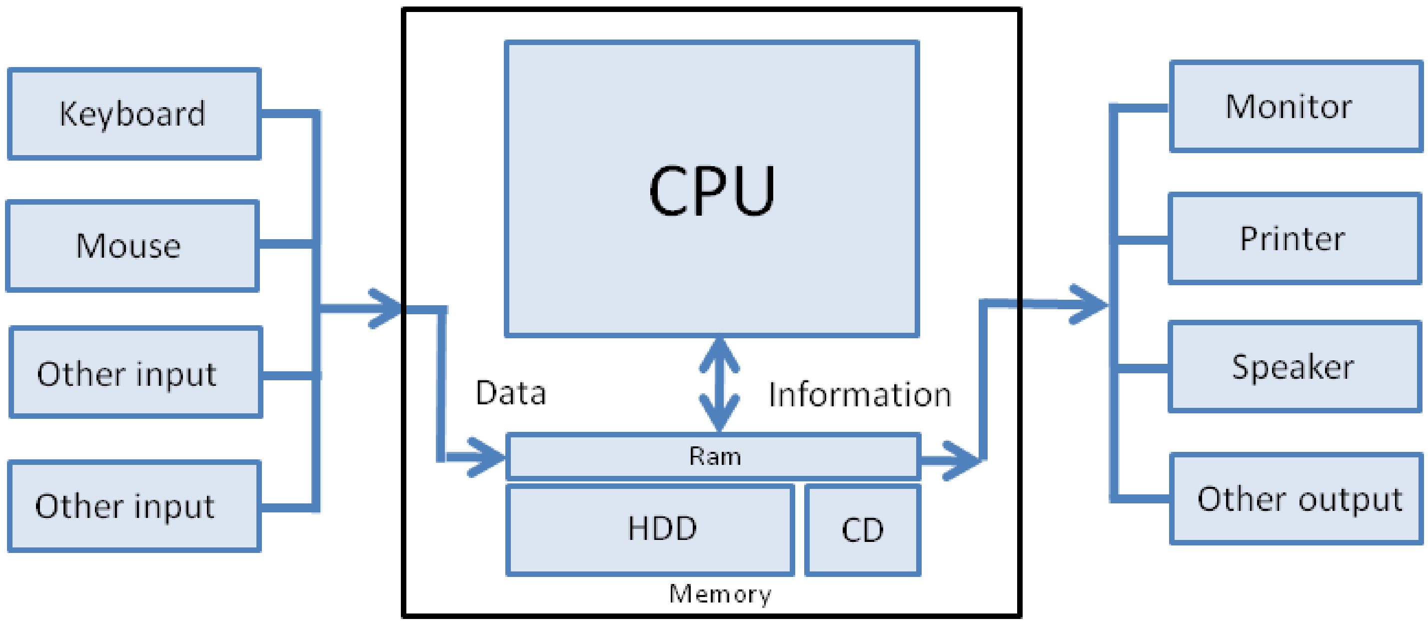
Operating System Structure



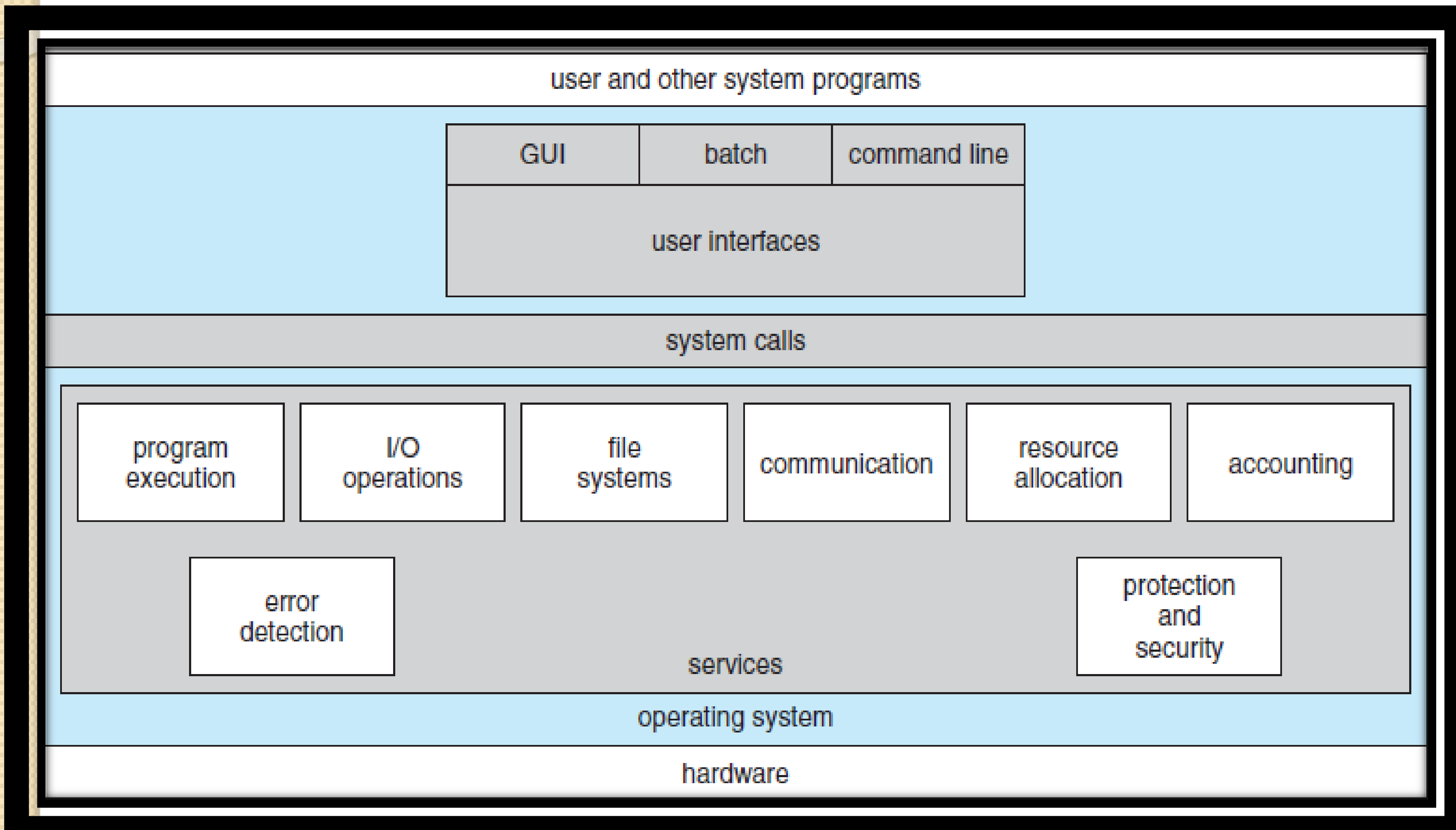
By
Meghana G Raj
Assistant Prof
IT Department

Contents

Objective

- To describe the services an operating system provides to users, processes, and other systems
- To discuss the various ways of structuring an operating system

OS services


user and other system programs
GUI
batch
command line
user interfaces
system calls
program execution
I/O operations
file systems
communication
resource allocation
accounting
error detection
protection and security
services
operating system
hardware

OS 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**

User Interface(1/3)

- A **CLI** (**command line interface**) is a user **interface** to a computer's operating system or an application in which the user responds to a visual **prompt** by typing in a **command** on a specified **line**, receives a response back from the system, and then enters another **command**, and so forth.

User Interface(2/3)

- Graphical User Interface(GUI)

1. Most commonly used interface in windows system.
2. Allows users to interact with electronic devices through graphical icons.
3. Choose from menu
4. Make selections along with keyboard to enter the text.

Comparison between GUI and CI

GUI

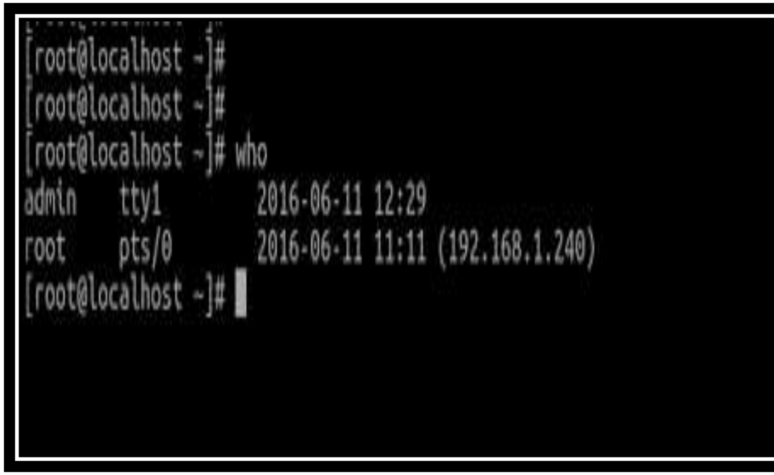
Interfacing

1. Commands in graphical form
2. Mouse is used to interact by clicking on specified icon.
3. No commands are used.

CI

Interfacing

1. Command prompt.
2. Keyboard is used to interact by typing specified commands.
3. Commands are used.



Comparison between GUI and CI

- *GUI*
  - **Control**

Although GUI offers control over file system and computer resources, often users have to use command line to complete specific task.

- *CI*
  - **Control**

Provides full access to computer resources.

Comparison between GUI and CI

- *GUI*
- **Control**

1. To launch the command line console (the black console box thingy), perform the following steps :
2. Open the *Start* menu
3. Click the *Run...* option

4. Type cmd /d into the text box and hit enter

This will launch the command console. The current directory will be shown before the prompt, it will look something like :
C:¥WINDOWS
You can now run command line programs by typing their name.

Comparison between GUI and CI

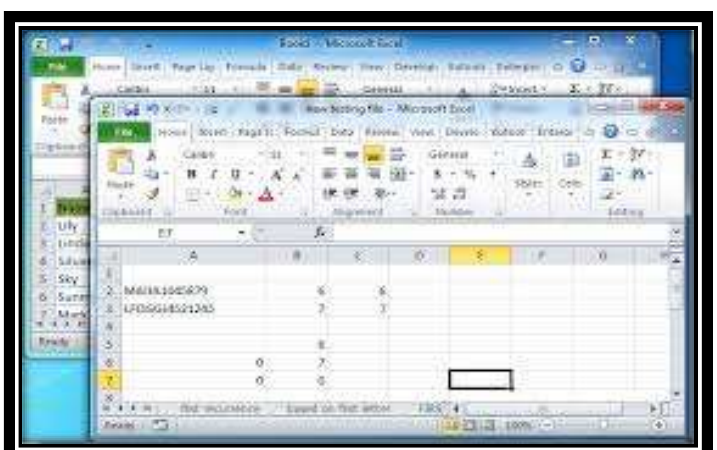
- *GUI*
  - **Complexity**
    Easy to learn and use.
  - **Speed**
    Slower to perform different tasks

- *CI*
  - **Complexity**
    Difficult to learn and use, involves remembering of commands.
  - **Speed**
    Faster compared to GUI in order to perform multiple task.

Comparison between GUI and CI

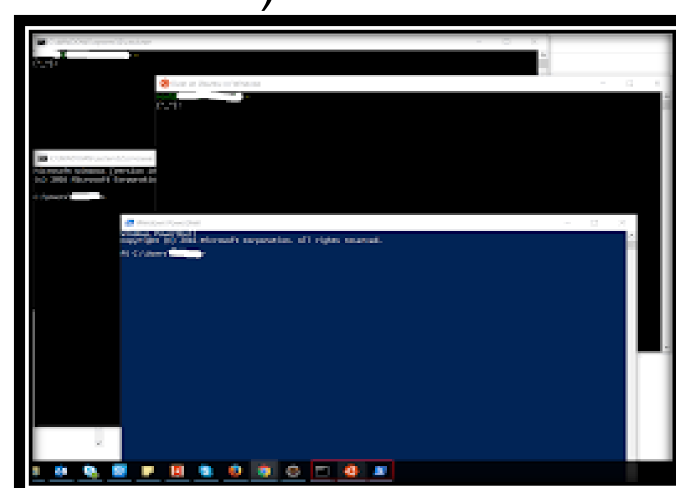
- *GUI*
  - **Multi-Tasking**

    Allows to open multiple programs in a separate window, so that user can view and manipulate many things.



- *CI*
  - **Multi-Tasking**

    Allows multi tasking, by use of different command interpreter, but difficult to view and track multiple activities(since various commands are involved in execution.)

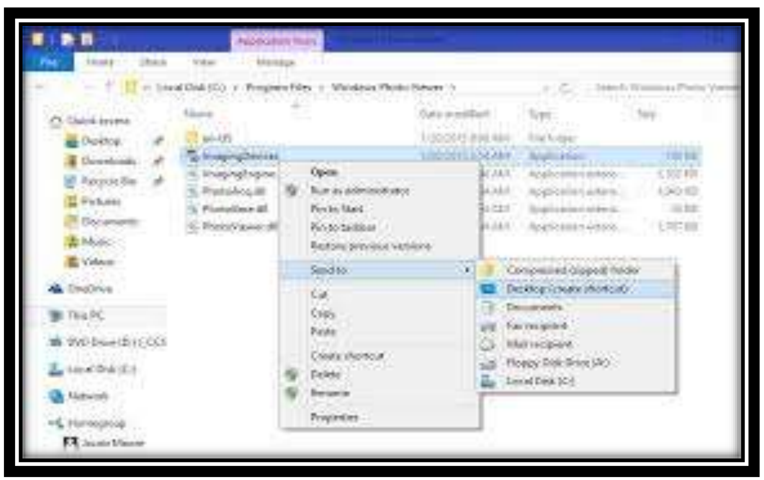


Comparison between GUI and CI

- ❑ *GUI*

➢ **<u>Scripting</u>**

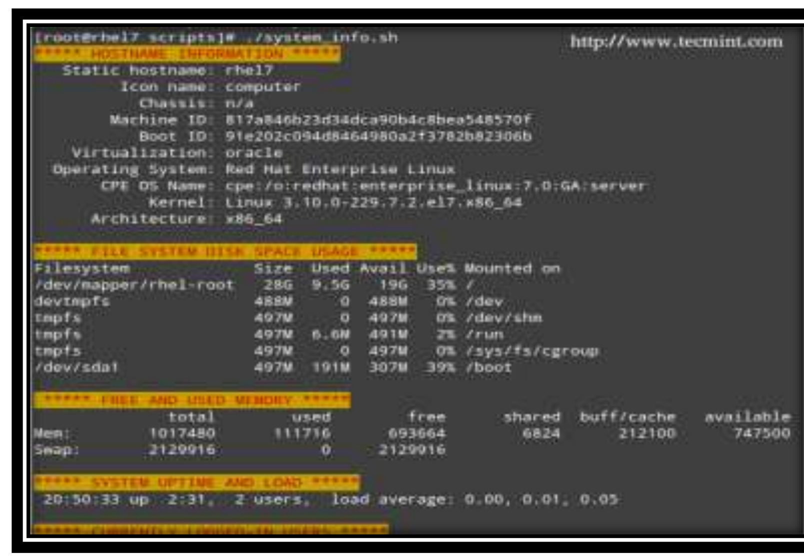
1. User can create shortcut to complete a task.
2. Cannot provide the facility of scripting a sequence of commands to perform a given task.



- ❑ *CI*

➢ **<u>Scripting</u>**

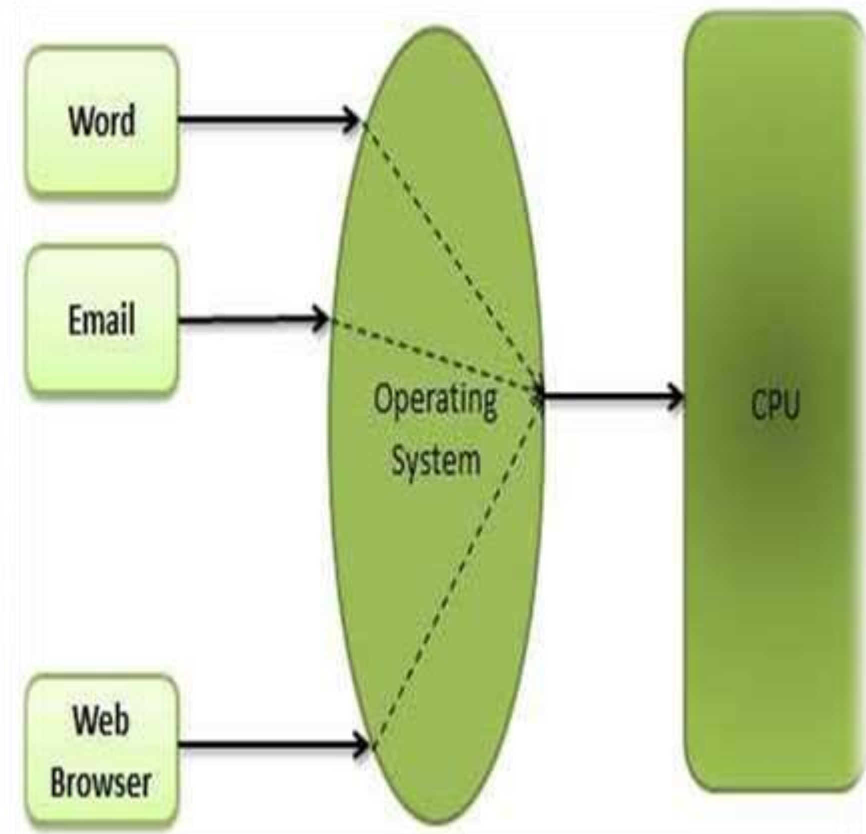
1. Can script sequence of commands to perform tasks.
2. Example: Use shell scripts to maintain: LINUX system maintenance tasks.



User Interface(3/3)

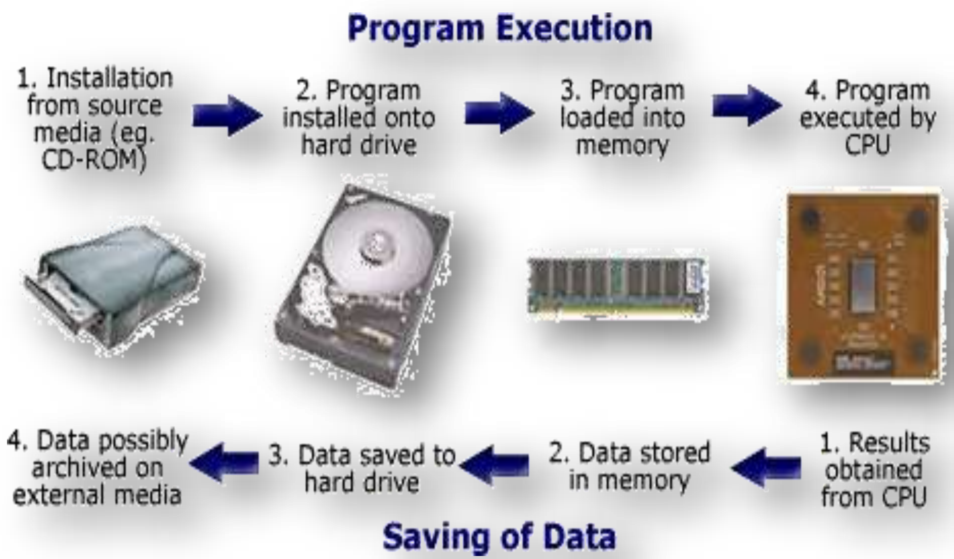
- Batch Interface

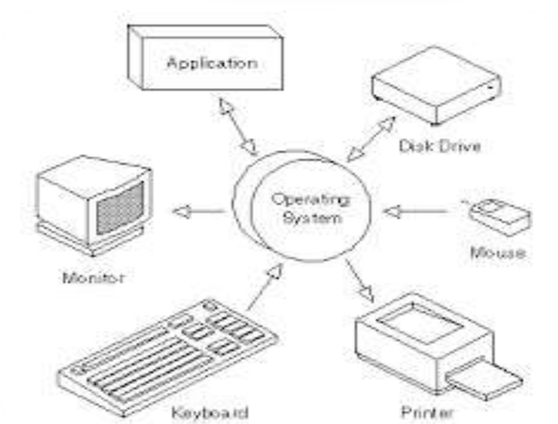
1. **Batch interfaces** are non-interactive user **interfaces**, where the user specifies all the details of the **batch** job in advance
   to **batch** processing, and receives the output when all the processing is done.
2. The computer does not prompt for further input after the processing has started.
3. Here its batch of inputs than a single input.



OS Services

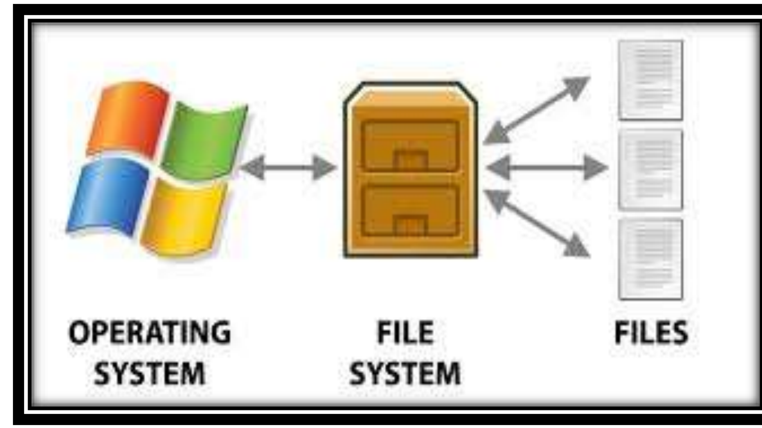
- **Program execution -** The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
- **Input/ Output operations -** A running program may require I/O, which may involve a file or an I/O device One set of operating-system services provides functions that are helpful to the user

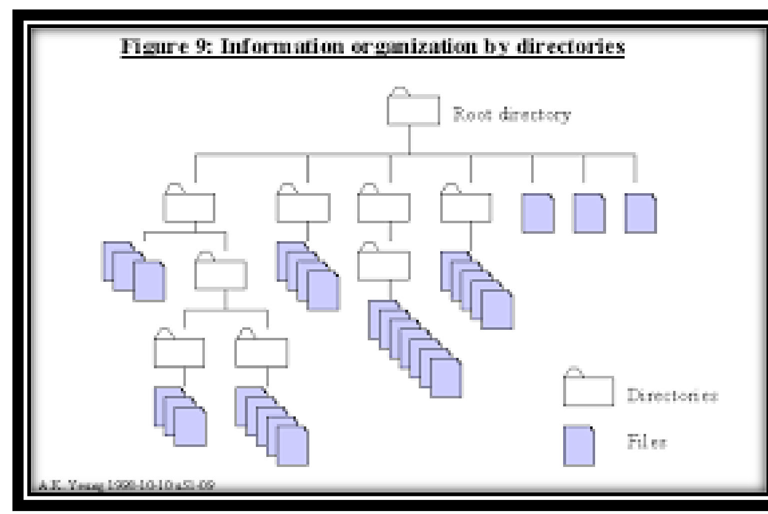




OS Services

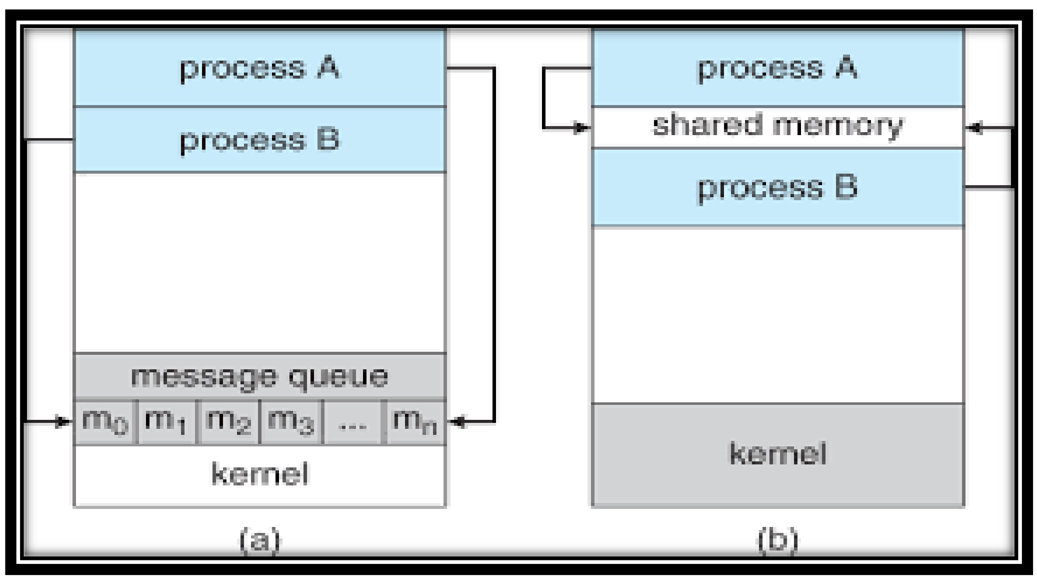
- **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management- to deny access to file and directories based on file ownership..





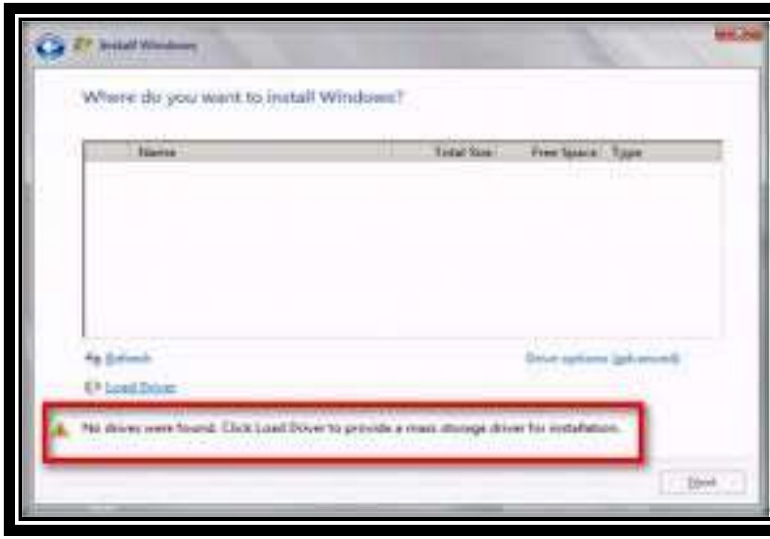
OS services

- **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)



OS services

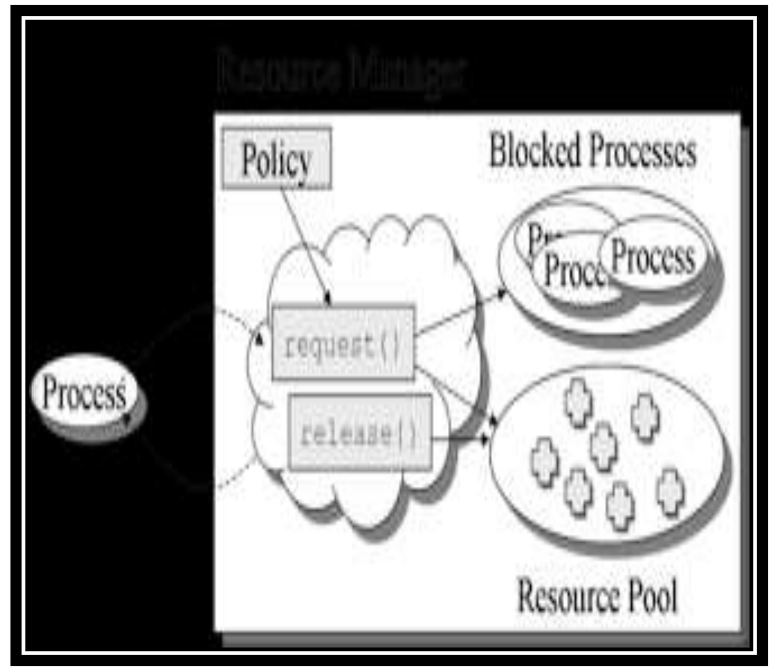
- **Error detection** – OS needs to be constantly aware of possible errors
  1. May occur in the CPU and memory hardware, in I/O devices, in user program
  2. For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  3. Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system



OS Services

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

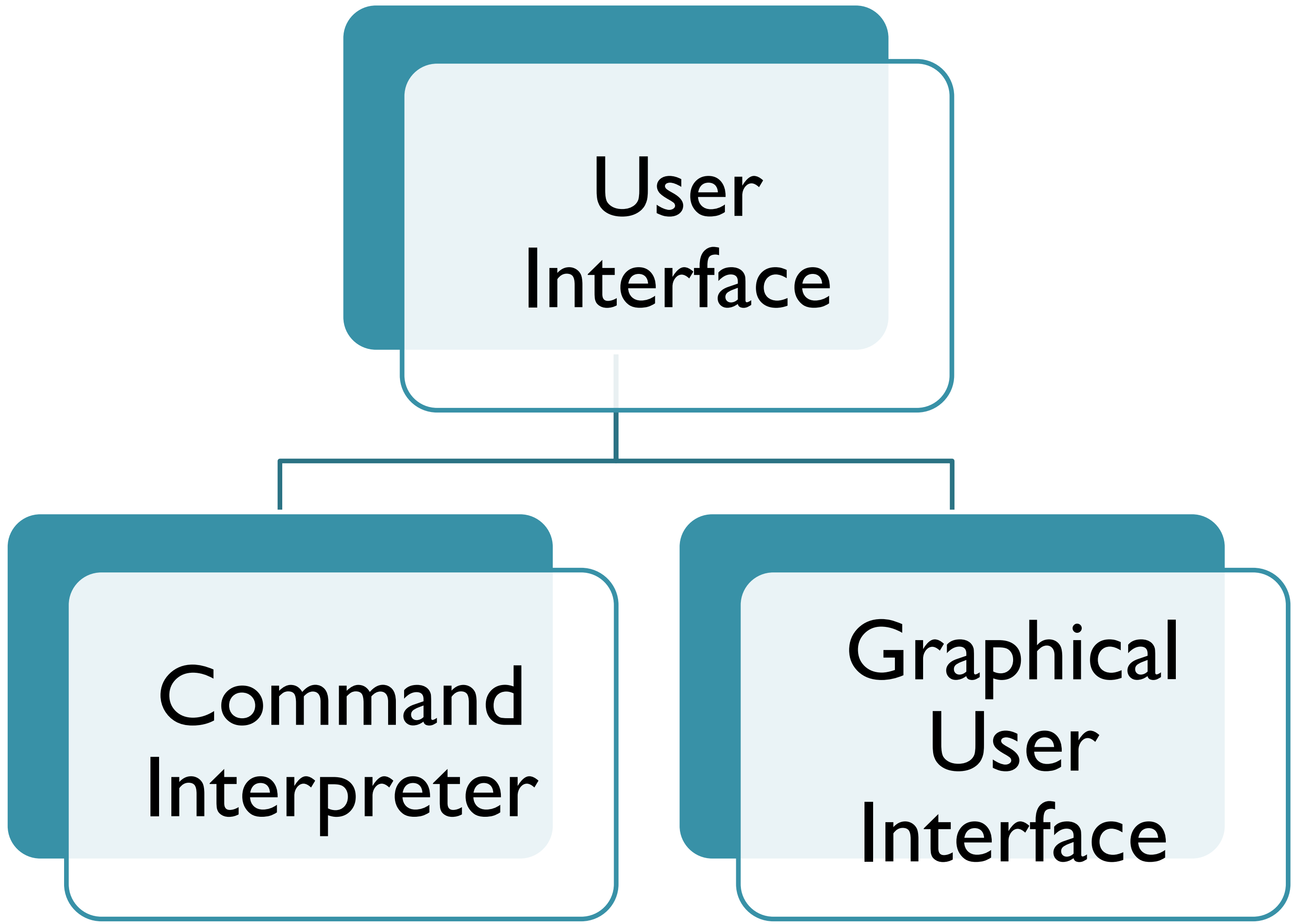
- **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
  - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
- **Accounting -** To keep track of which users use how much and what kinds of computer resources. Improves system efficiency.



OS Services

- **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

User OS interface



Command Interpreter(1/2)

- CLI or **command interpreter** allows direct command entry
  1. Sometimes implemented in kernel, sometimes by systems program
  2. Sometimes multiple flavors implemented – **shells(Bourne, C,Korn)**
  3. Primarily fetches a command from user and executes it
  4. Sometimes commands built-in, sometimes just names of programs

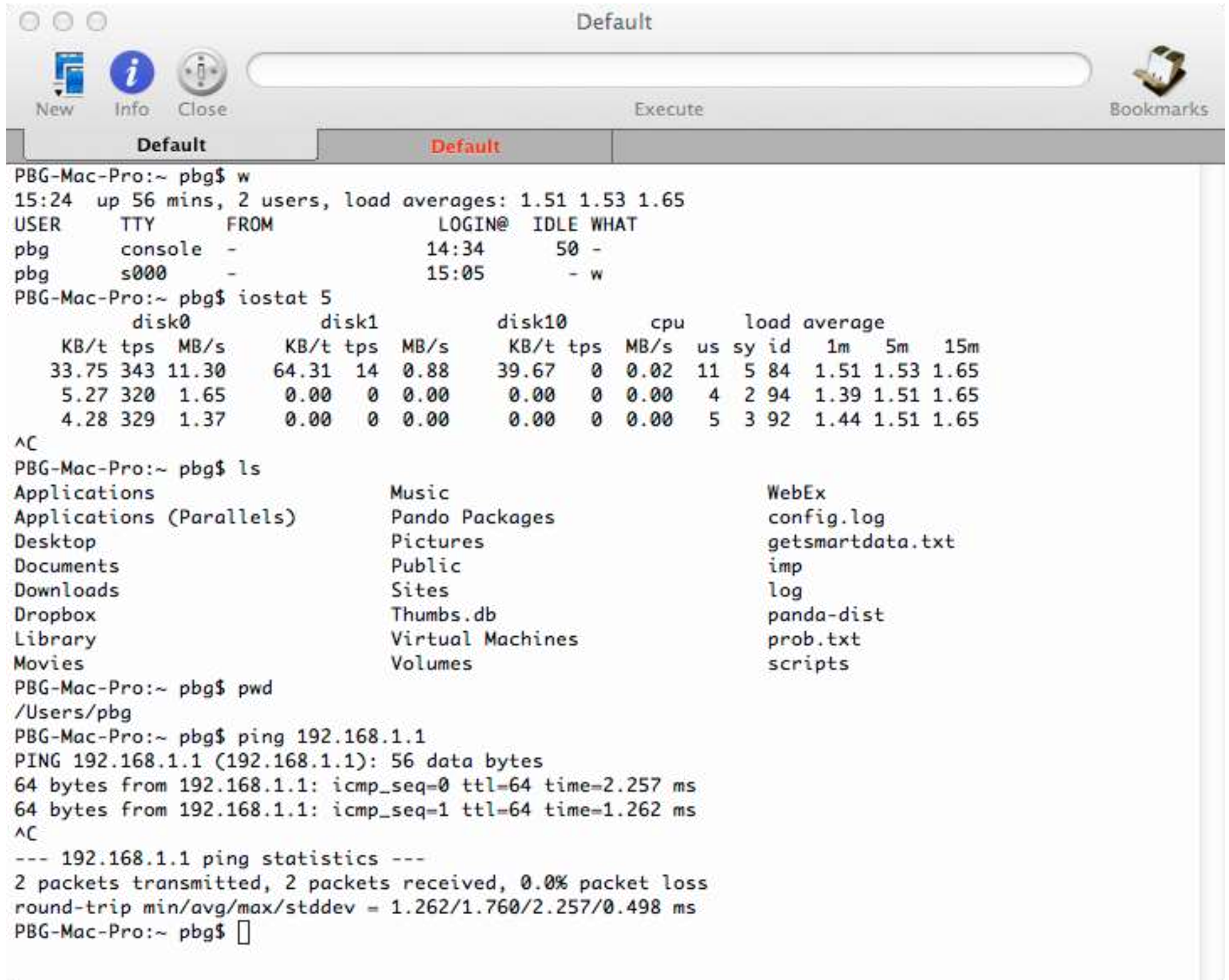
Command Interpreter(2/2)

- Commands are implemented in two ways:
  - Command interpreter itself has the code to execute the command.

    (Example: a delete file, this will result in command interpreter to go to a section of its code that sets up the parameters and makes appropriate system call. In this method the size of the command interpreter depends on number of commands that can be given.)
  - Alternative approach used by UNIX, is most commands are implemented through system programs. The command interpreter uses the command to identify a file to be loaded into memory and executed.

    (Example: rm file.txt would make the command interpreter search for a file rm, load that file into memory and execute it with parameter file.txt.)

Bourne shell command interpreter



Graphical User Interface

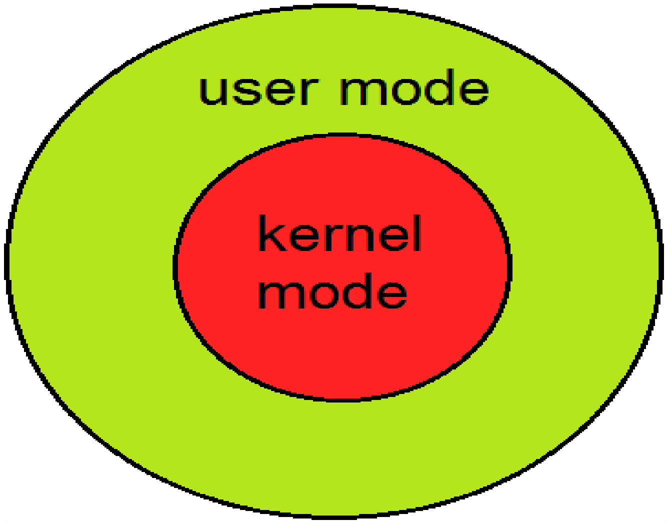
- User-friendly **desktop** metaphor interface
  1. Usually mouse, keyboard, and monitor
  2. **Icons** represent files, programs, actions, etc
  3. Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)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

Graphical User Interface

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

System Call

- To understand system calls, first one needs to understand the difference between **kernel mode** and **user mode** of a CPU. Every modern operating system supports these two modes.



Kernel and User Mode

- **Kernel Mode**

1. When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
2. Hence kernel mode is a very privileged and powerful mode.
3. If a program crashes in kernel mode, the entire system will be halted.

- **User Mode**

1. When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
2. In user mode, if any program crashes, only that particular program is halted.
3. That means the system will be in a safe state even if a program in user mode crashes.
4. Hence, most programs in an OS run in user mode.

System Call

- When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a **system call**.
- When a program makes a system call, the mode is switched from user mode to kernel mode. This is called a **context switch**.
- Then the kernel provides the resource which the program requested. After that, another context switch happens which results in change of mode from kernel mode back to user mode.
- Generally, system calls are made by the user level programs in the following situations:
- Creating, opening, closing and deleting files in the file system.
- Creating and managing new processes.
- Creating a connection in the network, sending and receiving packets.
- Requesting access to a hardware device, like a mouse or a printer.
- In a typical UNIX system, there are around 300 system calls.

Fork()

- The fork() system call is used to create processes. When a process (a program in execution) makes a fork() call, an exact copy of the process is created. Now there are two processes, one being the **parent** process and the other being the **child** process.
- The process which called the **fork()** call is the **parent** process and the process which is created newly is called the **child** process. The child process will be exactly the same as the parent. Note that the process state of the parent i.e., the address space, variables, open files etc. is copied into the child process. This means that the parent and child processes have identical but physically different address spaces. The change of values in parent process doesn't affect the child and vice versa is true too.
- Both processes start execution from the next line of code i.e., the line after the fork() call. Let's look at an example:

Fork()

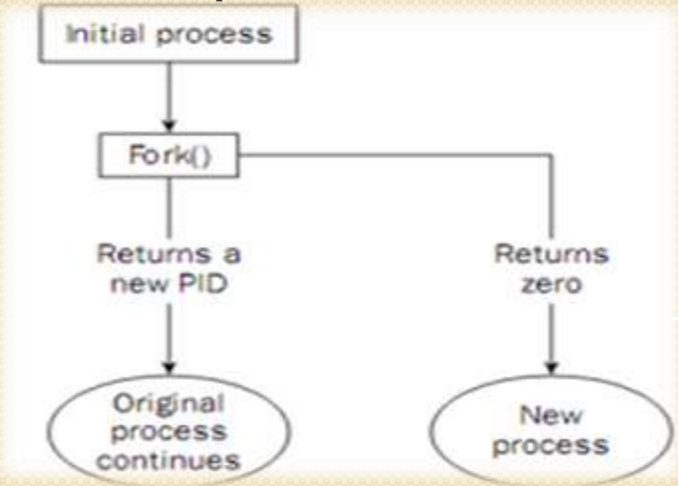
```
//example.c
 #include
 void main()
{ int val; val = fork(); // line A
printf("%d",val); // line B }
```

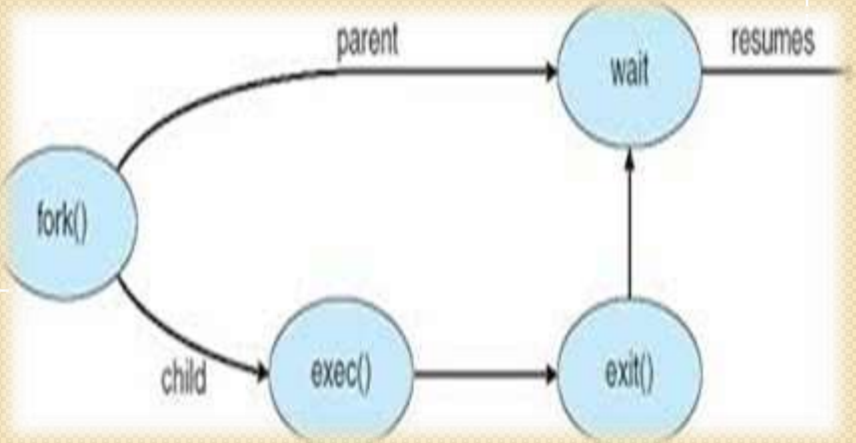
- When the above example code is executed, when **line A** is executed, a child process is created. Now both processes start execution from **line B**. To differentiate between the child process and the parent process, we need to look at the value returned by the fork() call.
- The difference is that, in the parent process, fork() returns a value which represents the **process ID** of the child process. But in the child process, fork() returns the value 0.
- This means that according to the above program, the output of parent process will be the **process ID** of the child process and the output of the child process will be 0.

Exec()

- The exec() system call is also used to create processes. But there is one big difference between **fork()** and **exec()** calls. The fork() call creates a new process while preserving the parent process. But, an exec() call replaces the address space, text segment, data segment etc. of the current process with the new process.
-

Fork()	Exec()
The **fork**() code return to tell you if you are the parent or the child. If you are the parent, the return is the id of the child. **fork**() creates a duplicate of the current process.	**exec**() replaces the program in the current process with another program.





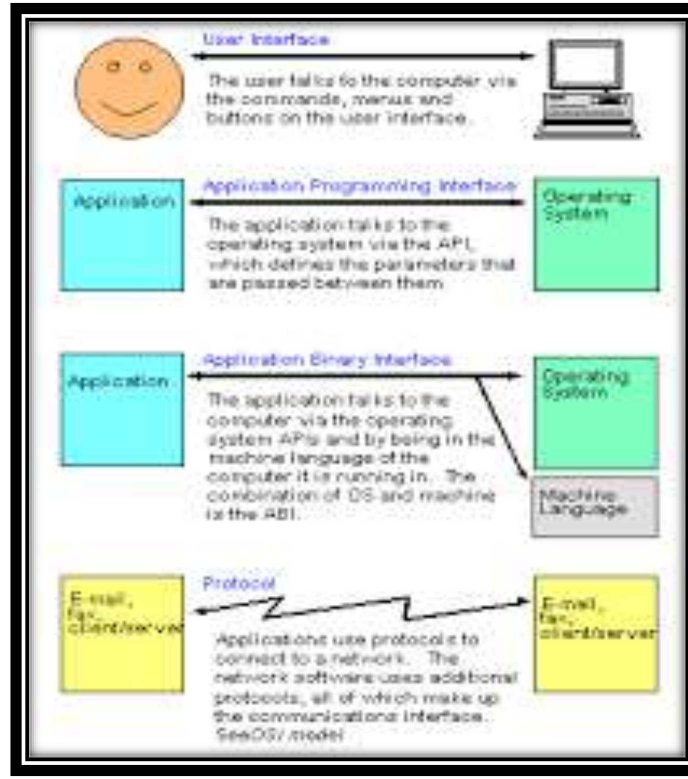
Comparison between fork and exec

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 (all versions of UNIX, Linux, and Mac OS X), and
  - Java API for the Java virtual machine (JVM)

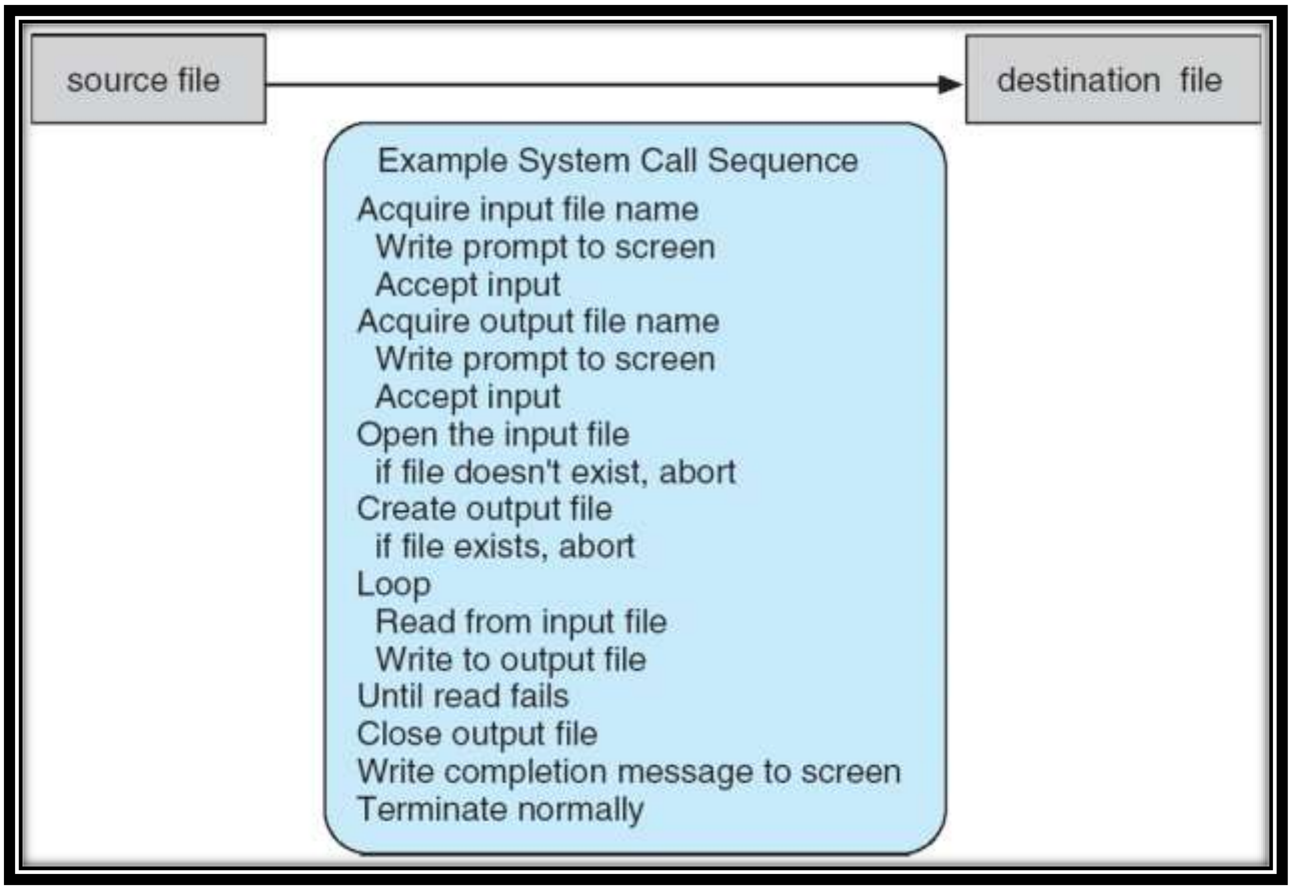
API

- An API is a set of commands, functions, protocols and objects that programmers can use to create software that interact with an external system.
- It provides developers with a standard commands for performing common operations so they do not have to write the code from scratch.



Example of System Call

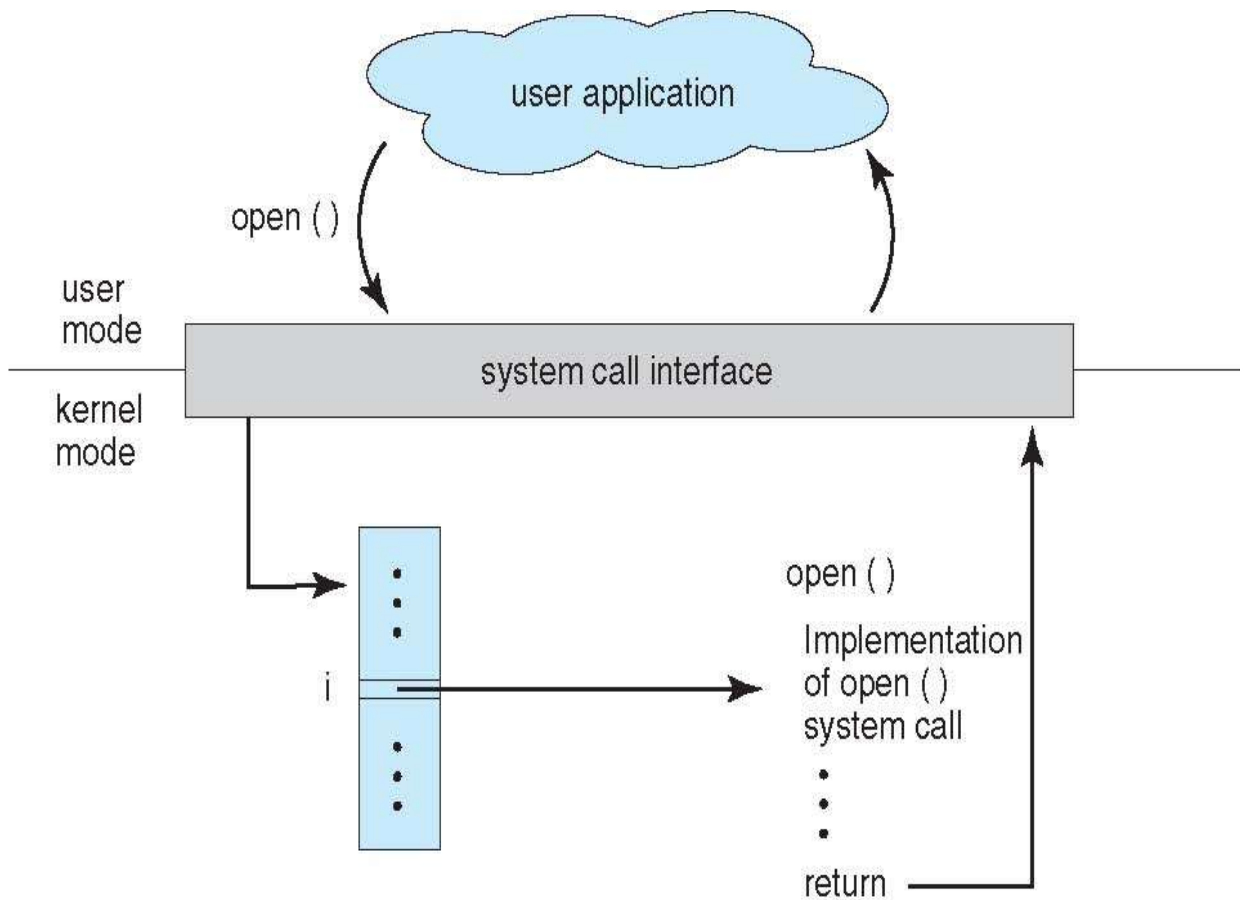
- System call sequence to copy the contents of one file to another file



Example of System Call

- The first input the program will need is the names of two files which can be specified in many ways.
- This sequence requires many I/O system calls.
- Next, the program must open the input file which requires another system call. If the opening of file fails, it should display error message on console(**another system call**) and should terminate abnormally(**another system call**).
- Next, the program must create the output file(**another system call**), it fails, it should display error message on console(**another system call**) and should also abort(**another system call**).
- Next we enter a loop that reads from input file(**system call**) and writes to the output file(**system call**). Write/read operation may fail, which needs (**another system call**) to continue.
- Finally, after the entire file is copied, the program may close both files(**system call**), and terminate normally(**system call**)

API – System Call – OS Relationship



System Call Interface

- This may include hardware-related services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling.
- **System calls** provide an essential **interface** between a process and the operating **system**.

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

- 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

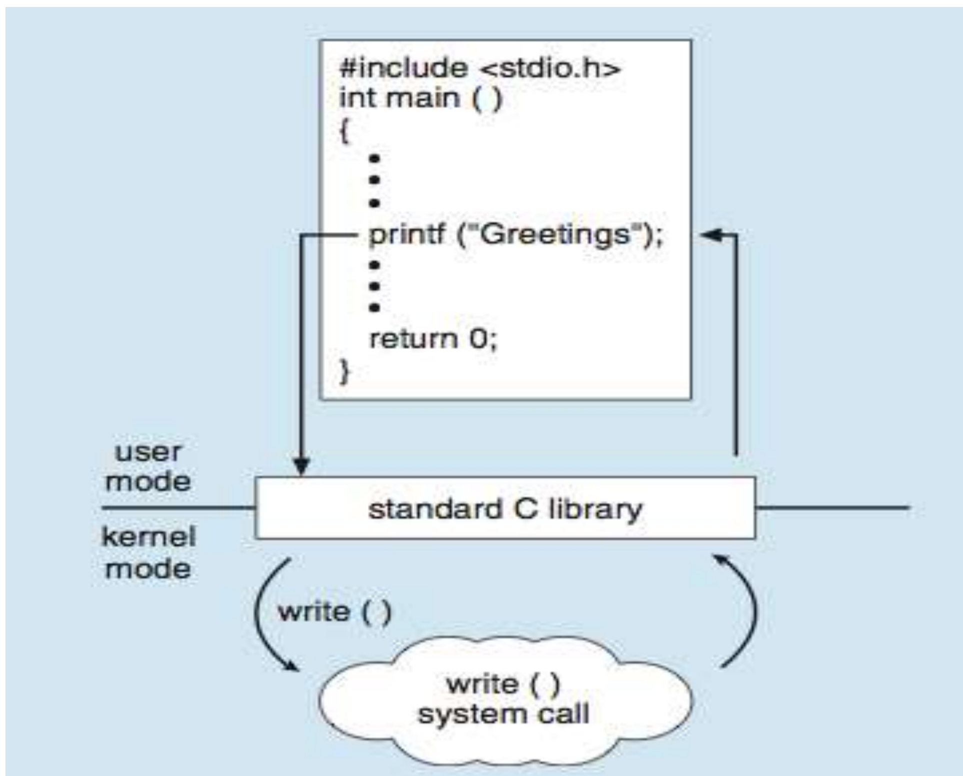
- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Examples of 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() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Standard C Library Example

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



- Thank You.