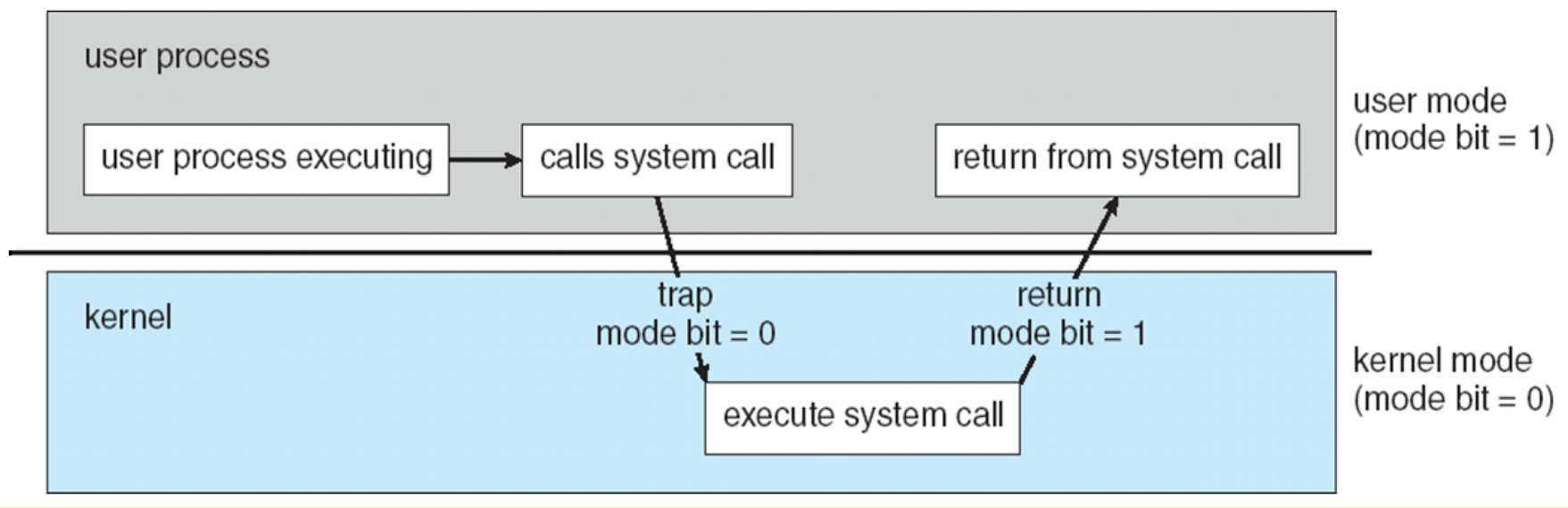
Transition from User to Kernel Mode



Timer

To prevent process to be in infinite loop (process hogging resources), a **timer** is used, which is a hardware device.

- Timer is a counter that is decremented by the physical clock.
- Timer is set to interrupt the computer after some time period
- Operating system sets the counter (privileged instruction)
- When counter reaches the value zero, and interrupt is generated.
- The OS sets up the value of the counter before scheduling a process to regain control or terminate program that exceeds allotted time

Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a ***passive entity***, process is an ***active entity***.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files, etc.
  - Initialization data
- Process termination requires reclaim of any reusable resources
- A thread is a basic unit of CPU utilization within a process.
- Single-threaded process. Instructions are executed sequentially, one at a time, until completion
  - Process has one **program counter** specifying location of next instruction to execute
- Multi-threaded process has one program counter per thread
- Typically, a system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the threads

Process Management Activities

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Memory Management

- To execute a program all (or part) of the instructions must be in memory
- All (or part) of the data that is needed by the program must be in memory.
- Memory management determines what is in memory and when
  - Optimizing CPU utilization and computer response to users
- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed

Storage Management

- OS provides uniform, logical view of information storage
- Abstracts physical properties to logical storage unit - **file**
- Files are stored in a number of different storage medium.
  - Disk
  - Flash Memory
  - Tape
- Each medium is controlled by device drivers (i.e., disk drive, tape drive)
  - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)

File System Management

- Files usually organized into directories
- Access control on most systems to determine who can access what
- OS activities include
  - Creating and deleting files and directories
  - Primitives to manipulate files and directories
  - Mapping files onto secondary storage
  - Backup files onto stable (non-volatile) storage media

Secondary-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- OS activities
  - Free-space management
  - Storage allocation
  - Disk scheduling
- Some storage need not be fast
  - Tertiary storage includes optical storage, magnetic tape
  - Still must be managed – by OS or applications

Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache are smaller (size-wise) than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

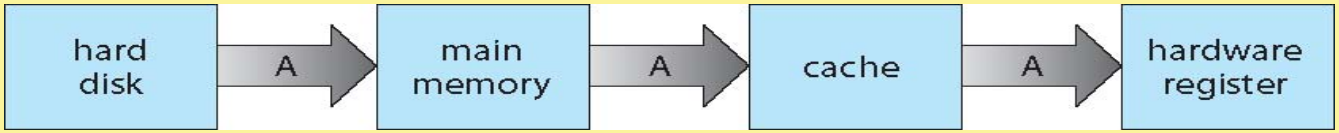
Performance of Various Levels of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Movement between levels of storage hierarchy can be explicit or implicit

Migration of data “A” from Disk to Register

- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy



- Multiprocessor environment must provide **cache coherency** in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex
  - Several copies of a datum can exist

I/O Subsystem

- One purpose of an operating system is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
  - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
  - General device-driver interface
  - Drivers for specific hardware devices

Protection and Security

- **Protection** – A mechanism for controlling access of processes (or users) to resources defined by the OS
- **Security** – A defense of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
  - User identities (**user IDs**, security IDs) include name and associated number, one per user
  - User ID is associated with all files and processes of that user to determine access control
  - Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
  - **Privilege escalation** allows user to change to effective ID with more rights

Conclusion

Conclusion:

- Operating system acts as an interface between computer hardware and users
- Makes the system usable in an user-friendly manner
- Controls usage of different hardware and software resources in a computer system
- What is an Operating System?
- Specific activities include:
  - Process Management
  - Memory Management
  - Storage Management
  - Protection and Security

NPTEL ONLINE CERTIFICATION COURSES

Thank you

Operating System Fundamentals

Santanu Chattopadhyay
Electronics and Electrical Communication Engg.

Operating System Structures

Concepts Covered:

- Services an operating system provides to users, processes, and other systems
- Various ways of structuring an operating system
- How operating systems are installed and customized and how they boot

Operating System Services

- Provides an environment for the execution of programs.
- Provides certain services to:
  - Programs
  - Users of those programs
- Basically two types of services:
  - Set of operating-system services provides functions that are helpful to the user
  - Set of operating-system functions for ensuring the efficient operation of the system itself via resource sharing


OS Services Helpful to the User

- **User interface** - Almost all operating systems have a **user interface** (**UI**). Several forms:
  - **Command-Line (CLI)** -- uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options).
  - **Graphics User Interface (GUI)** -- the interface is a window system with a pointing device to direct I\O, choose from menus, and make selections and a keyboard to enter text..
  - **Batch Interface** -- commands and directives to control those commands are entered into files, and those files are executed

  Some systems provide two or all three of these variations.


OS Services Helpful to the User (Contd.)

- **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** - Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.


OS Services Helpful to the User (Contd.)

- **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

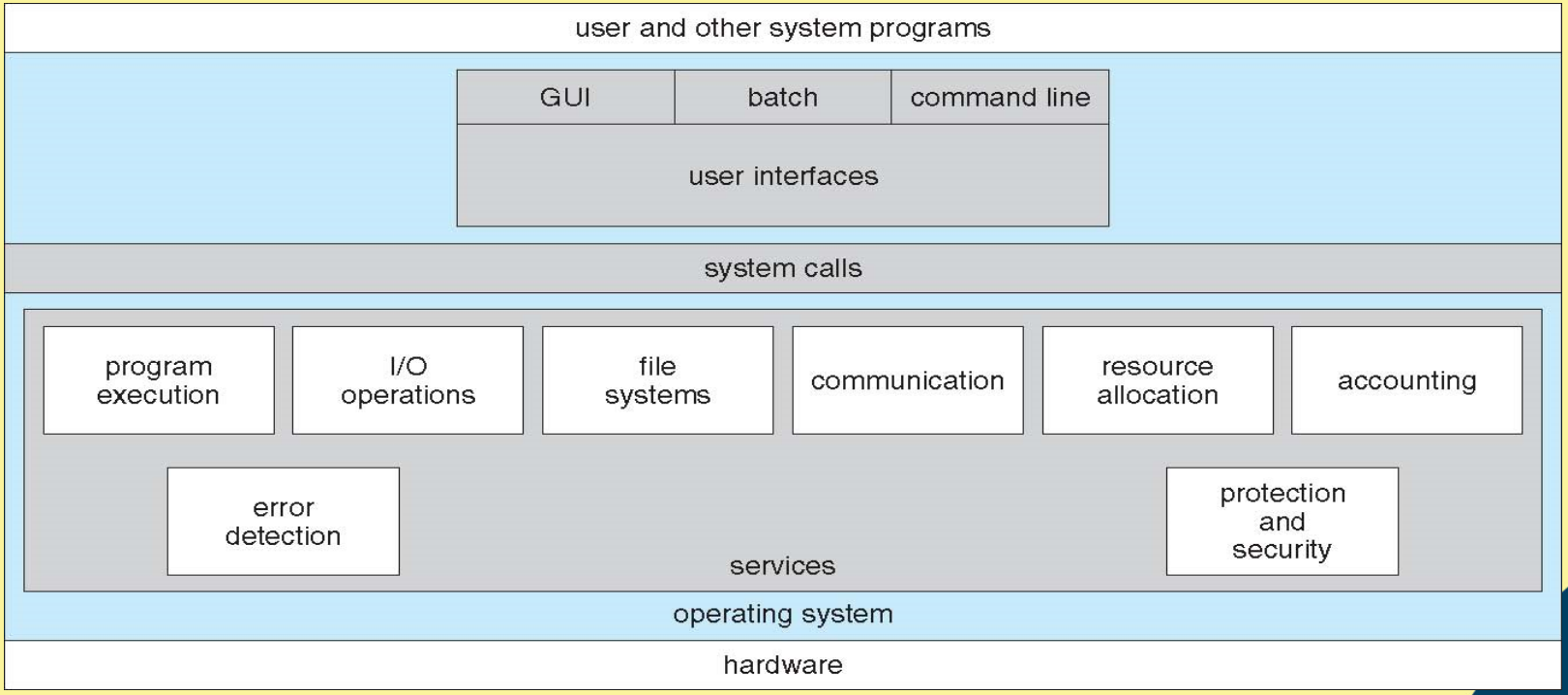
OS Services for Ensuring Efficient Operation

- **Resource allocation -** When multiple users or multiple jobs are 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

OS Services for Ensuring Efficient Operation (Contd.)

- **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


A View of Operating System Services

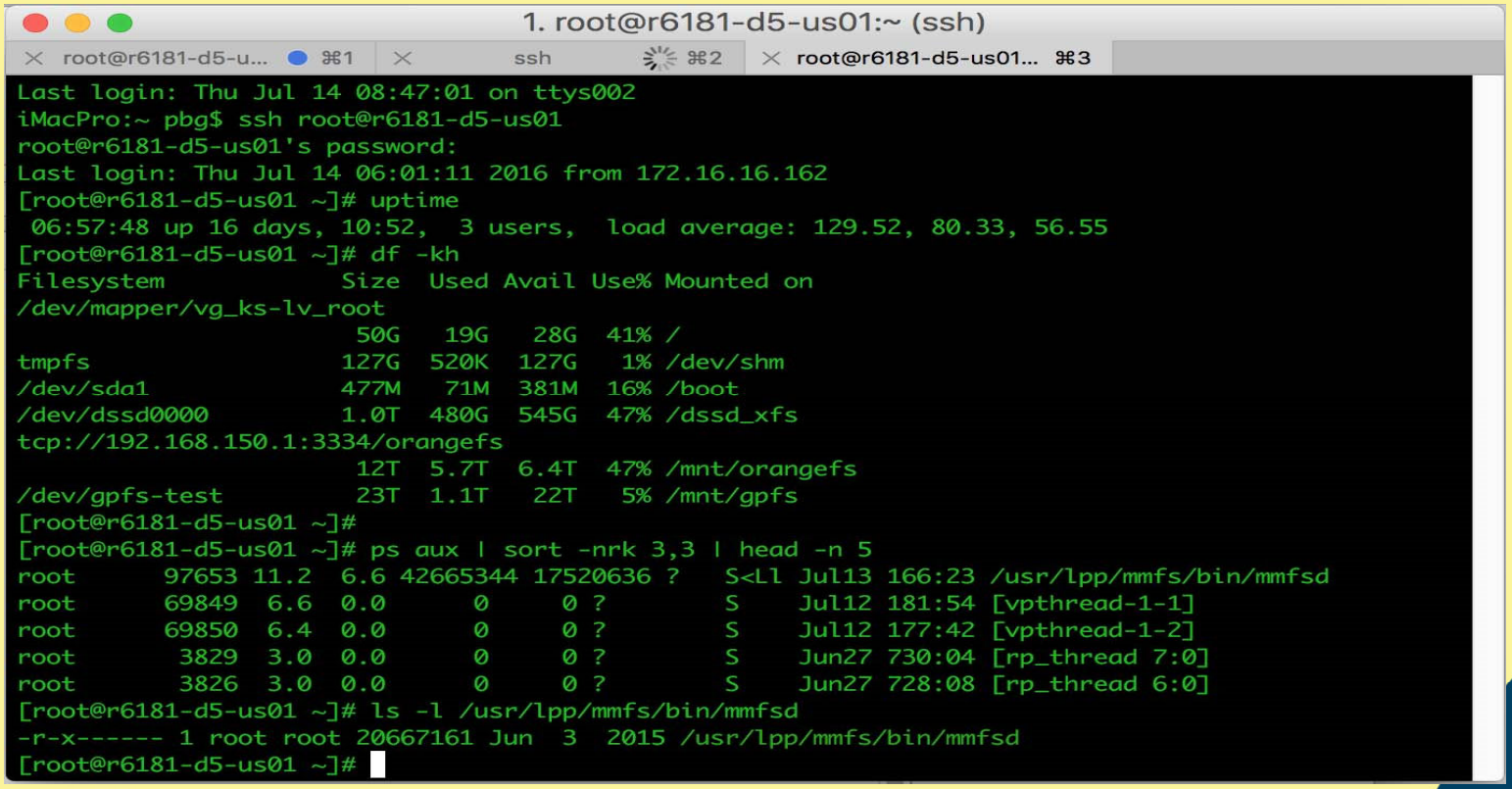


Command Interpreters (CLI)

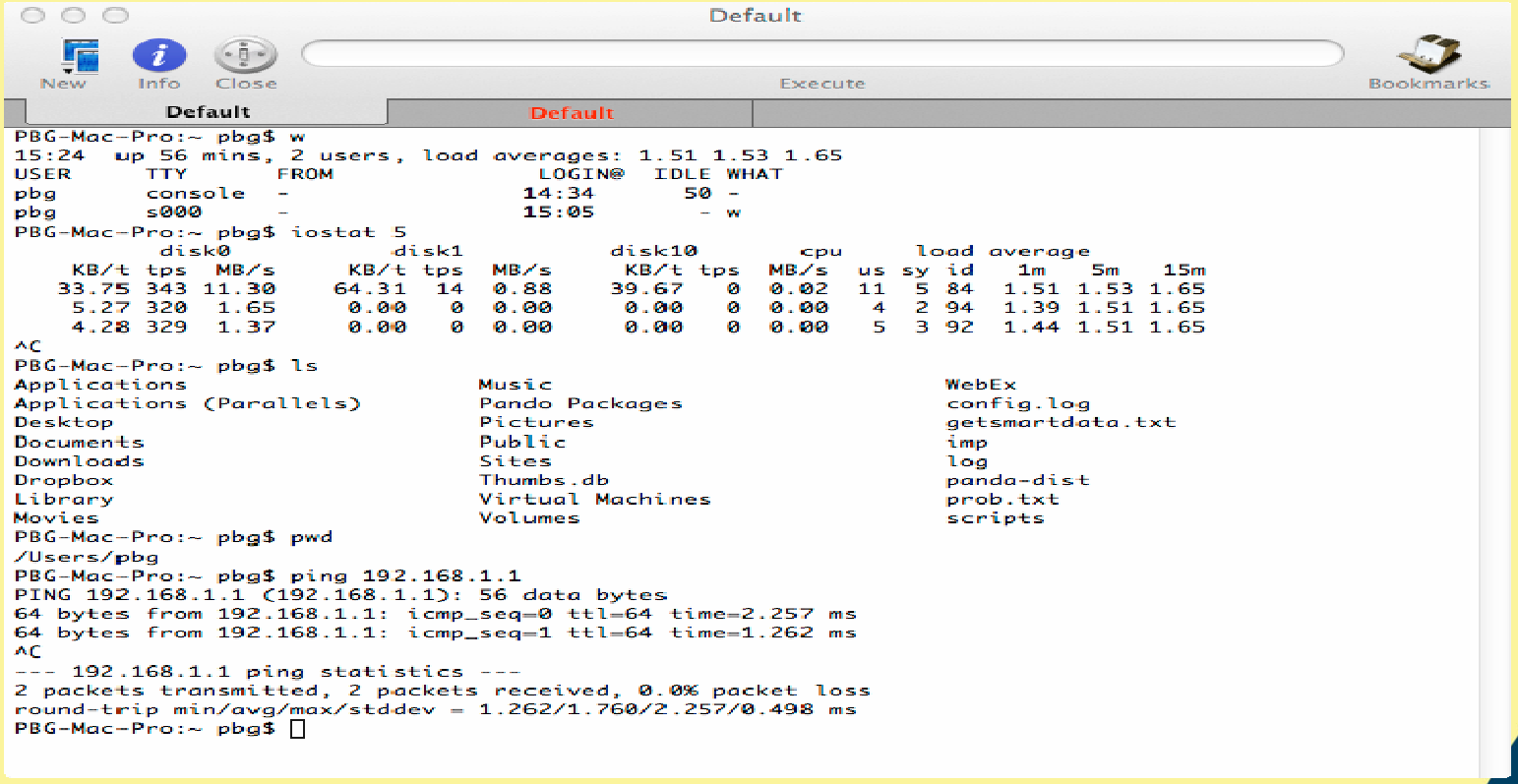
CLI allows users to directly enter commands to be performed by the operating system.

- Some operating systems include the command interpreter in the kernel.
- Some operating systems, such as Windows and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on.
- On systems with multiple command interpreters to choose from, the interpreters are known as **shells.**
- The main function of the command interpreter is to get and execute the next user-specified command.
- Sometimes commands built-in, sometimes just names of programs
  - If the latter, adding new features doesn't require shell modification


The Bourne shell command interpreter in Solaris



Bourne Shell Command Interpreter



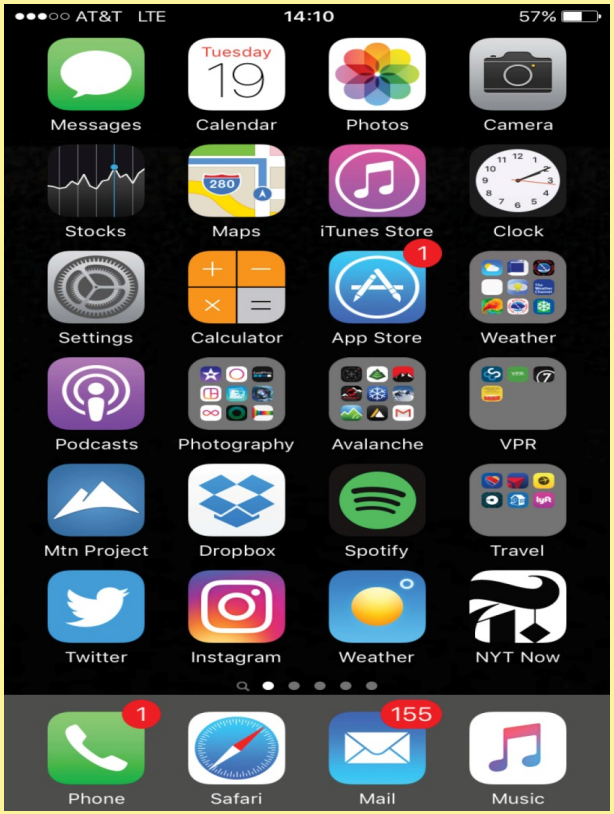

```
PBG-Mac-Pro:~ pbg$ w
15:24  up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER     TTY      FROM              LOGIN@  IDLE WHAT
pbg      console  -                 14:34     50 -
pbg      s000     -                 15:05      - w
PBG-Mac-Pro:~ pbg$ iostat 5
          disk0           disk1           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$
```

Graphical User Interfaces (GUI)

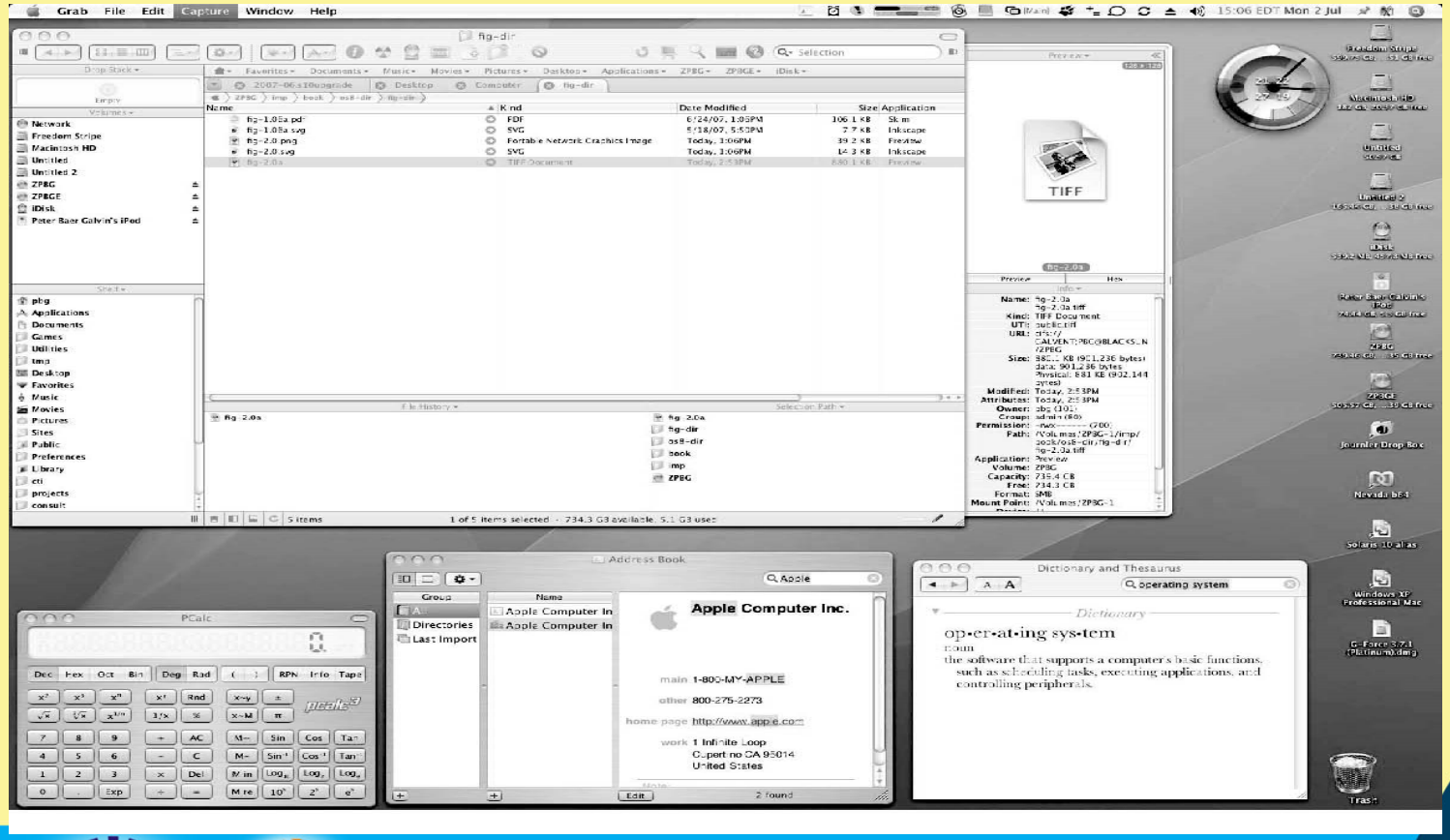
- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)


Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands.



The Mac OS X GUI

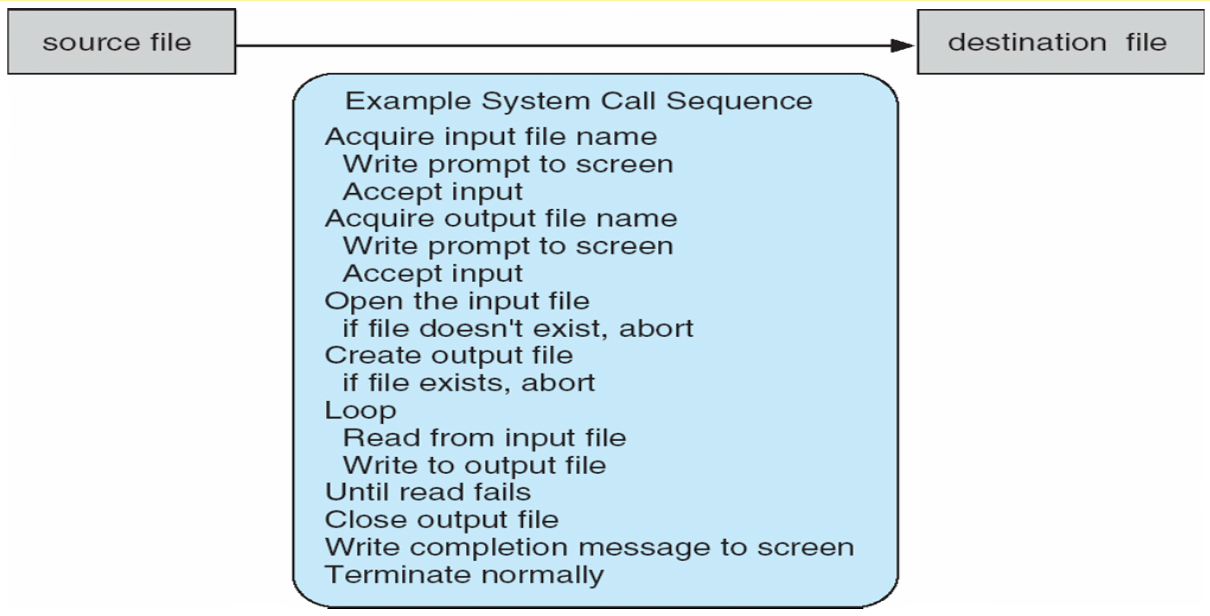


System Calls

- 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
- 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),
  - 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:

```
#include <unistd.h>

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

return value — function name — 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

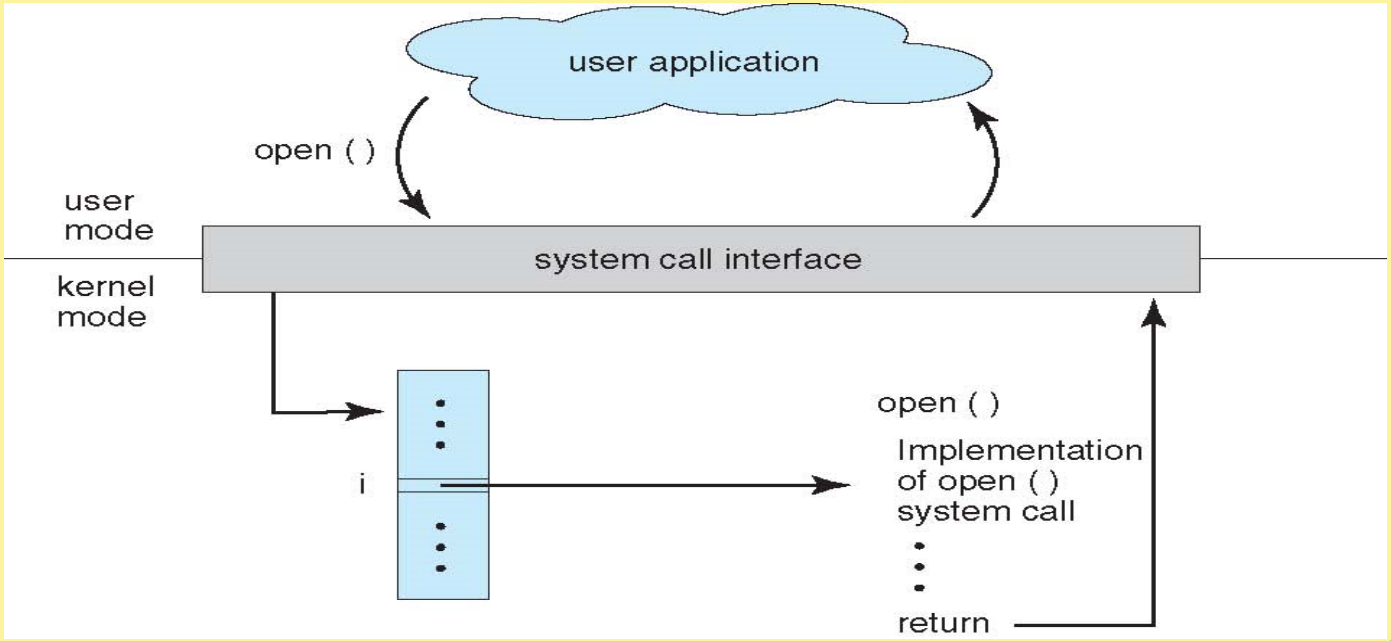
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.


System Call Implementation

- Typically, a number is 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 not know a thing about how the system call is implemented
  - Just needs to obey the API and understand what the 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)


System Call -- OS Relationship

The handling of a user application invoking the open() system call

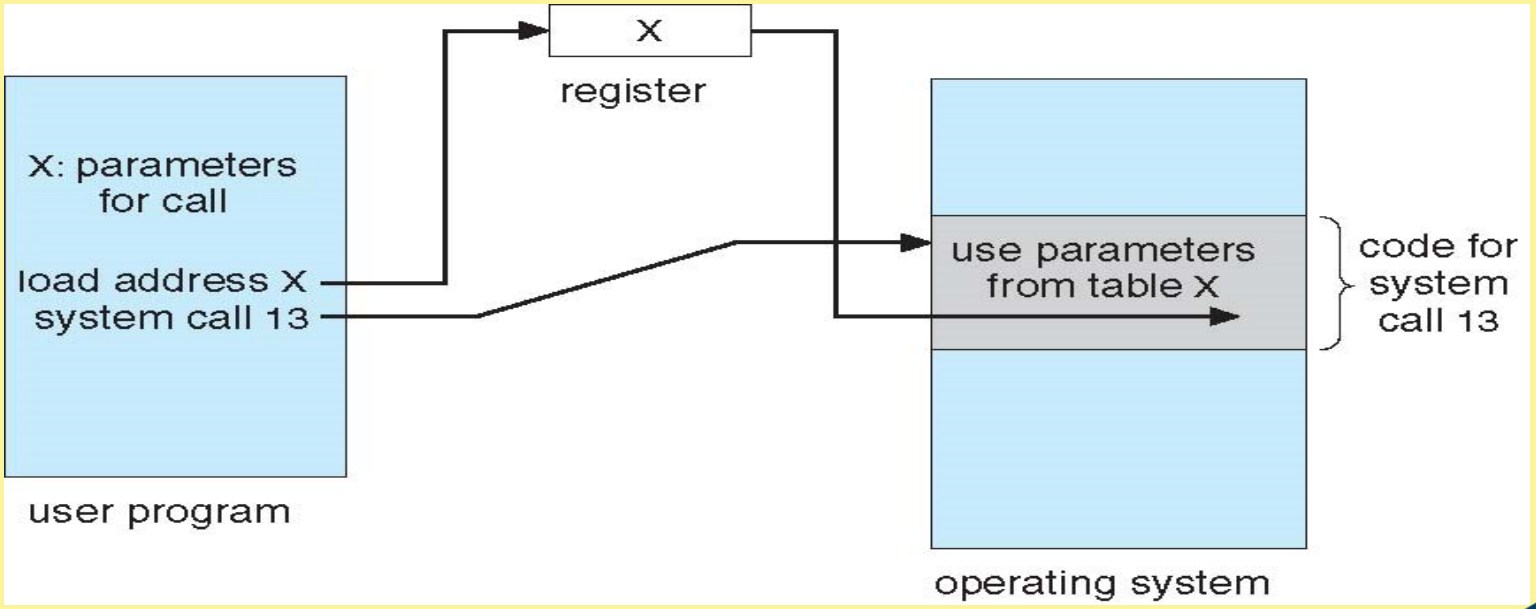


System Call Parameter Passing

- 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

x points to a block of parameters. x is loaded into a register



Types of System Calls

- System calls can be grouped roughly into six major categories:
  - Process control,
  - File manipulation,
  - Device manipulation,
  - Information maintenance,
  - Communications,
  - Protection.
- The figure in the slide # 28 summarizes the types of system calls normally provided by an operating system.


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


System Calls – File Management

- Create file
- Delete file
- Open and Close file
- Read, Write, Reposition
- Get and Set file attributes


System Calls – Device Management

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

System Calls -- Information Maintenance

- get time or date,
- set time or date
- get system data,
- set system data
- get and set process, file, or device attributes


System Calls – Communications

- create, delete communication connection
- if **message passing model:**
  - send, receive message
    - To **host name** or **process name**
    - From **client** to **server**
- If **shared-memory model:**
  - create and gain access to memory regions
- transfer status information
- attach and detach remote devices


System Calls -- Protection

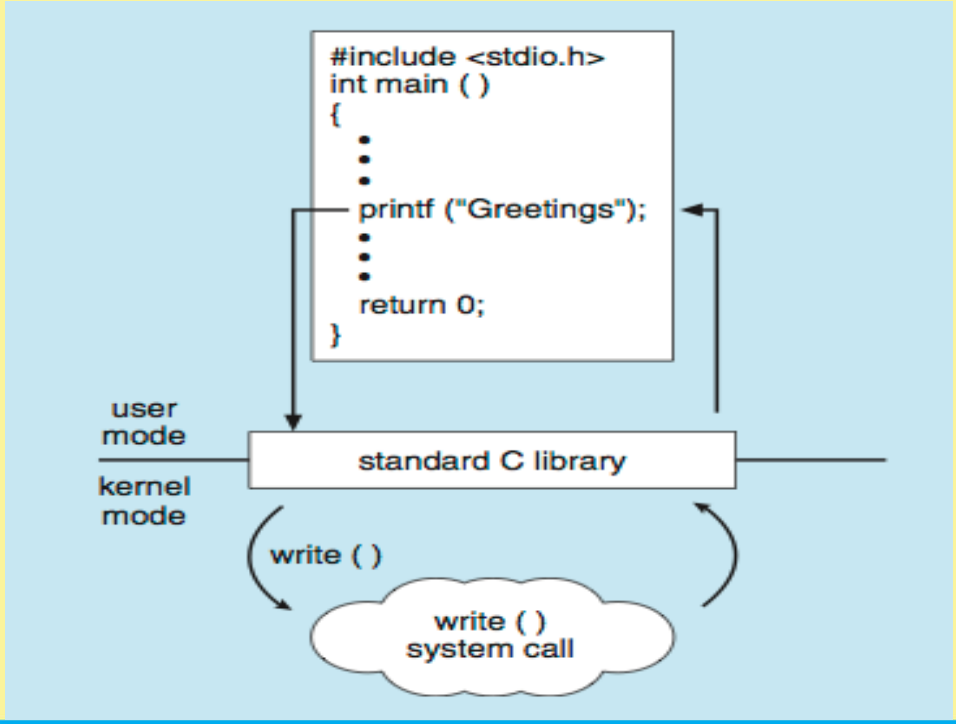
- Control access to resources
- Get and set permissions
- Allow and deny user access

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()


Example -- Standard C Library

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



System Programs

- 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.
- 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

- **File management**
  - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some programs ask the system for information - date, time, amount of available memory, disk space, number of users
  - Others programs 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 – batch, time sharing, single user, multiuser, distributed, real-time
- Two groups in terms of defining goals:
  - User goals –should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals –should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Specifying and designing an OS is highly creative task of **software engineering**


Mechanisms and Polices

- 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)

Implementation

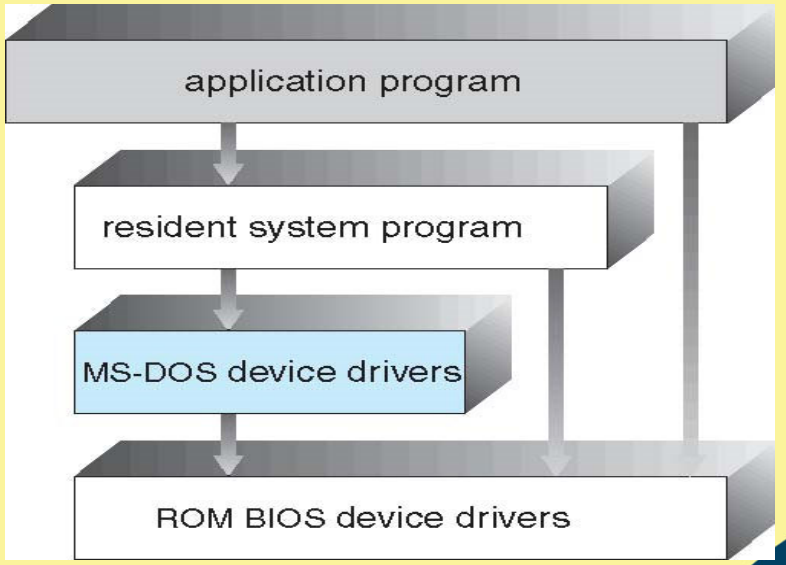
- Much variation
  - Early Operating Systems were written in assembly language
  - Then with system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL,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

- Various ways to structure an operating system:
  - Monolithic structure
    - Simple structure – MS-DOS
    - More complex – UNIX
    - More complex – Linux
  - Layered – An abstraction
  - Microkernel - Mach


MS-DOS

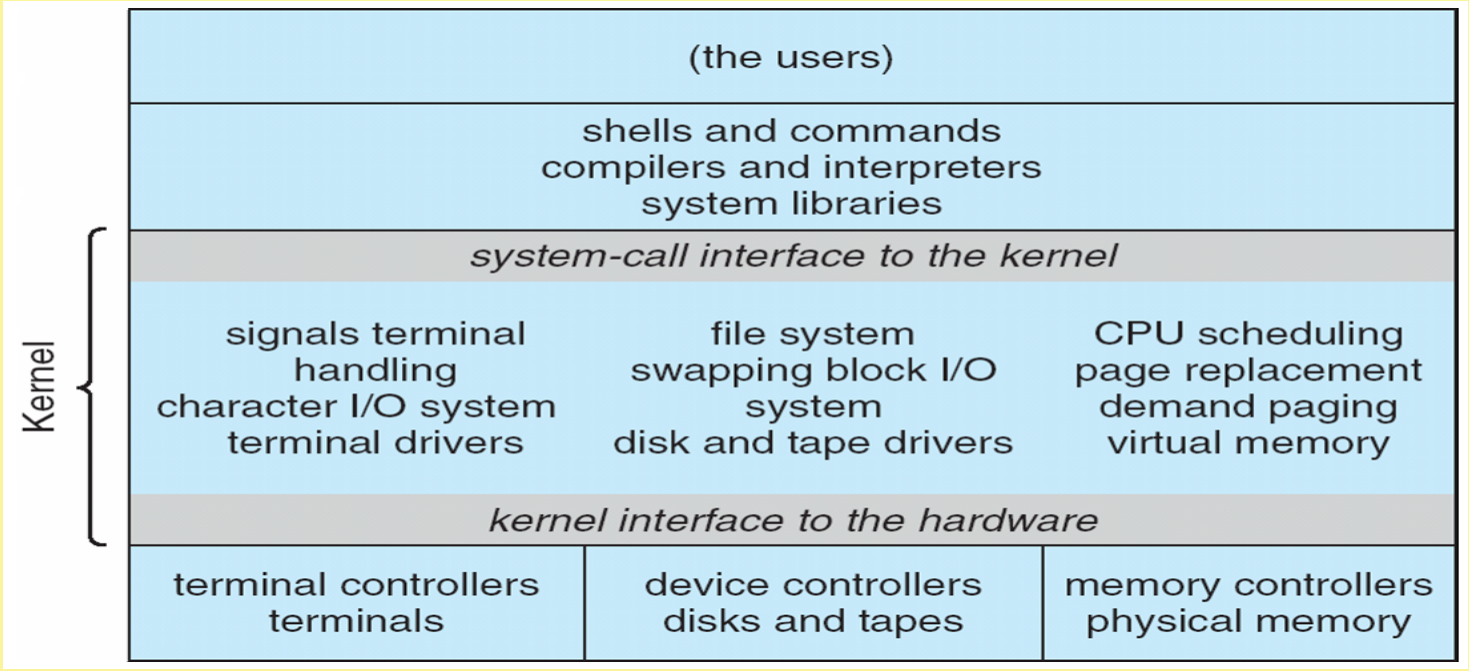
- MS-DOS – written to provide the most functionality in the least amount of space
- MS-DOS was limited by hardware funcionality.
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated



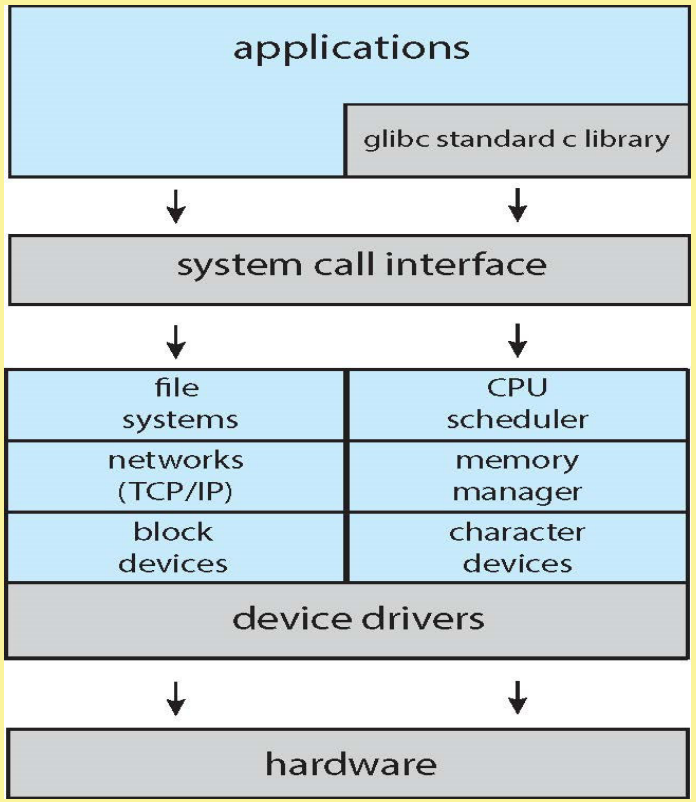
UNIX

- UNIX – the original UNIX operating system had limited structuring and was limited by hardware functionality.
- 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



Linux System Structure



Modularity

- The monolithic approach results in a situation where changes to one part of the system can have wide-ranging effects to other parts.
- Alternatively, we could design system where the operating system is divided into separate, smaller components that have specific and limited functionality. The sum of all these components comprises the kernel.
- Advantage: Changes in one component only affect that component, and no others, allowing system implementers more freedom when changing the inner workings of the system and in creating modular operating systems.