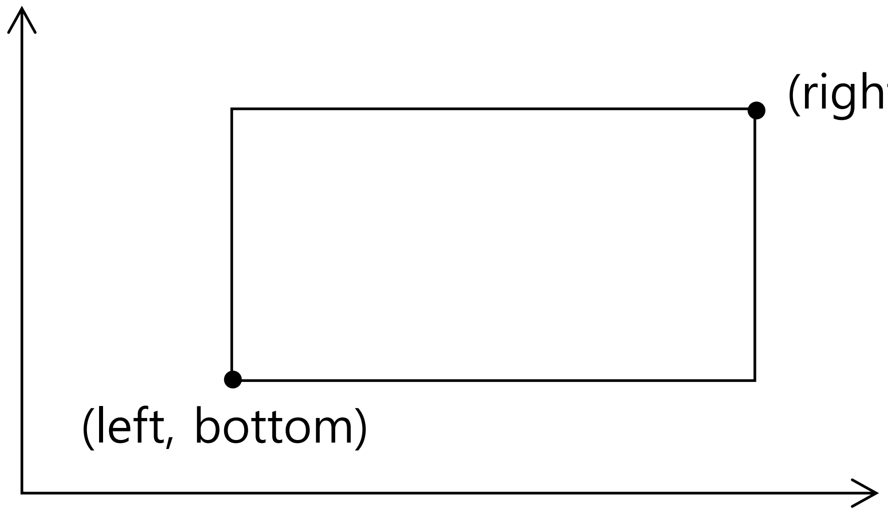
Everyday example of mechanism/policy separation

- the use of “card keys” to gain access to locked doors
  - The mechanisms do not impose any limitations on entrance policy (which people should be allowed to enter which doors, at which times).
  - These decisions are made by a centralized security server, which (in turn) probably makes its decisions by consulting a database of room access rules.
  - Specific authorization decisions can be changed by updating a room access database.
  - If the rule schema of that database proved too limiting, the entire security server could be replaced while leaving the fundamental mechanisms (readers, locks, and connections) unchanged.
- Contrast this with issuing physical keys: if you want to change who can open a door, you have to issue new keys and change the lock.
- This intertwines the unlocking mechanisms with the access policies. For a hotel, this is significantly less effective than using key cards.

Separating mechanism from policy - function implementation

- Function that calculates area of rectangle

```
int rect_area(int left, int top, int right, int bottom) {
        return (right - left) * (top - bottom);
}
```



Adding some exception/error handling

```
int rect_area(int left, int top,
              int right, int bottom)
{
  if (left >= right)
   left = right;
  if (bottom > = top)
    bottom = top;

  return (right-left)*(top-bottom)
}
```

```
int rect_area2(int left, int top,
               int right, int bottom)
{
  if (left >= right || bottom > top)
   return -1;

  return (right-left)*(top-bottom)
}
```

rect_area() vs. rect_area2()

- Code for calculating area of rectangle is duplicated. Just one-line code. So, dulplicating is not a big deal. But, it's not good in terms of code structure. Why did this happen?

- Calculating rectangle area is 'Mechanism'. But, error handling is 'Policy' at this example.

Example should be implemented like this:

```
static inline int _rect_area(int left, int top, int right, int bottom) {
        return (right - left) * (bottom - top);
}

int rect_area(int left, int top, int right, int bottom) {
        if (left >= right)
                left = right;
        if (top >= bottom)
                top = bottom;
        return _rect_area(left, top, right, bottom);
}
int rect_area2(int left, int top, int right, int bottom) {
        if (left > right || top > bottom)
                return -1;
        return _rect_area(left, top, right, bottom);
}
```

Mechanism

Policy 1

Policy 2