OPERATING SYSTEM

CHAPTER-1: INTRODUCTION

What is an Operating System?

A program that acts as an intermediary between a user of a computer and the computer hardware
Operating system goals:

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

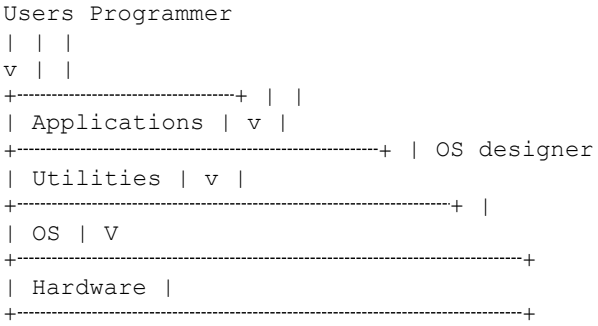
Operating System Overview:

This part aims to provide an overview of operating system principles and its history.We talk about the objectives of operating system first, then look at how operation systems evolve to get closer to those goals step by step, and finally summarize the major progresses that have been made till now.

1 Objectives and functions

1.1 OS as a user/computer interface - Usability

The reason for an operation system to exist is to make computers more convenient to use. An OS aims to wrap the underneath hardware resources and provides services to end users in a systematic way. These services may be divided into two types: services directly available for end users through all kinds of I/O devices, such as mouse, keyboard, monitor, printer, and so on; and services for application programs, which in turn provides services for end users. If we look on these services as interfaces among different components of a computer system, then the following hierarchical architecture may be obtained:



It is also common to consider Utilities and Applications that are distributed together with an OS parts of the OS, but obviously they are not essential. Utilities are usually called libraries or APIs, providing frequently used functions for the upper-level applications.

From the point of view of end users, a computer system consists of a variety of applications they may use. An application is developed by programmers in a programming language. The higher level the utilities are, the easier and more comfortable it is for programmers to code in the corresponding programming language; or the lower, the harder. In an extreme case, the assembly language is almost the same as machine instructions except that mnemonic symbols are used to replace binary strings or opcodes. In this kind of language, programmers have to deal with an overwhelmingly complexity of manipulating

computer hardware. On the contrary, in a higher-level language, more user-friendly APIs are available, e.g. opening a file by calling open("C:/filename.txt", "rw")

1.2 OS as resource manager – Efficiency

It is not the OS itself but the hardware that makes all kinds of services possible and available to application programs. An OS merely exploits the hardware to provide easily accessible interfaces. Exploitation means management upon the hardware resources, and thus also imposes control upon or manages the entities that use the services so that the resources are used efficiently. In the classes later on, we will discuss this aspect, including process scheduling, memory management, I/O device management, etc. One thing worth mentioning here is that, different from other control systems where the controlling facility, the controller, is distinct and external to the controlled parts, the OS has to depend on the hardware resources it manages to work.
As we know, an OS is in nature a program, consisting instructions, thus it also needs CPU to execute instructions so as to function as a controller, and main memory to hold instructions for CPU to fetch. At the same time, the OS has to be able to relinquish and regain later the control of CPU so that other programs can get chance to run but still under the control of the OS (An analogy to this is that an administrator of an apartment building might live in the same building himself). By utilizing the facilities provided by hardware, the OS may schedule different processes to run at different moments and exchange the instructions and data of programs between external storage devices, like hard disks, and main memory. These topics will be covered as the course proceeds.

1.3 Evolution of OS – Maintainability

It does not suffice to simply consider an operating system an unvariable unit. An OS may
evolve while time elapses due to the following reasons:
• hardware upgrades or new types of hardware: With hardware technologies development, the OS also needs to upgrade so as to utilize the new mechanisms introduced by new hardware. For example, Pentium IV extended instruction set of Pentium III for multimedia applications and internet transmission. An OS designed for the previous versions of Intel x86 series will have to be upgraded to be able to accommodate these new instructions.
• new services: An OS may also expand to include more services in response to user demand.
• fixes: No software is perfect, and any program may contain more or less bugs or defects, thus fixes should be made from time to time. Microsoft Windows is a vivid example of this kind. These situations all require OS designers to build an OS in the way that the system can be maintained and upgraded easily. All the common software design techniques may be applied to an OS, such as modularization. With modularization, the OS is split into multiple modules with clearly defined interfaces between them. Thus, as long as the interfaces are left untouched, each single module may be upgraded independently.

2. The evolution of operating systems

To better understand the requirements for an operating system, it is useful to consider how operating systems have evolved over the years.

2.1 Serial processing

The earliest computer system has no OS at all, and is characterized as serial processing because users have to reserve time slots in advance, and during the alloted period, they occupy the computer

exclusively. Thus the computer will be used in sequence by different users. These early systems presented two major problems:

1. Users may finish their tasks earlier than you have expected, and unfortunately the rest time is simply wasted. Or they may run into problems, cannot finish in the allotted time, and thus are forced to stop, which causes much inconvenience and delays the development.
2. In such systems, programs are presented by cards. Each card has several locations on it, where there may be a hole or not, respectively indicating 0 or 1. Programs are loaded into memory via a card reader. With no OS available, to compile their programs, users have to manually load the compiler program first with the user program as input. This involves mounting, or dismounting tapes or setting up card decks. If an error occurred, the user has to repeat the whole process from the very beginning. Thus much time is wasted.

2.2 Simple batch systems

To improve the utilization of computer systems, the concept of a batch operating system was developed later on. The central idea is the use of a piece of software known as the monitor. With it, users don't have direct access to the computer systems any longer; instead, the operator of the system collects the user programs and batches them together sequentially for use by the monitor.
To process user programs, the monitor first has to be loaded into memory. Then it reads in programs one at a time from the input devices. As each program is read in, it will be placed in the user program area of main memory, and control is passed to this program. When the execution of the program is completed, it returns control to the monitor, which moves on to process the next program.
To assist in this, a special language is used, called job control language or JCL (At that time, each user program was called a job), which provides instructions to the monitor. For example, the following is a FORTRAN program with JCL commands.

```
$JOB
$FTN
...
FORTRAN instructions
...
$RUN
...
Data
...
$END
```

$JOB indicates the beginning of a job. $FTN tells the monitor to load the FORTRAN compiler, which generates object code from the FORTRAN source program. The monitor regains control after the compile operation. $RUN makes control transferred from the monitor to the current program, which works on the data following until a successful or unsuccessful completion. The monitor then may move on to another job. To much extent, JCL instructions are similar to the command sequence in a DOS batch file or a UNIX shell file. The latters tell the operating systems to run multiple commands automatically.
The advantage of this mode is that the monitor automates the execution of multiple jobs thus much time is saved by avoiding manual operations.

2.3 Multiprogrammed batch systems

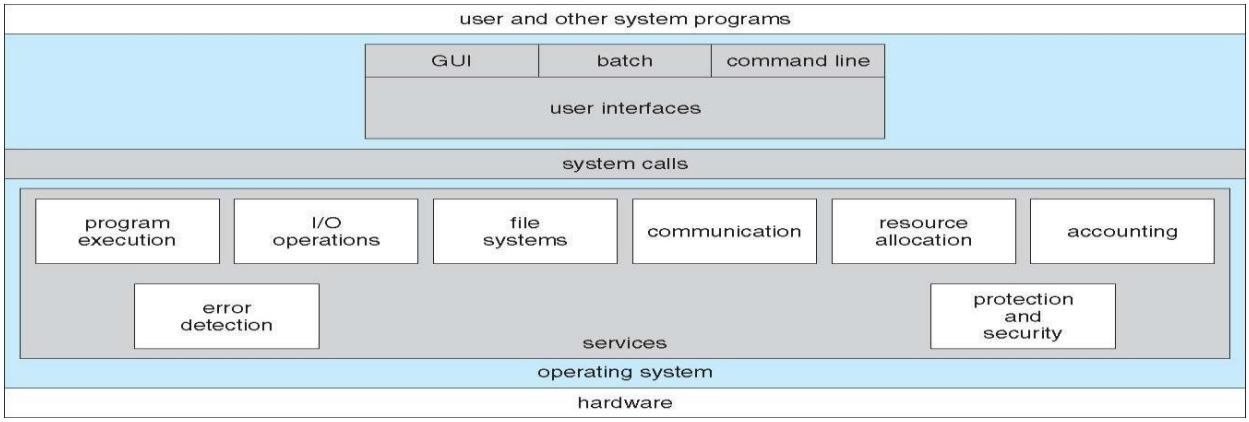
Even with the automatic job processing by a monitor, the processor is still often idle. The problem is actually what we have discussed before regarding programmed I/O. That is a program may have to wait for I/O operation to finish and thus leads to the processor's idling. The solution is to run multiple programs concurrently during a certain period so that whenever the current program has to wait for I/O devices, control may be transferred to another program. If needed, a third program may be loaded, or even more. This scheme is called multiprogramming or multitasking.
With multiprogramming, the utilization of processor is greatly improved, but it has its own problems. To run multiple programs concurrently, the memory should be organized properly so that each program has

its own space and does not invade others'. What's more, at some moment, there may be more than programs ready to run. Thus some form of scheduling is needed to obtain better performance.

2.4 Time-sharing system

With multiprogramming, the overall system is quite efficient. However a problem remains. That is those jobs that come late in the batch job list won't get chance to run until the jobs before them have completed, thus their users have to wait a long time to obtain the results. Some programs may even need interaction with users, which requires the processor to switch to these programs frequently. To reach this new goal, a similar technique to multiprogramming can be used, called time sharing. In such a system, multiple users simultaneously access the system through terminals, with the operating system interleaving the execution of each user program in a short burst of computation. For example, suppose a computer system may have at most 10 users at the same time, and the human reaction time is 200 ms. Then we may assign 200/10 = 20ms CPU time to the user programs one by one in a cyclic manner, thus each user will be responded within the human reaction time so that the computer system seems to service the user program itself. The following table gives the difference between the batch multiprogramming and time sharing:

A View of Operating System Services



Operating System Services

One set of 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 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 .

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 - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may 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 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

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 Operating System 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 – 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

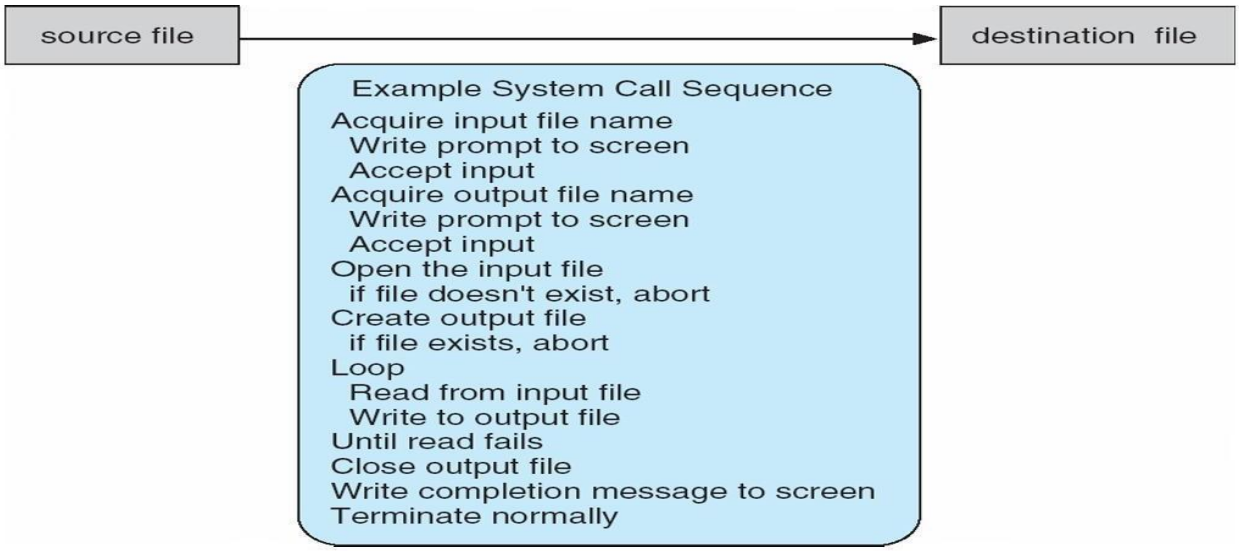
User Operating System Interface - GUI

- 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
- Many systems now include both CLI and GUI interfaces
- Microsoft Windows is GUI with CLI "command" shell
- Apple Mac OS X as "Aqua" GUI interface with UNIX kernel underneath and shells available
- Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

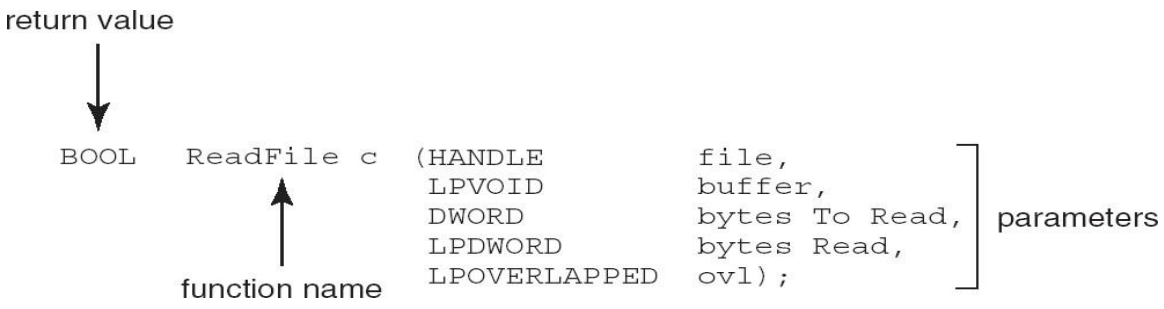
System Calls

- 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 direct system call usenThree 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?(Note that the system-call names used throughout this text are generic)

Example of System Calls



Example of Standard API
Consider the ReadFile() function in the
Win32 API—a function for reading from a file



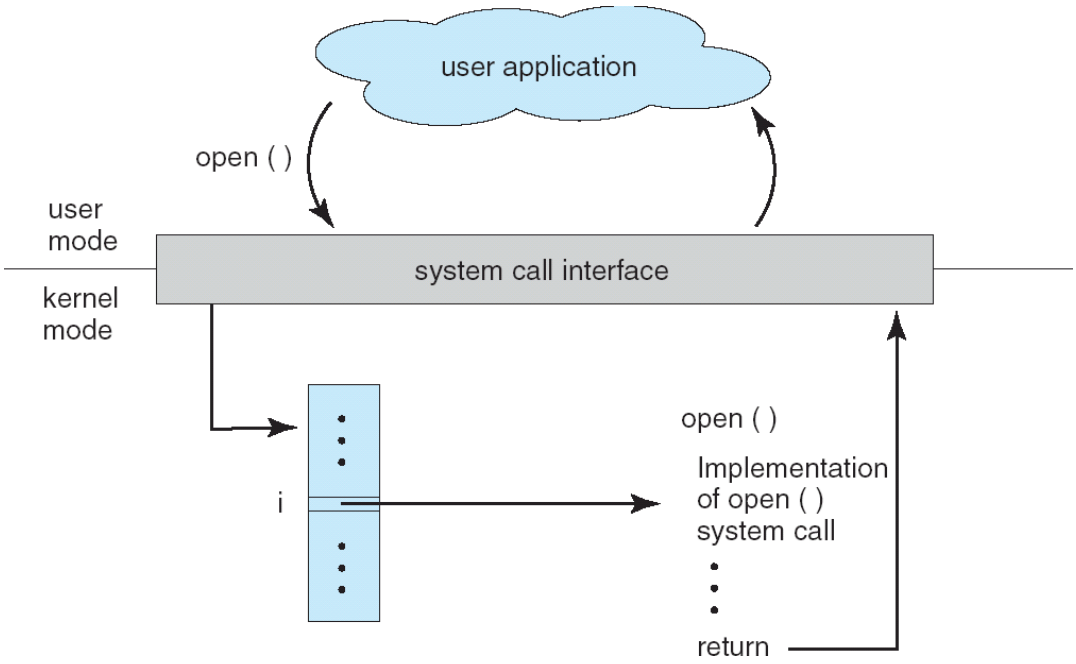
A description of the parameters passed to ReadFile()

- HANDLE file—the file to be read
- LPVOID buffer—a buffer where the data will be read into and written from
- DWORD bytesToRead—the number of bytes to be read into the buffer
- LPDWORD bytesRead—the number of bytes read during the last read
- LPOVERLAPPED ovl—indicates if overlapped I/O is being used

System Call Implementation

- Typically, a number associated with each system call
- System-call interface maintains a table indexed according to these
- Numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
- Just needs to obey API and understand what OS will do as a result call
- Most details of OS interface hidden from programmer by API Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



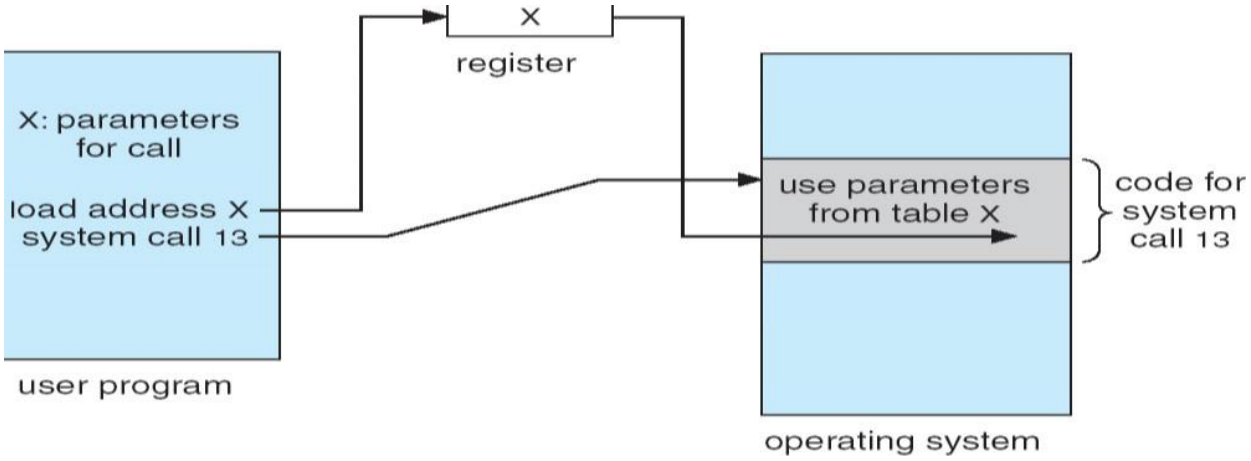
System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
- Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
- Simplest: pass the parameters in *registers*
  - In some cases, may be more parameters than registers
- Parameters stored in a *block*, or table, in memory, and address of block passed as a parameter in a register

- This approach taken by Linux and Solaris
- Parameters placed, or *pushed*, onto the *stack* by the program and *popped* off the stack by the operating system

- Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications

1. Process Control:

End, abort: A running program needs to be able to has its execution either normally (end) or abnormally (abort).

Load, execute:A process or job executing one program may want to load and executes another program. Create Process, terminate process: There is a system call specifying for the purpose of creating a new process or job (create process or submit job). We may want to terminate a job or process that we created (terminates process, if we find that it is incorrect or no longer needed). Get process attributes, set process attributes: If we create a new job or process we should able to control its execution. This control requires the ability to determine & reset the attributes of a job or processes (get process attributes, set process attributes).

Wait time: After creating new jobs or processes, we may need to wait for them to finish their execution (wait time). Wait event, signal event: We may wait for a specific event to occur (wait event). The jobs or processes then signal when that event has occurred (signal event).

2. File Manipulation:

Create file, delete file: We first need to be able to create & delete files. Both the system calls require the name of the file & some of its attributes.

Open file, close file: Once the file is created, we need to open it & use it. We close the file when we are no longer using it.

Read, write, reposition file: After opening, we may also read, write or reposition the file (rewind or skip to the end of the file).

Get file attributes, set file attributes: For either files or directories, we need to be able to determine the values of various attributes & reset them if necessary. Two system calls get file attribute & set file attributes are required for their purpose.

3. Device Management:

Request device, release device: If there are multiple users of the system, we first request the device. After we finished with the device, we must release it.

Read, write, reposition: Once the device has been requested & allocated to us, we can read, write & reposition the device.

4. Information maintenance:

Get time or date, set time or date: Most systems have a system call to return the current date & time or set the current date & time.

Get system data, set system data: Other system calls may return information about the system like number of current users, version number of OS, amount of free memory etc.

Get process attributes, set process attributes: The OS keeps information about all its processes & there are system calls to access this information.

5. Communication: There are two modes of communication such as:

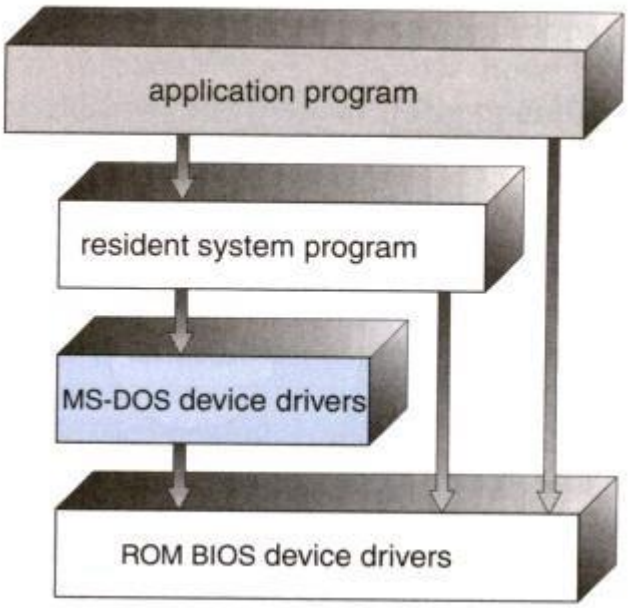
Message passing model: Information is exchanged through an inter process communication facility provided by operating system. Each computer in a network has a name by which it is known. Similarly, each process has a process name which is translated to an equivalent identifier by which the OS can refer to it. The get hostid and get processed systems calls to do this

translation. These identifiers are then passed to the general purpose open & close calls provided by the file system or to specific open connection system call. The recipient process must give its permission for communication to take place with an accept connection call. The source of the communication known as client & receiver known as server exchange messages by read message & write message system calls. The close connection call terminates the connection.

Shared memory model: processes use map memory system calls to access regions of memory owned by other processes. They exchange information by reading & writing data in the shared areas. The processes ensure that they are not writing to the same location simultaneously.

Operating-System Structure

1. Simple Structure



MS-DOS, operating system.

Many commercial systems do not have well-defined structures. Frequently, such operating systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular. It was written to provide the most functionality in the least space, so it was not divided into modules carefully. Figure shows its structure. In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also

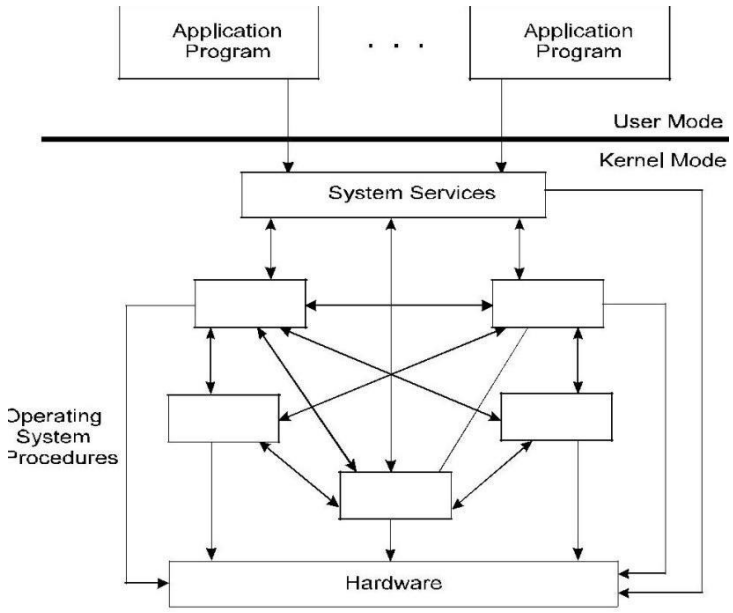
limited by the hardware of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.

Operating systems such as MS-DOS and the original UNIX did not have well-defined structures.

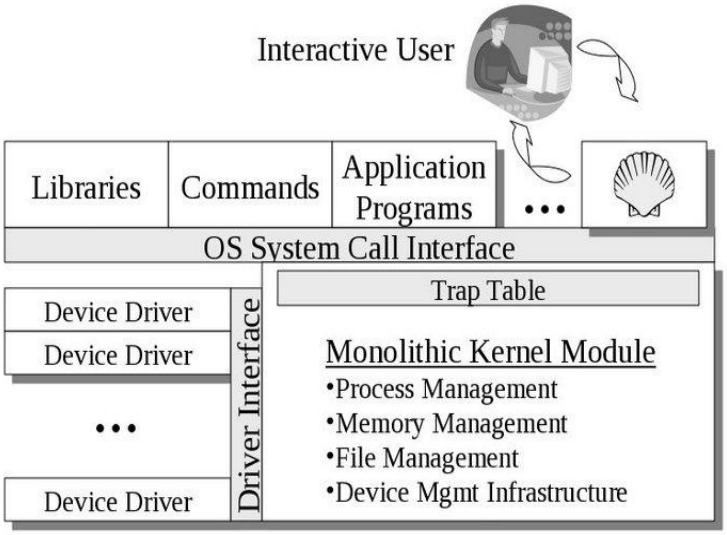
- There is no CPU Execution Mode (user and kernel), and so errors in applications can cause the whole system to crash.

2. Monolithic Approach

The components of monolithic operating system are organized haphazardly and any module can call any other module without any reservation. Similar to the other operating systems, applications in monolithic OS are separated from the operating system itself. That is, the operating system code runs in a privileged processor mode (referred to as kernel mode), with access to system data and to the hard-ware; applications run in a non-privileged processor mode (called the user mode), with a limited set of interfaces available and with limited access to system data. The monolithic operating system structure with separate user and kernel processor mode is shown in figure.



- Functionality of the OS is invoked with simple function calls within the kernel, which is one large program.
- Device drivers are loaded into the running kernel and become part of the kernel.



A monolithic kernel, such as Linux and other Unix systems.

When a user-mode program calls a system service, the processor traps the call and then switches the calling thread to kernel mode. Completion of system service, switches the thread back to the user mode, by the operating system and allows the caller to continue.

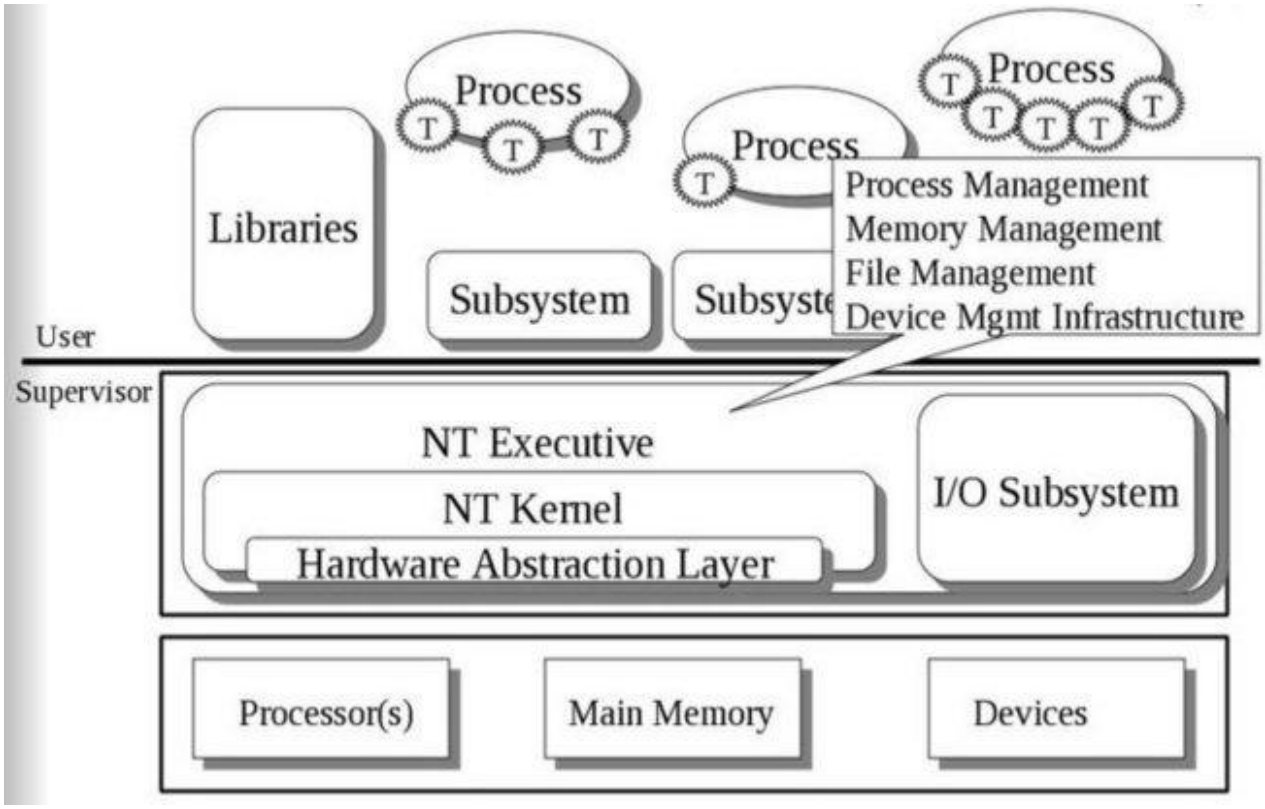
The monolithic structure does not enforce data hiding in the operating system. It delivers better application performance, but extending such a system can be difficult work because modifying a proce-dure can introduce bugs in seemingly unrelated parts of the system.

Example Systems: CP/M and MS-DOS

3. Layered Approach

This approach breaks up the operating system into different layers.

- This allows implementers to change the inner workings, and increases modularity.
- As long as the external interface of the routines don't change, developers have more freedom to change the inner workings of the routines.
- With the layered approach, the bottom layer is the hardware, while the highest layer is the user interface.
    - The main *advantage* is simplicity of construction and debugging.
    - The main *difficulty* is defining the various layers.
    - The main *disadvantage* is that the OS tends to be less efficient than other implementations.



The Microsoft Windows NT Operating System. The lowest level is a monolithic kernel, but many OS components are at a higher level, but still part of the OS.

The components of layered operating system are organized into modules and layers them one on top of the other. Each module provide a set of functions that other module can call. Interface functions at any particular level can invoke services provided by lower layers but not the other way around. The layered operating system structure with hierarchical organization of modules is shown in Figure.

One advantage of a layered operating system structure is that each layer of code is given access to only the lower-level interfaces (and data structures) it requires, thus limiting the amount of code that wields unlimited power. That is in this approach, the Nth layer can access services provided by the (N-1)th layer and provide services to the (N+1)th layer. This structure also allows the operating system to be debugged starting at the lowest layer, adding one layer at a time until the whole system works correctly. Layering also makes it easier to enhance the operating system; one entire layer can be replaced without affecting other parts of the system. Layered operating system delivers low application performance in comparison to monolithic operating system.

Example Systems: VAX/VMS, Multics, UNIX

4. Microkernels

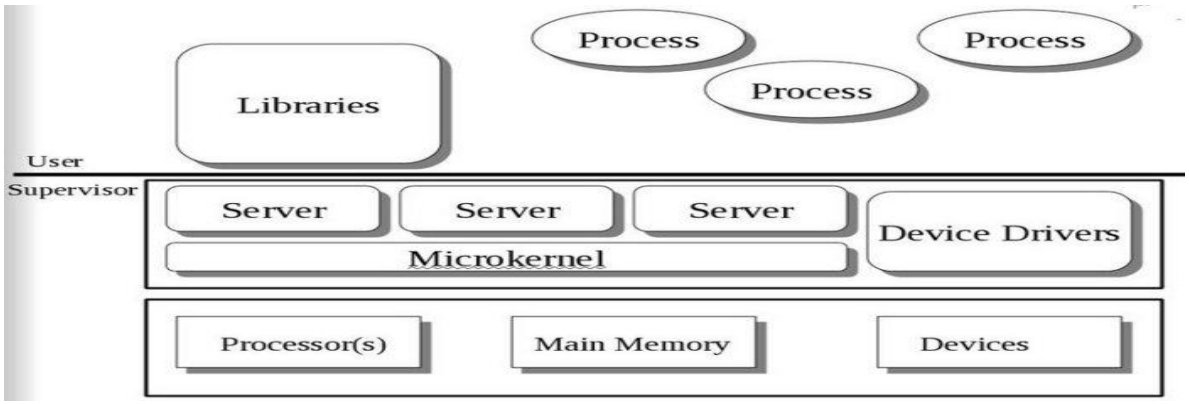
This structures the operating system by removing all nonessential portions of the kernel and implementing them as system and user level programs.

- Generally they provide minimal process and memory management, and a communications facility.
- Communication between components of the OS is provided by message passing.

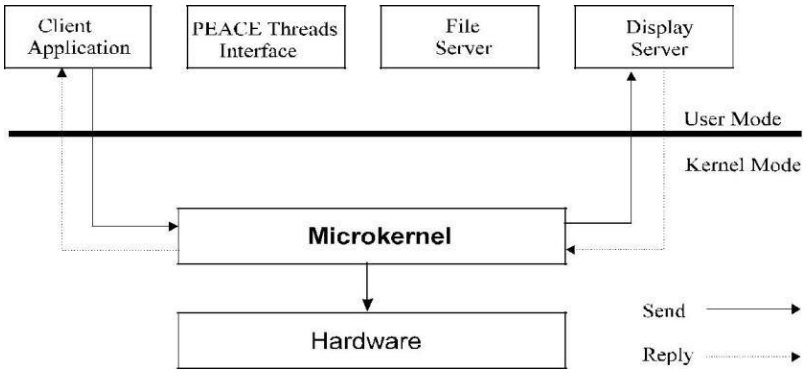
The *benefits* of the microkernel are as follows:

- Extending the operating system becomes much easier.
- Any changes to the kernel tend to be fewer, since the kernel is smaller.
- The microkernel also provides more security and reliability.

Main *disadvantage* is poor performance due to increased system overhead from message passing.



A Microkernel architecture.



The advent of new concepts in operating system design, microkernel, is aimed at migrating traditional services of an operating system out of the monolithic kernel into the user-level process. The idea is to divide the operating system into several processes, each of which implements a single set of services - for example, I/O servers, memory server, process server, threads interface system. Each server runs in user mode, provides services to the requested client. The client, which can be either another operating system component or application program, requests a service by sending a message to the server. An OS kernel (or microkernel) running in kernel mode delivers the message to the appropriate server; the server performs the operation; and microkernel delivers the results to the client in another message, as illustrated in Figure

In general, services that were traditionally an integral part of the file systems, windowing systems, security services, etc. are becoming peripheral modules that interact with the kernel and each other known as subsystem. The microkernel approach replaces the vertical stratification of operating system functions with a horizontal one. Components above the microkernel communicate directly with one another, although using messages that pass through the microkernel itself. The microkernel plays a traffic cop. It validates messages, passes them between the components, and grants access to hard-ware.

Booting the Computer

Very briefly, the Pentium boot process is as follows. Every Pentium contains a parentboard (formerly called a motherboard before political correctness hit the computer industry). On the parentboard is a program called the system **BIOS (Bask Input Output System).** The BIOS contains low-level I/O software, including procedures to read the keyboard, write to the screen, and do disk I/O, among other things. Nowadays, it is held in a flash RAM, which is nonvolatile but which can be updated by the operating system when bugs are found in the BIOS. When the computer is booted, the BIOS is started. It first checks to see how much RAM is installed and whether the keyboard and other basic devices are installed and responding correctly. It starts out by scanning the ISA and PCI buses to detect all the devices attached to them. Some of these devices are typically **legacy** (i.e., designed before plug and play was invented) and have fixed interrupt levels and I/O addresses (possibly set by switches or jumpers on the I/O card, but not modifiable by the operating system). These devices are recorded. The plug and play devices are also recorded. If the devices present are different from when the system was last booted, the new devices are configured. The BIOS then determines the boot device by trying a list of devices stored in the CMOS memory. The user can change this list by entering a BIOS configuration program just after booting. Typically, an attempt is made to boot'from the floppy disk, if one is present. If that fails the CD-ROM drive is queried to see if a bootable CD-ROM is present. If neither a floppy nor a CD-ROM is present, the system is booted from the hard disk. The first sector from the boot device is read into memory and executed. This sector contains a program that normally examines the partition table at the end of the boot sector to determine which partition is active. Then a secondary boot loader is read in from that partition. This loader reads in the operating system from the active partition and starts it. The operating system then queries the BIOS to get the configuration information. For each device, it checks to see if it has the

device driver. If not, it asks the user to insert a CD-ROM containing the driver (supplied by the device**'s** manufacturer). Once it has all the device drivers, the operating system loads them into the kernel. Then it initializes its tables, creates whatever background processes are needed, and starts up a login program or GUI.