CCS 3301 SYSTEMS PROGRAMMING

Prerequisite

Operating Systems

Purpose

To introduce students to programming concepts of operating systems modules.

Objectives

At the end of the course a student is expected to be able :

- To explain the scope and tasks of systems programming.
- To describe the processes as fundamental objects in OS.
- To describe the basic concepts of threads and multithreaded programming.
- To describe the system calls, their purpose and usage in systems interactions.

COURSE CONTENTS

1. Processes as fundamental objects in OS.
    - Process concepts from OS point of view
2. Various computing environments
    - Traditional
    - Mobile
    - Distributed
    - Client server
    - Peer to peer
    - Cloud computing
    - Visualization
    - Real time embedded systems
3. OS structures from systems programmer point of view.
    - Operating system services
    - User-OS interface
4. System calls
    - Basic concepts
    - Types of system calls
    - System programs
5. Operating system structure a review and latest developments
    - Pre visualization and virtual machine
    - Java virtual machine and .net framework
6. Process
    - Process state

- Interprocess communication(IPC)
- IPC models

7. Interprocess communication mechanisms
    - Message queues.
    - Signals and semaphores.
    - Shared memory.
    - Client/Server communication
    - Sockets
    - Remote procedure calls
    - Remote method invocation
8. Thread
    - Overview
    - Multi core programming
    - Parallelism
    - Multithreaded models.

Teaching methods

Lecture, class discussion and lab demonstration.

Evaluation

Lab exercises and assignment	15 marks
CAT sit in	15 marks
Final written examination	70 marks

Reference:

(1) Operating systems internals and design William stalling ,Pearson 2009
(2) UNIX System Programming Haviland, K. Addison-Wesley, 1999.
(3) Operating system concepts 8th or 9th edition Silberchatz, Galvin,gagneReilly
(4) Understanding Unix/Linux Programming: A guide
(5) Interprocess Communication in UNIX: The Nooks & Crannies
(6) Win32 System Programming, Addison-Wesley, 1997.

INTRODUCTION

Run-Time Environments (RTE)

One of the main objectives of this course is learning to be intelligent users of modern, sophisticated RTEs.

The general topics we will cover are:

1. What is a runtime environment (RTE)?

A **runtime environment** is a virtual machine state which provides software services for processes or programs while a computer is running. It may pertain to the operating system itself, or the software that runs beneath it. The primary purpose is to accomplish the objective of "platform independent" programming.

Runtime activities include loading and linking of the classes needed to execute a program, optional machine code generation and dynamic optimization of the program, and actual program execution.

For example, a program written in Java would receive services from the Java Runtime Environment by issuing commands from which the expected result is returned by the Java software. By providing these services, the Java software is considered the runtime environment of the program. Both the program and the Java software combined request services from the operating system. The operating system kernel provides services for itself and all processes and software running under its control. The Operating System may be considered as providing a runtime environment for itself.

In most cases, the operating system handles loading the program with a piece of code called the loader, doing basic memory setup and linking the program with any dynamically linked libraries it references. In some cases a language or implementation will have these tasks done by the language runtime instead, though this is unusual in mainstream languages on common consumer operating systems.

Some program debugging can only be performed (or are more efficient or accurate) when performed at runtime. Logical errors and array bounds checking are examples. For this reason, some programming bugs are not discovered until the program is tested in a "live" environment with real data, despite sophisticated compile-time checking and pre-release testing. In this case, the end user may encounter a *runtime error* message.

Early runtime libraries such as that of Fortran provided such features as mathematical operations. Other languages add more sophisticated memory garbage collection, often in association with support for objects.

More recent languages tend to have considerably larger runtimes with considerably more functionality. Many object oriented languages also include a system known as the "dispatcher" and "classloader". The Java Virtual Machine (JVM) is an example of such a runtime: It also interprets or compiles the portable binary Java programs (bytecode) at runtime. The .NET framework is another example of a runtime library.

Examples of RTEs:

1. Unix and Win32 Operating Systems
2. Java Virtual Machine (JVM)
3. On occasion, Servlet containers and distributed RTEs
4. COMMON LANGUAGE RUNTIME(CLR) IN WINDOWS

An Operating System provides services to both the users and to the programs.

- It provides programs an environment to execute.
- It provides users the services to execute the programs in a convenient manner.

Following are a few common services provided by an operating system −

- Program execution
- I/O operations
- File System manipulation
- Communication
- Error Detection
- Resource Allocation
- Protection

Program execution

Operating systems handle many kinds of activities from user programs to system programs like printer spooler, name servers, file server, etc. Each of these activities is encapsulated as a process.

A process includes the complete execution context (code to execute, data to manipulate, registers, OS resources in use). Following are the major activities of an operating system with respect to program management −

- Loads a program into memory.
- Executes the program.
- Handles program's execution.

- Provides a mechanism for process synchronization.
- Provides a mechanism for process communication.
- Provides a mechanism for deadlock handling.

I/O Operation

An I/O subsystem comprises of I/O devices and their corresponding driver software. Drivers hide the peculiarities of specific hardware devices from the users.

An Operating System manages the communication between user and device drivers.

- I/O operation means read or write operation with any file or any specific I/O device.
- Operating system provides the access to the required I/O device when required.

File system manipulation

A file represents a collection of related information. Computers can store files on the disk (secondary storage), for long-term storage purpose. Examples of storage media include magnetic tape, magnetic disk and optical disk drives like CD, DVD. Each of these media has its own properties like speed, capacity, data transfer rate and data access methods.

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions. Following are the major activities of an operating system with respect to file management −

- Program needs to read a file or write a file.
- The operating system gives the permission to the program for operation on file.
- Permission varies from read-only, read-write, denied and so on.
- Operating System provides an interface to the user to create/delete files.
- Operating System provides an interface to the user to create/delete directories.
- Operating System provides an interface to create the backup of file system.

Communication

In case of distributed systems which are a collection of processors that do not share memory, peripheral devices, or a clock, the operating system manages communications between all the processes. Multiple processes communicate with one another through communication lines in the network.

The OS handles routing and connection strategies, and the problems of contention and security. Following are the major activities of an operating system with respect to communication −

- Two processes often require data to be transferred between them

- Both the processes can be on one computer or on different computers, but are connected through a computer network.
- Communication may be implemented by two methods, either by Shared Memory or by Message Passing.

Error handling

Errors can occur anytime and anywhere. An error may occur in CPU, in I/O devices or in the memory hardware. Following are the major activities of an operating system with respect to error handling −

- The OS constantly checks for possible errors.
- The OS takes an appropriate action to ensure correct and consistent computing.

Resource Management

In case of multi-user or multi-tasking environment, resources such as main memory, CPU cycles and files storage are to be allocated to each user or job. Following are the major activities of an operating system with respect to resource management −

- The OS manages all kinds of resources using schedulers.
- CPU scheduling algorithms are used for better utilization of CPU.

Protection

Considering a computer system having multiple users and concurrent execution of multiple processes, the various processes must be protected from each other's activities.

Protection refers to a mechanism or a way to control the access of programs, processes, or users to the resources defined by a computer system. Following are the major activities of an operating system with respect to protection −

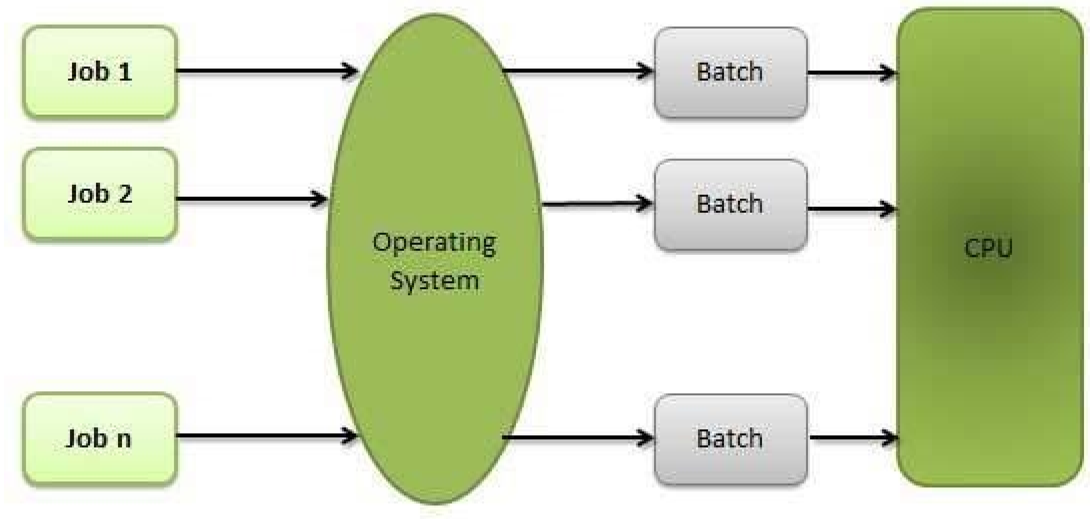
- The OS ensures that all access to system resources is controlled.
- The OS ensures that external I/O devices are protected from invalid access attempts.
- The OS provides authentication features for each user by means of passwords.

Batch processing

Batch processing is a technique in which an Operating System collects the programs and data together in a batch before processing starts. An operating system does the following activities related to batch processing −

- The OS defines a job which has predefined sequence of commands, programs and data as a single unit.

- The OS keeps a number a jobs in memory and executes them without any manual information.
- Jobs are processed in the order of submission, i.e., first come first served fashion.
- When a job completes its execution, its memory is released and the output for the job gets copied into an output spool for later printing or processing.



Advantages

- Batch processing takes much of the work of the operator to the computer.
- Increased performance as a new job get started as soon as the previous job is finished, without any manual intervention.

Disadvantages

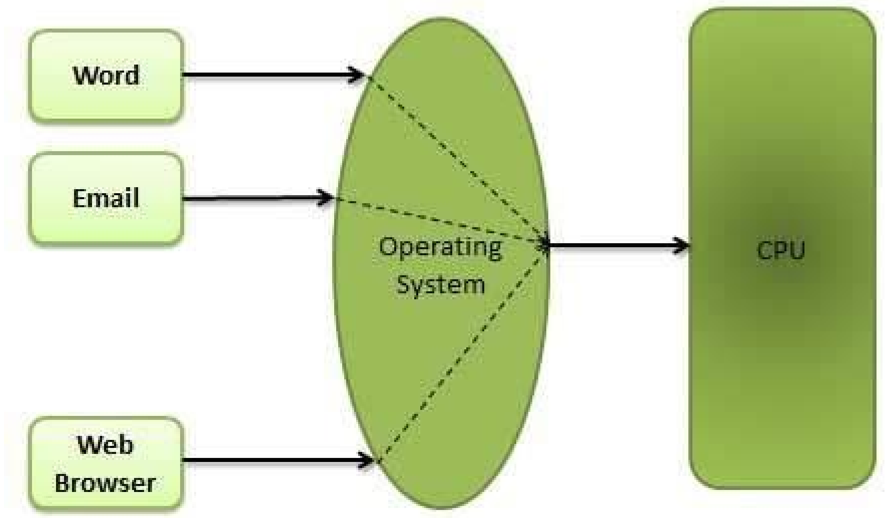
- Difficult to debug program.
- A job could enter an infinite loop.
- Due to lack of protection scheme, one batch job can affect pending jobs.

Multitasking

Multitasking is when multiple jobs are executed by the CPU simultaneously by switching between them. Switches occur so frequently that the users may interact with each program while it is running. An OS does the following activities related to multitasking −

- The user gives instructions to the operating system or to a program directly, and receives an immediate response.
- The OS handles multitasking in the way that it can handle multiple operations/executes multiple programs at a time.
- Multitasking Operating Systems are also known as Time-sharing systems.

- These Operating Systems were developed to provide interactive use of a computer system at a reasonable cost.
- A time-shared operating system uses the concept of CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared CPU.
- Each user has at least one separate program in memory.

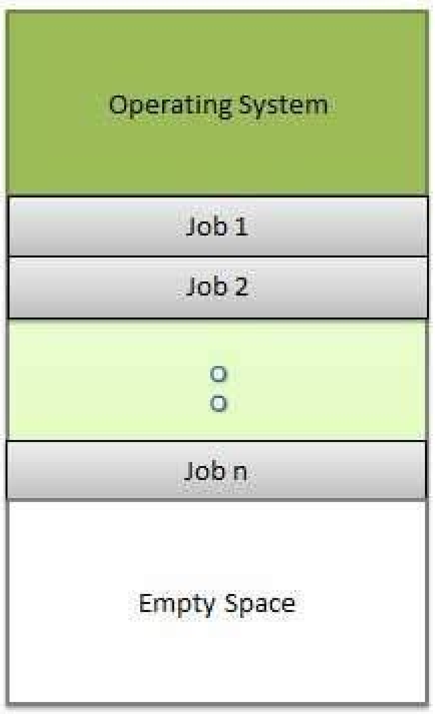


- A program that is loaded into memory and is executing is commonly referred to as a **process**.
- When a process executes, it typically executes for only a very short time before it either finishes or needs to perform I/O.
- Since interactive I/O typically runs at slower speeds, it may take a long time to complete. During this time, a CPU can be utilized by another process.
- The operating system allows the users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user.
- As the system switches CPU rapidly from one user/program to the next, each user is given the impression that he/she has his/her own CPU, whereas actually one CPU is being shared among many users.

Multiprogramming

Sharing the processor, when two or more programs reside in memory at the same time, is referred as **multiprogramming**. Multiprogramming assumes a single shared processor. Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute.

The following figure shows the memory layout for a multiprogramming system.



An OS does the following activities related to multiprogramming.

- The operating system keeps several jobs in memory at a time.
- This set of jobs is a subset of the jobs kept in the job pool.
- The operating system picks and begins to execute one of the jobs in the memory.
- Multiprogramming operating systems monitor the state of all active programs and system resources using memory management programs to ensures that the CPU is never idle, unless there are no jobs to process.

Advantages

- High and efficient CPU utilization.
- User feels that many programs are allotted CPU almost simultaneously.

Disadvantages

- CPU scheduling is required.
- To accommodate many jobs in memory, memory management is required.

Interactivity

Interactivity refers to the ability of users to interact with a computer system. An Operating system does the following activities related to interactivity −

- Provides the user an interface to interact with the system.

- Manages input devices to take inputs from the user. For example, keyboard.
- Manages output devices to show outputs to the user. For example, Monitor.

The response time of the OS needs to be short, since the user submits and waits for the result.

Real Time System

Real-time systems are usually dedicated, embedded systems. An operating system does the following activities related to real-time system activity.

- In such systems, Operating Systems typically read from and react to sensor data.
- The Operating system must guarantee response to events within fixed periods of time to ensure correct performance.

Distributed Environment

A distributed environment refers to multiple independent CPUs or processors in a computer system. An operating system does the following activities related to distributed environment −

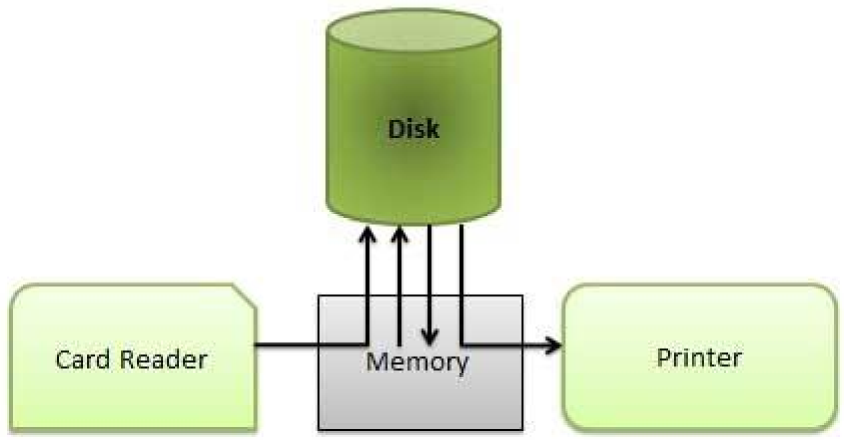
- The OS distributes computation logics among several physical processors.
- The processors do not share memory or a clock. Instead, each processor has its own local memory.
- The OS manages the communications between the processors. They communicate with each other through various communication lines.

Spooling

Spooling is an acronym for simultaneous peripheral operations on line. Spooling refers to putting data of various I/O jobs in a buffer. This buffer is a special area in memory or hard disk which is accessible to I/O devices.

An operating system does the following activities related to distributed environment −

- Handles I/O device data spooling as devices have different data access rates.
- Maintains the spooling buffer which provides a waiting station where data can rest while the slower device catches up.
- Maintains parallel computation because of spooling process as a computer can perform I/O in parallel fashion. It becomes possible to have the computer read data from a tape, write data to disk and to write out to a tape printer while it is doing its computing task.



Advantages

- The spooling operation uses a disk as a very large buffer.
- Spooling is capable of overlapping I/O operation for one job with processor operations for another job.

Process management

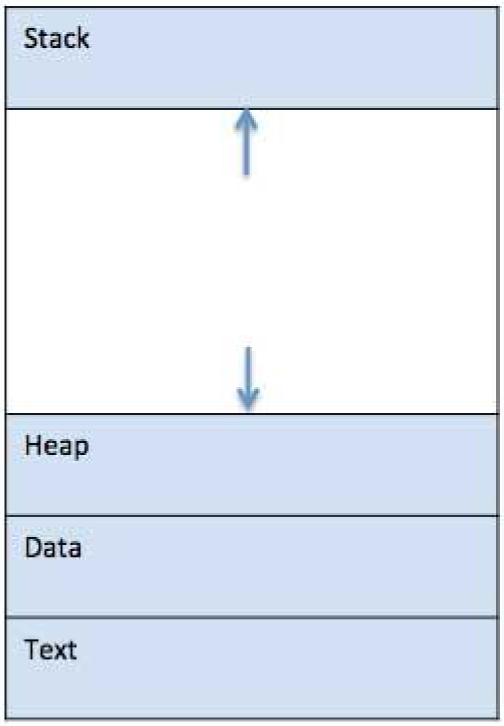
Process

A process is basically a program in execution. The execution of a process must progress in a sequential fashion.

A process is defined as an entity which represents the basic unit of work to be implemented in the system.

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory −



1.

S.N.	Component & Description
1	**Stack** The process Stack contains the temporary data such as method/function parameters, return address and local variables.
2	**Heap** This is dynamically allocated memory to a process during its run time.
3	**Text** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
4	**Data** This section contains the global and static variables.

2. Program

3. A program is a piece of code which may be a single line or millions of lines. A computer program is usually written by a computer programmer in a programming language. For example, here is a simple program written in C programming language −

```c
#include <stdio.h>

int main() {
   printf("Hello, World! \n");
   return 0;
}
```

A computer program is a collection of instructions that performs a specific task when executed by a computer. When we compare a program with a process, we can conclude that a process is a dynamic instance of a computer program.

A part of a computer program that performs a well-defined task is known as an **algorithm**. A collection of computer programs, libraries and related data are referred to as a **software**.

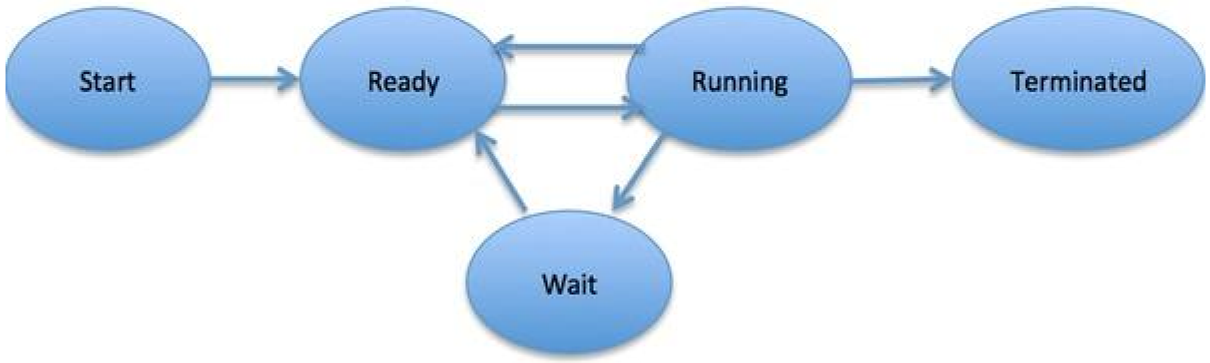
Process Life Cycle

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

In general, a process can have one of the following five states at a time.

S.N.	State & Description
1	**Start** This is the initial state when a process is first started/created.
2	**Ready** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3	**Running** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.

4	**Waiting** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
5	**Terminated or Exit** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.



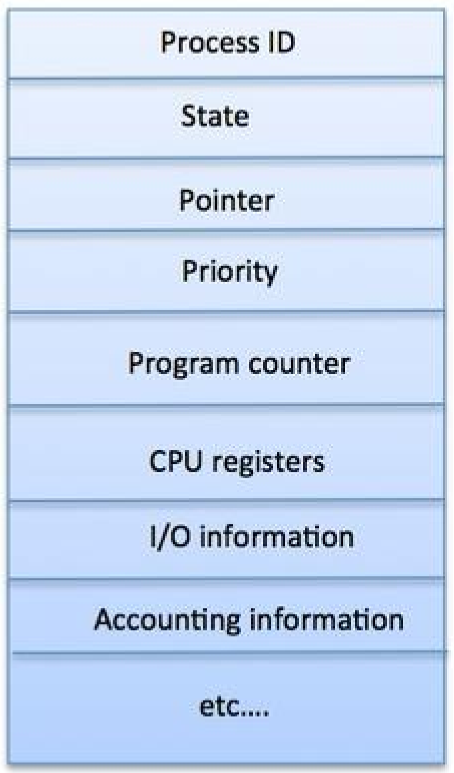
Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table −

S.N.	Information & Description
1	**Process State** The current state of the process i.e., whether it is ready, running, waiting, or whatever.
2	**Process privileges** This is required to allow/disallow access to system resources.
3	**Process ID** Unique identification for each of the process in the operating system.

4	**Pointer** A pointer to parent process.
5	**Program Counter** Program Counter is a pointer to the address of the next instruction to be executed for this process.
6	**CPU registers** Various CPU registers where process need to be stored for execution for running state.
7	**CPU Scheduling Information** Process priority and other scheduling information which is required to schedule the process.
8	**Memory management information** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
9	**Accounting information** This includes the amount of CPU used for process execution, time limits, execution ID etc.
10	**IO status information** This includes a list of I/O devices allocated to the process.

10. The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB −



The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

Multithreading

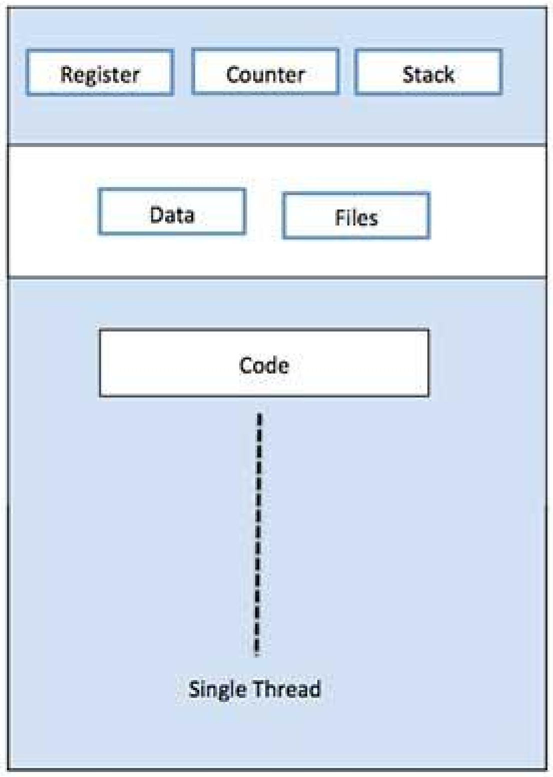
Thread

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

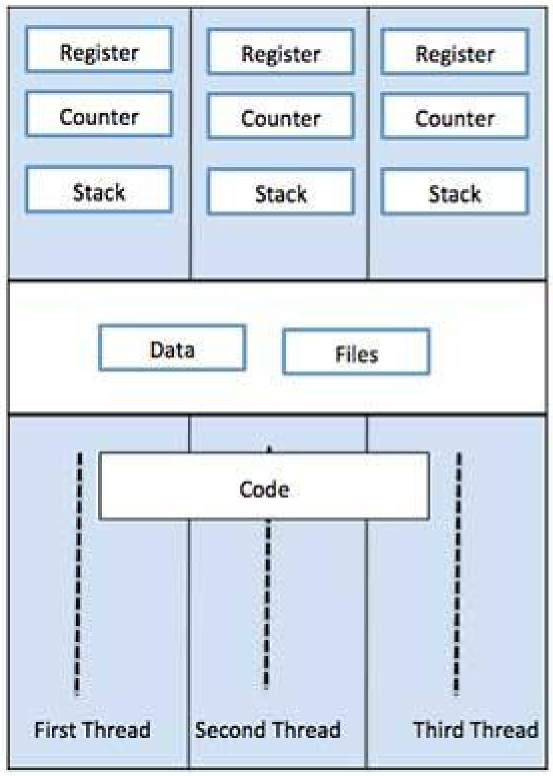
A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.



Single Process P with single thread



Single Process P with three threads

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.

4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

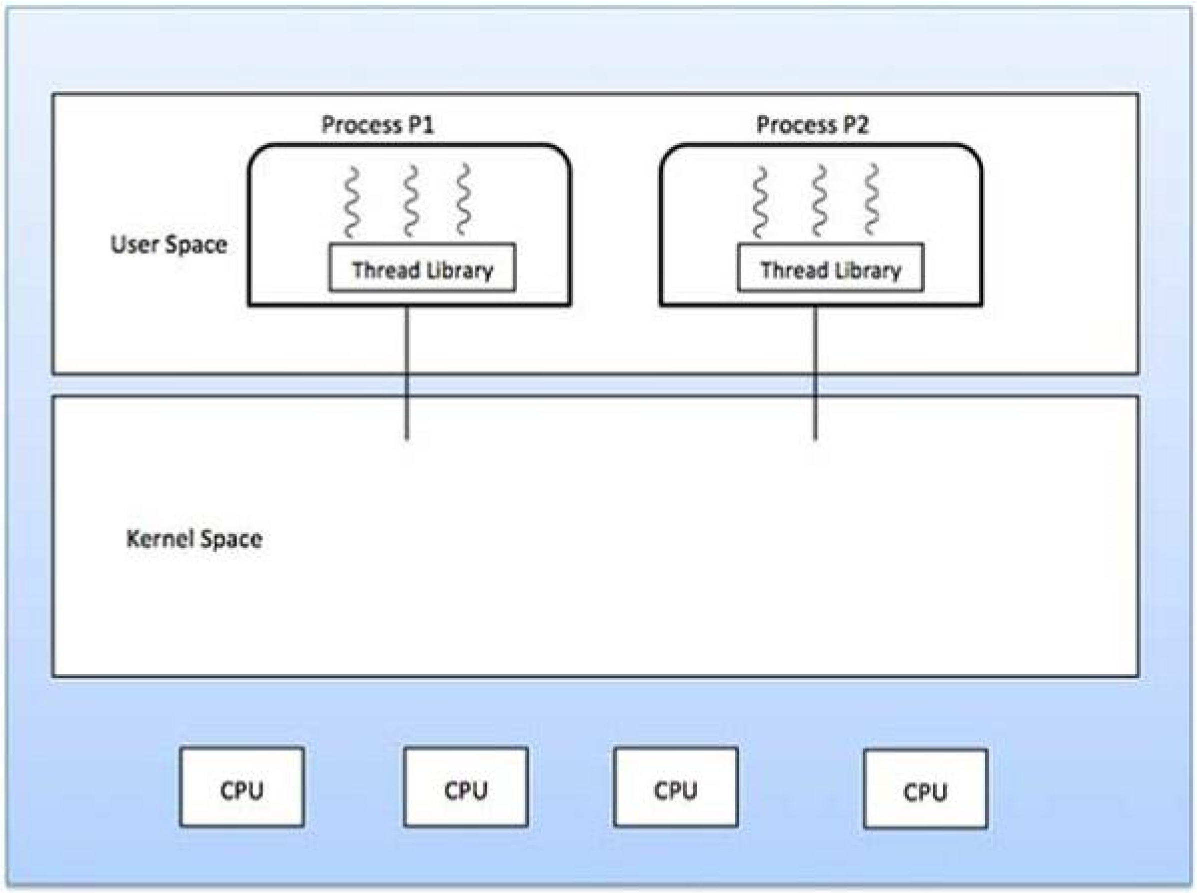
Types of Thread

Threads are implemented in following two ways −

- **User Level Threads** − User managed threads.
- **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core.

User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.



Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The

Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

Multithreading Models

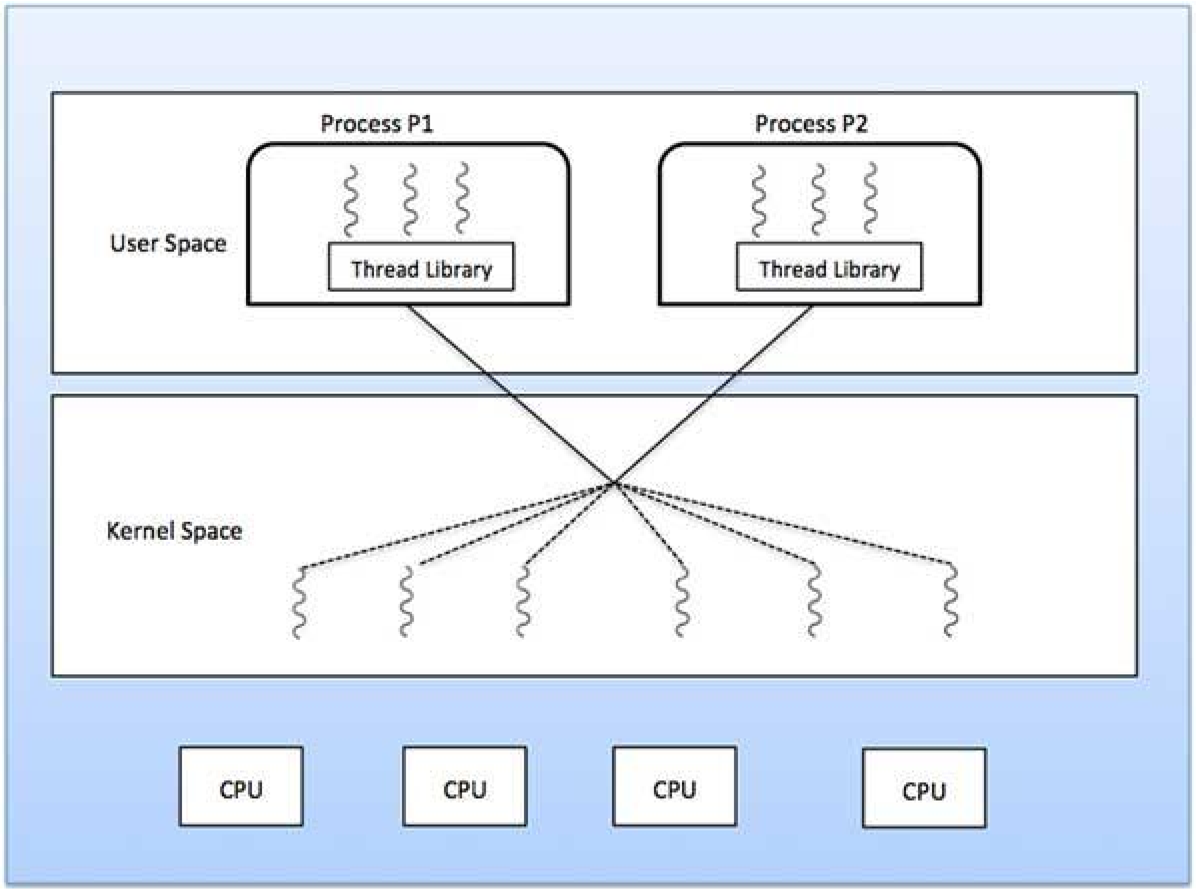
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

Many to Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

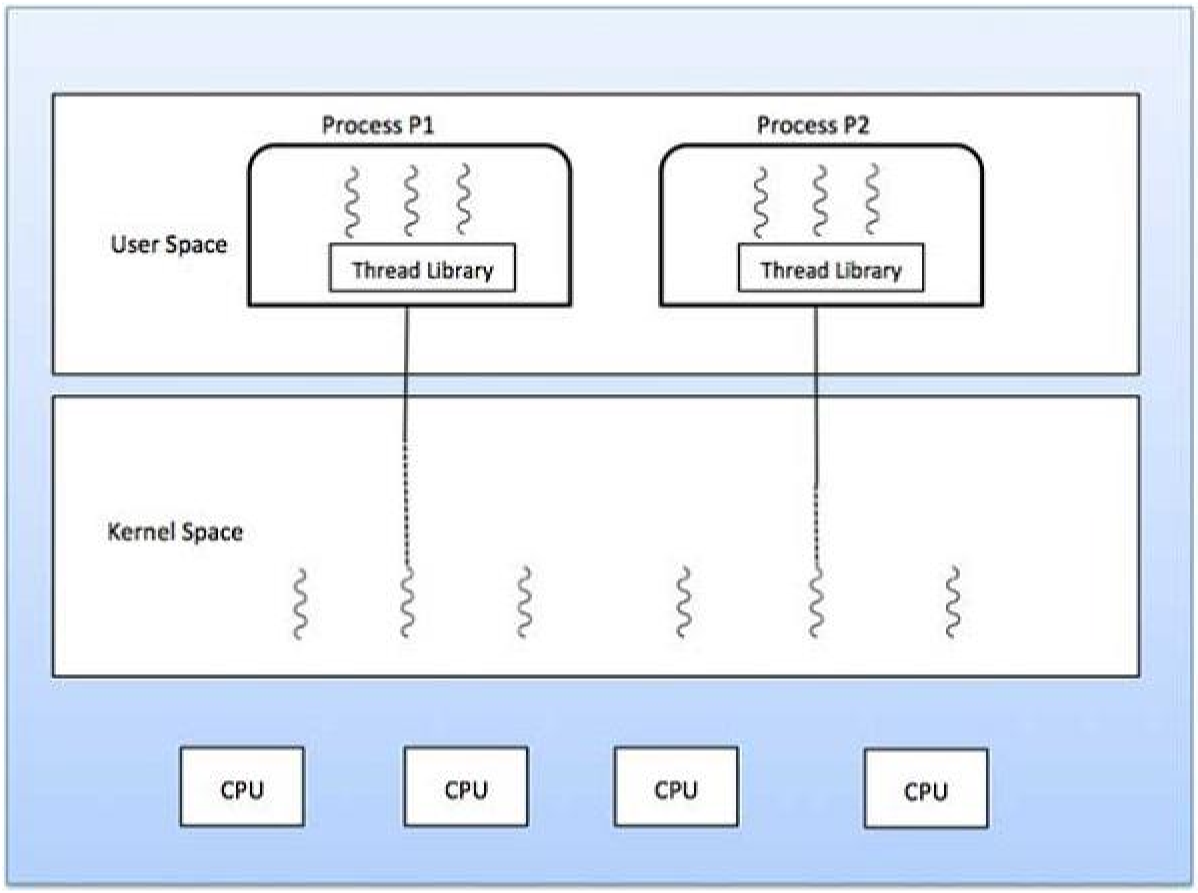
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.



Many to One Model

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

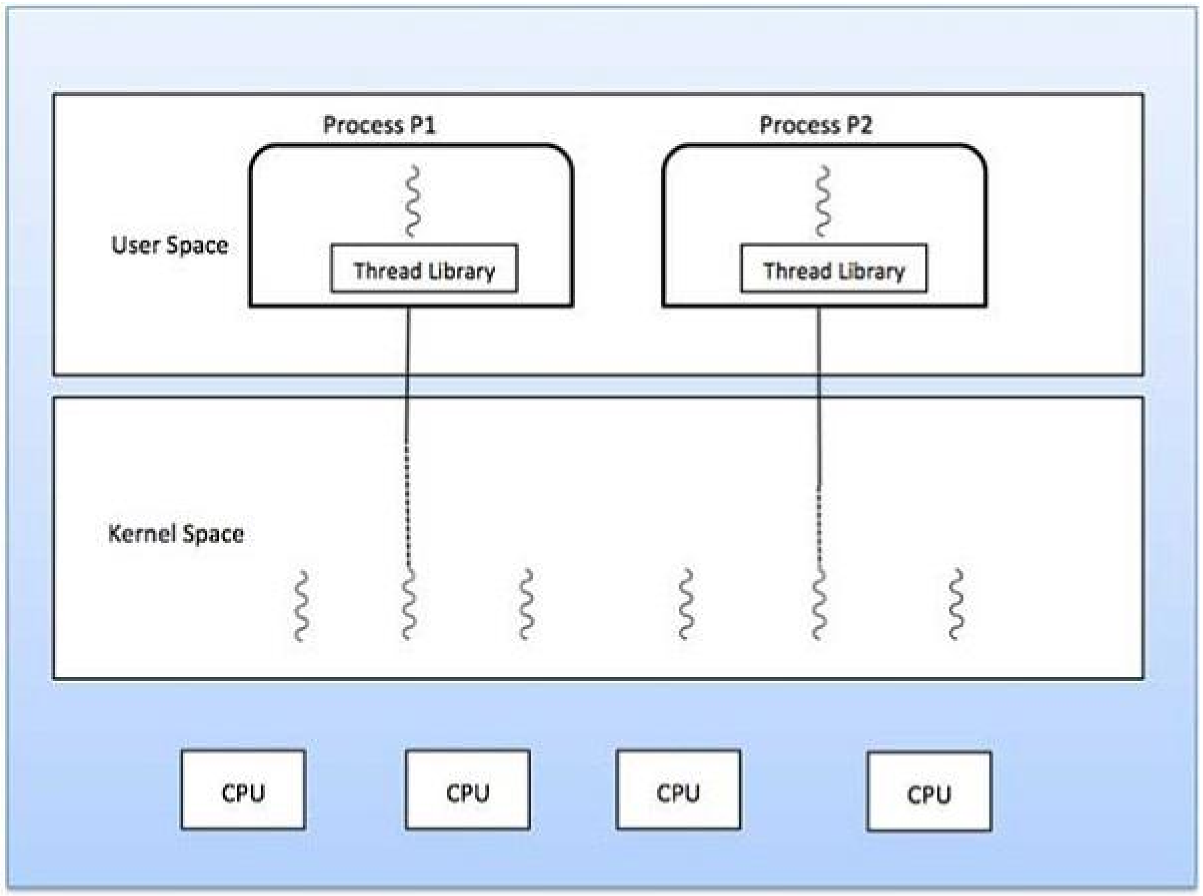
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



One to One Model

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.



Difference between User-Level & Kernel-Level Thread

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Windows Processes and Threads

Every process contains one or more threads, and the Windows thread is the basic executable unit;. Threads are scheduled on the basis of the usual factors: availability of resources such as CPUs and physical memory, priority, fairness, and so on. Windows has long supported multiprocessor systems, so threads can be allocated to separate processors within a computer.

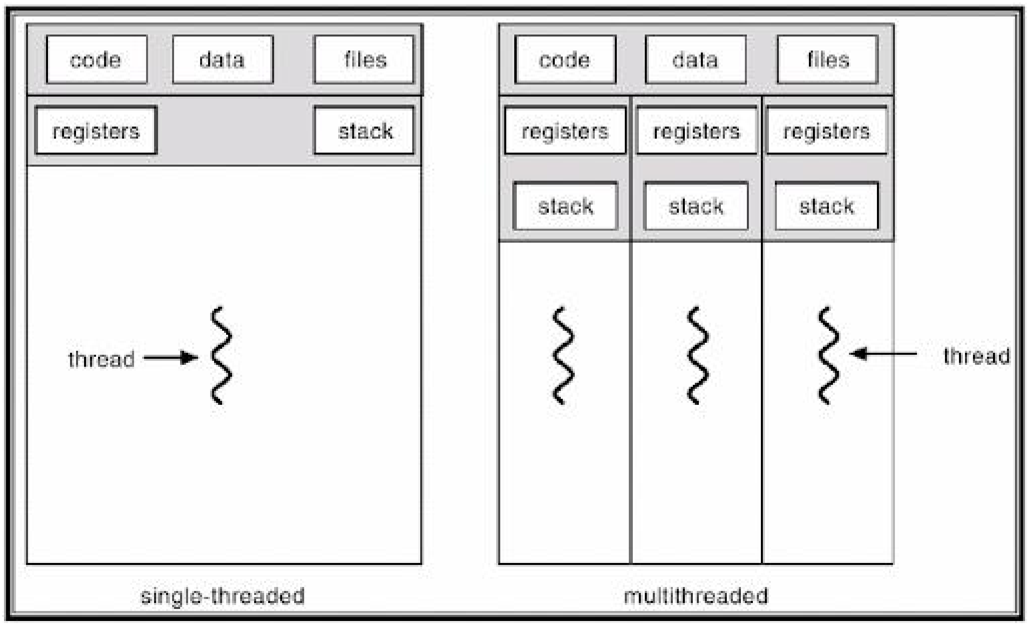
From the programmer's perspective, each Windows process includes resources such as the following components:

- One or more threads.
- A virtual address space that is distinct from other processes' address spaces. Note that shared memory-mapped files share physical memory, but the sharing processes will probably use different virtual addresses to access the mapped file.
- One or more code segments, including code in DLLs.
- One or more data segments containing global variables.
- Environment strings with environment variable information, such as the current search path.
- The process heap.
- Resources such as open handles and other heaps.

Each thread in a process shares code, global variables, environment strings, and resources. Each thread is independently scheduled, and a thread has the following elements:

- A stack for procedure calls, interrupts, exception handlers, and automatic storage.
- Thread Local Storage (TLS)—An arraylike collection of pointers giving each thread the ability to allocate storage to create its own unique data environment.
- An argument on the stack, from the creating thread, which is usually unique for each thread.
- A context structure, maintained by the kernel, with machine register values.

Figure below shows a process with several threads. This figure is schematic and does not indicate actual memory addresses, nor is it drawn to scale.



A Process and Its Threads

This chapter shows how to work with processes consisting of a single thread. Chapter 7 shows how to use multiple threads.

INTER PROCESS COMMUNICATION

Inter process communication (IPC) is used for exchanging data between multiple threads in one or more processes or programs. The Processes may be running on single or multiple computers connected by a network. The full form of IPC is Inter-process communication.

It is a set of programming interface which allow a programmer to coordinate activities among various program processes which can run concurrently in an operating system. This allows a specific program to handle many user requests at the same time.

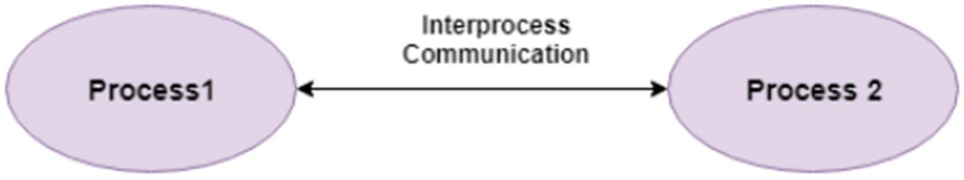
Since every single user request may result in multiple processes running in the operating system, the process may require to communicate with each other. Each IPC protocol approach has its own advantage and limitation, so it is not unusual for a single program to use all of the IPC methods

There are numerous reasons for providing an environment or situation which allows process co-operation:

- **Information sharing:** Since some users may be interested in the same piece of information (for example, a shared file), you must provide a situation for allowing concurrent access to that information.
- **Computation speedup:** If you want a particular work to run fast, you must break it into sub-tasks where each of them will get executed in parallel with the other tasks. Note that such a speed-up can be attained only when the computer has compound or various processing elements like CPUs or I/O channels.
- **Modularity:** You may want to build the system in a modular way by dividing the system functions into split processes or threads.
- **Convenience:** Even a single user may work on many tasks at a time. For example, a user may be editing, formatting, printing, and compiling in parallel.

Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or transferring of data from one process to another.

A diagram that illustrates interprocess communication is as follows −



The models of interprocess communication are as follows −

<u>Shared Memory Model</u>

Shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. All POSIX systems, as well as Windows operating systems use shared memory.

Advantage of Shared Memory Model

Memory communication is faster on the shared memory model as compared to the message passing model on the same machine.

Disadvantages of Shared Memory Model

Some of the disadvantages of shared memory model are as follows −

- All the processes that use the shared memory model need to make sure that they are not writing to the same memory location.
- Shared memory model may create problems such as synchronization and memory protection that need to be addressed.

Message Passing Model

Multiple processes can read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

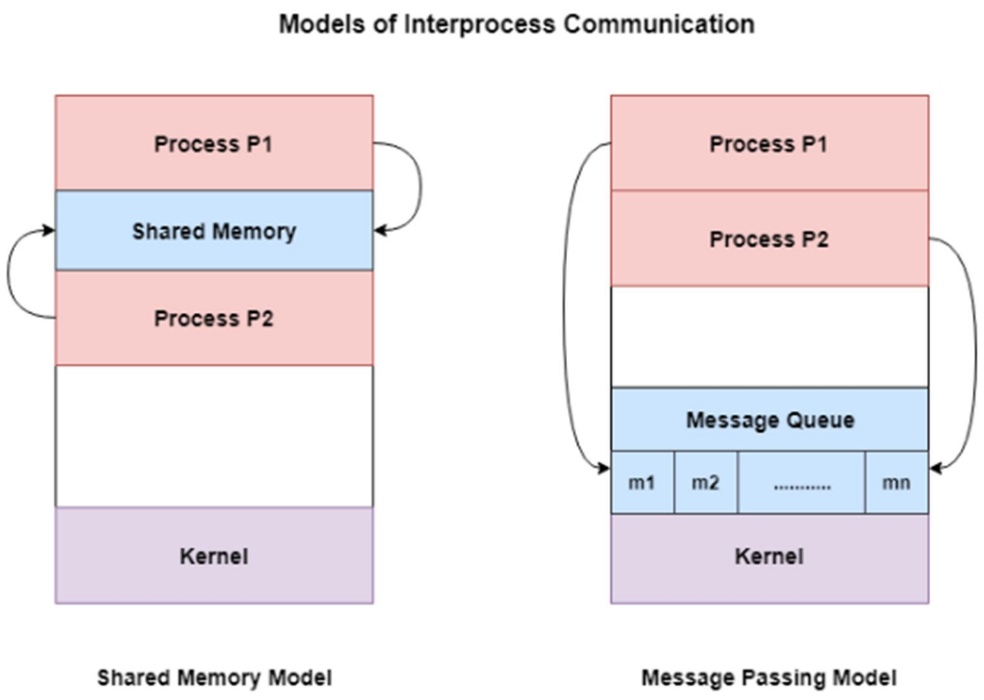
Advantage of Messaging Passing Model

The message passing model is much easier to implement than the shared memory model.

Disadvantage of Messaging Passing Model

The message passing model has slower communication than the shared memory model because the connection setup takes time.

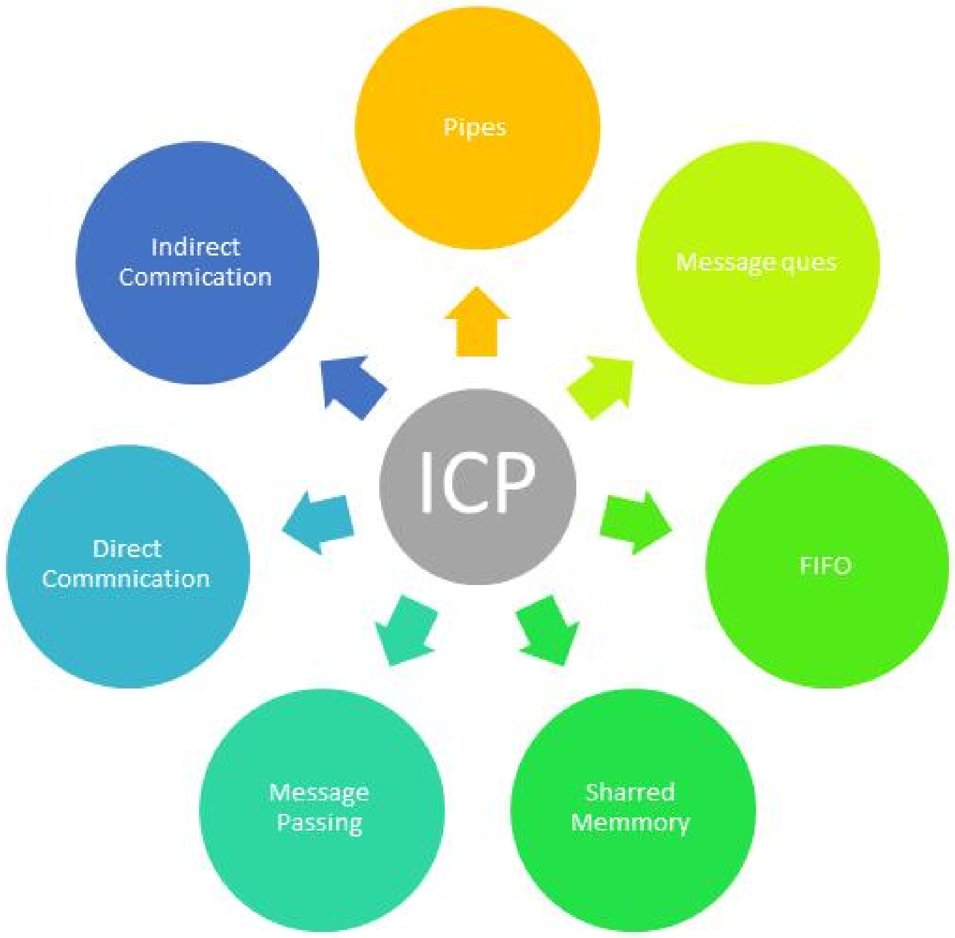
A diagram that demonstrates the shared memory model and message passing model is given as follows −



The form of the data and the location gets established by these processes and are not under the control of the operating system. The processes are also in charge to ensure that they are not writing to the same old location simultaneously.

Methods in Inter process Communication

Inter-process communication (IPC) is set of interfaces, which is usually programmed in order for the programs to communicate between series of processes. This allows running programs concurrently in an Operating System. These are the methods in IPC:



1. **Pipes (Same Process) –**
   This allows flow of data in one direction only. Analogous to simplex systems (Keyboard). Data from the output is usually buffered until input process receives it which must have a common origin.
2. **Names Pipes (Different Processes) –**
   This is a pipe with a specific name it can be used in processes that don't have

a shared common process origin. E.g. is FIFO where the details written to a pipe is first named.

3. **Message Queuing –**
   This allows messages to be passed between processes using either a single queue or several message queue. This is managed by system kernel these messages are coordinated using an API.
4. **Semaphores –**
   This is used in solving problems associated with synchronization and to avoid race condition. These are integer values which are greater than or equal to 0.
5. **Shared memory –**
   This allows the interchange of data through a defined area of memory. Semaphore values have to be obtained before data can get access to shared memory.
6. **Sockets –**
   This method is mostly used to communicate over a network between a client and a server. It allows for a standard connection which is computer and OS independent.

SOCKET IPC MECHANISMS

Sockets allow communication between two different processes on the same or different machines. To be more precise, it's a way to talk to other computers using standard Unix file descriptors. In Unix, every I/O action is done by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file and it can be a network connection, a text file, a terminal, or something else.

To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files and pipes.

Sockets were first introduced in 2.1BSD and subsequently refined into their current form with 4.2BSD. The sockets feature is now available with most current UNIX system releases.

A Socket is used in a client-server application framework. A server is a process that performs some functions on request from a client. Most of the application-level protocols like FTP, SMTP, and POP3 make use of sockets to establish connection between client and server and then for exchanging data.

Socket Types

There are four types of sockets available to the users. The first two are most commonly used and the last two are rarely used.

Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.

- **Stream Sockets** − Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order − "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.
- **Datagram Sockets** − Delivery in a networked environment is not guaranteed. They're connectionless because you don't need to have an open connection as in Stream Sockets − you build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).
- **Raw Sockets** − These provide users access to the underlying communication protocols, which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.
- **Sequenced Packet Sockets** − They are similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as a part of the Network Systems (NS) socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the Sequence Packet Protocol (SPP) or Internet Datagram Protocol (IDP) headers on a packet or a group of packets, either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets.

The next few chapters are meant to strengthen your basics and prepare a foundation before you can write Server and Client programs using *socket*. If you directly want to jump to see how to write a client and server program, then you can do so but it is not recommended. It is strongly recommended that you go step by step and complete these initial few chapters to make your base before moving on to do programming.

Socket - Network Addresses

The IP Address.

The IP host address, or more commonly just IP address, is used to identify hosts connected to the Internet. IP stands for Internet Protocol and refers to the Internet Layer of the overall network architecture of the Internet.

An IP address is a 32-bit quantity interpreted as four 8-bit numbers or octets. Each IP address uniquely identifies the participating user network, the host on the network, and the class of the user network.

An IP address is usually written in a dotted-decimal notation of the form N1.N2.N3.N4, where each Ni is a decimal number between 0 and 255 decimal (00 through FF hexadecimal).

Address Classes

IP addresses are managed and created by the *Internet Assigned Numbers Authority* (IANA). There are five different address classes. You can determine which class an IP address is in by examining the first four bits of the IP address.

- **Class A** addresses begin with **0xxx**, or **1 to 126** decimal.
- **Class B** addresses begin with **10xx**, or **128 to 191** decimal.
- **Class C** addresses begin with **110x**, or **192 to 223** decimal.
- **Class D** addresses begin with **1110**, or **224 to 239** decimal.
- **Class E** addresses begin with **1111**, or **240 to 254** decimal.

Addresses beginning with **01111111**, or **127** decimal, are reserved for loopback and for internal testing on a local machine [You can test this: you should always be able to ping **127.0.0.1**, which points to yourself]; Class D addresses are reserved for multicasting; Class E addresses are reserved for future use. They should not be used for host addresses.

Example

Class	**Leftmost bits**	**Start address**	**Finish address**
A	0xxx	0.0.0.0	127.255.255.255
B	10xx	128.0.0.0	191.255.255.255
C	110x	192.0.0.0	223.255.255.255
D	1110	224.0.0.0	239.255.255.255
E	1111	240.0.0.0	255.255.255.255

Subnetting

Subnetting or subnetworking basically means to branch off a network. It can be done for a variety of reasons like network in an organization, use of different physical media (such as Ethernet, FDDI, WAN, etc.), preservation of address space, and security. The most common reason is to control network traffic.

The basic idea in subnetting is to partition the host identifier portion of the IP address into two parts −

- A subnet address within the network address itself; and
- A host address on the subnet.

For example, a common Class B address format is N1.N2.S.H, where N1.N2 identifies the Class B network, the 8-bit S field identifies the subnet, and the 8-bit H field identifies the host on the subnet.

Socket - Network Host Names

Host names in terms of numbers are difficult to remember and hence they are termed by ordinary names such as Takshila or Nalanda. We write software applications to find out the dotted IP address corresponding to a given name.

The process of finding out dotted IP address based on the given alphanumeric host name is known as **hostname resolution**.

A hostname resolution is done by special software residing on high-capacity systems. These systems are called Domain Name Systems (DNS), which keep the mapping of IP addresses and the corresponding ordinary names.

The /etc/hosts File

The correspondence between host names and IP addresses is maintained in a file called *hosts*. On most of the systems, this file is found in ***/etc*** directory.

Entries in this file look like the following −

```
# This represents a comments in /etc/hosts file.
127.0.0.1       localhost
192.217.44.207  nalanda metro
153.110.31.18   netserve
153.110.31.19   mainserver centeral
153.110.31.20   samsonite
64.202.167.10   ns3.secureserver.net
64.202.167.97   ns4.secureserver.net
66.249.89.104   www.google.com
68.178.157.132  services.amrood.com
```

Note that more than one name may be associated with a given IP address. This file is used while converting from IP address to host name and vice versa.

You would not have access to edit this file, so if you want to put any host name along with IP address, then you would need to have root permission.

Socket - Client Server Model

Most of the Net Applications use the Client-Server architecture, which refers to two processes or two applications that communicate with each other to exchange some information. One of the two processes acts as a client process, and another process acts as a server.

Client Process

This is the process, which typically makes a request for information. After getting the response, this process may terminate or may do some other processing.

Example, Internet Browser works as a client application, which sends a request to the Web Server to get one HTML webpage.

Server Process

This is the process which takes a request from the clients. After getting a request from the client, this process will perform the required processing, gather the requested information, and send it to the requestor client. Once done, it becomes ready to serve another client. Server processes are always alert and ready to serve incoming requests.

Example − Web Server keeps waiting for requests from Internet Browsers and as soon as it gets any request from a browser, it picks up a requested HTML page and sends it back to that Browser.

Note that the client needs to know the address of the server, but the server does not need to know the address or even the existence of the client prior to the connection being established. Once a connection is established, both sides can send and receive information.

2-tier and 3-tier architectures

There are two types of client-server architectures −

- **2-tier architecture** − In this architecture, the client directly interacts with the server. This type of architecture may have some security holes and performance problems. Internet Explorer and Web Server work on two-tier architecture. Here security problems are resolved using Secure Socket Layer (SSL).
- **3-tier architectures** − In this architecture, one more software sits in between the client and the server. This middle software is called 'middleware'. Middleware are used to perform all the security checks and load balancing in case of heavy load. A middleware takes all requests from the client and after performing the required authentication, it passes that request to the server. Then the server does the required processing and sends the response back to the middleware and finally the middleware passes this response back to the client. If you want to implement

a 3-tier architecture, then you can keep any middleware like Web Logic or WebSphere software in between your Web Server and Web Browser.

Types of Server

There are two types of servers you can have −

- **Iterative Server** − This is the simplest form of server where a server process serves one client and after completing the first request, it takes request from another client. Meanwhile, another client keeps waiting.
- **Concurrent Servers** − This type of server runs multiple concurrent processes to serve many requests at a time because one process may take longer and another client cannot wait for so long. The simplest way to write a concurrent server under Unix is to *fork* a child process to handle each client separately.

How to Make Client

The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. Both the processes establish their own sockets.

The steps involved in establishing a socket on the client side are as follows −

- Create a socket with the **socket()** system call.
- Connect the socket to the address of the server using the **connect()** system call.
- Send and receive data. There are a number of ways to do this, but the simplest way is to use the **read()** and **write()** system calls.

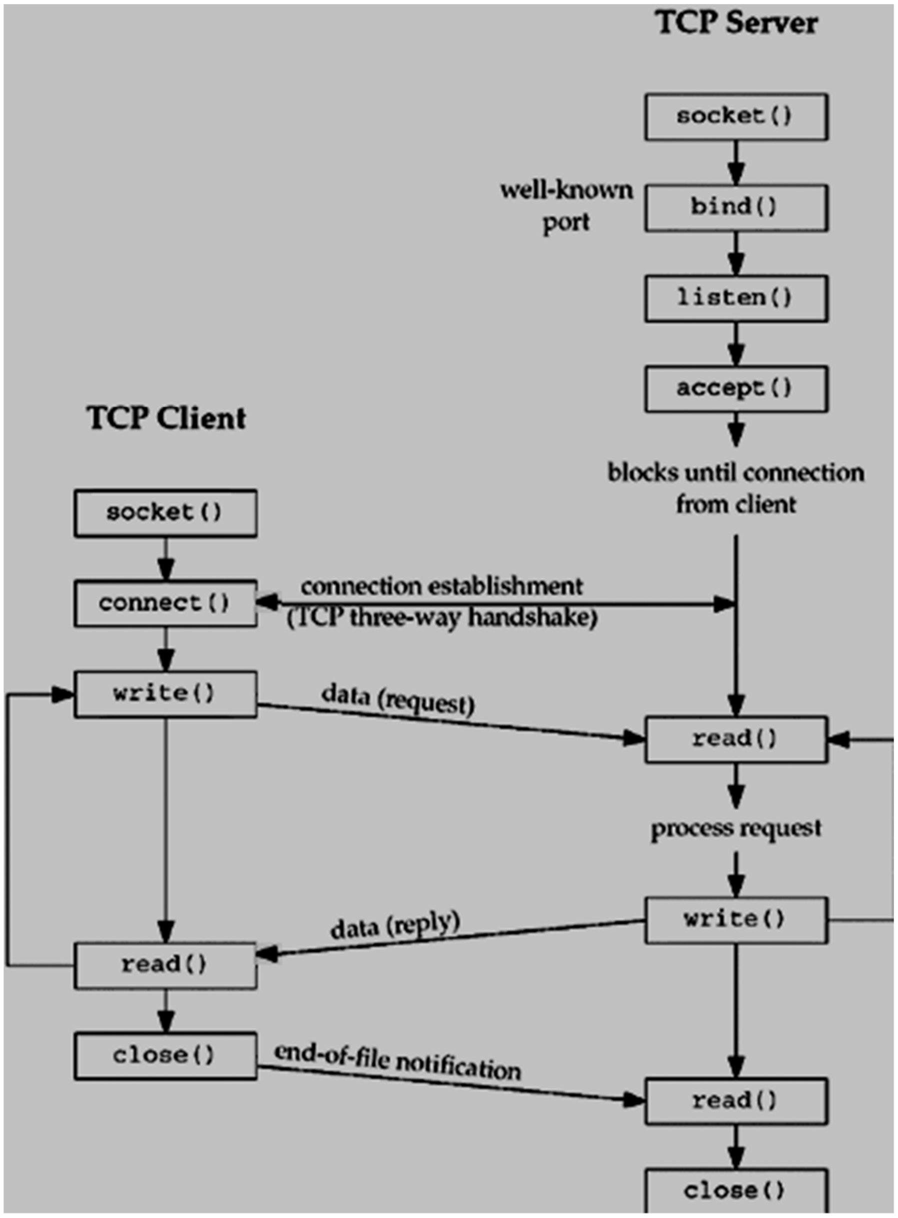
How to make a Server

The steps involved in establishing a socket on the server side are as follows −

- Create a socket with the **socket()** system call.
- Bind the socket to an address using the **bind()** system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the **listen()** system call.
- Accept a connection with the **accept()** system call. This call typically blocks the connection until a client connects with the server.
- Send and receive data using the **read()** and **write()** system calls.

Client and Server Interaction

Following is the diagram showing the complete Client and Server interaction −



Unix Socket - Structures

Various structures are used in Unix Socket Programming to hold information about the address and port, and other information. Most socket functions require a pointer to a socket address structure as an argument. Structures defined in this chapter are related to Internet Protocol Family.

sockaddr

The first structure is *sockaddr* that holds the socket information −

```
struct sockaddr {
  unsigned short   sa_family;
  char           sa_data[14];
};
```

This is a generic socket address structure, which will be passed in most of the socket function calls. The following table provides a description of the member fields −

Attribute	Values	Description
sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	It represents an address family. In most of the Internet-based applications, we use AF_INET.
sa_data	Protocol-specific Address	The content of the 14 bytes of protocol specific address are interpreted according to the type of address. For the Internet family, we will use port number IP address, which is represented by *sockaddr_in* structure defined below.

sockaddr in

The second structure that helps you to reference to the socket's elements is as follows −

```
struct sockaddr_in {
  short int            sin_family;
  unsigned short int   sin_port;
  struct in_addr       sin_addr;
  unsigned char        sin_zero[8];
};
```

Here is the description of the member fields −

Attribute	Values	Description
sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	It represents an address family. In most of the Internet-based applications, we use AF_INET.

sin_port	Service Port	A 16-bit port number in Network Byte Order.
sin_addr	IP Address	A 32-bit IP address in Network Byte Order.
sin_zero	Not Used	You just set this value to NULL as this is not being used.

in addr

This structure is used only in the above structure as a structure field and holds 32 bit netid/hostid.

```
struct in_addr {
  unsigned long s_addr;
};
```

Here is the description of the member fields −

Attribute	Values	Description
s_addr	service port	A 32-bit IP address in Network Byte Order.

hostent

This structure is used to keep information related to host.

```
struct hostent {
  char *h_name;
  char **h_aliases;
  int h_addrtype;
  int h_length;
  char **h_addr_list

#define h_addr  h_addr_list[0]
};
```

Here is the description of the member fields −

Attribute	Values	Description

h_name	ti.com etc.	It is the official name of the host. For example, tutorialspoint.com, google.com, etc.
h_aliases	TI	It holds a list of host name aliases.
h_addrtype	AF_INET	It contains the address family and in case of Internet based application, it will always be AF_INET.
h_length	4	It holds the length of the IP address, which is 4 for Internet Address.
h_addr_list	in_addr	For Internet addresses, the array of pointers h_addr_list[0], h_addr_list[1], and so on, are points to structure in_addr.

NOTE − h_addr is defined as h_addr_list[0] to keep backward compatibility.

servent

This particular structure is used to keep information related to service and associated ports.

```
struct servent {
   char  *s_name;
   char  **s_aliases;
   int   s_port;
   char  *s_proto;
};
```

Here is the description of the member fields −

Attribute	Values	Description
s_name	http	This is the official name of the service. For example, SMTP, FTP POP3, etc.
s_aliases	ALIAS	It holds the list of service aliases. Most of the time this will be set to NULL.
s_port	80	It will have associated port number. For example, for HTTP, this will be 80.

s_proto	TCP UDP	It is set to the protocol used. Internet services are provided using either TCP or UDP.

Socket Structures

Socket address structures are an integral part of every network program. We allocate them, fill them in, and pass pointers to them to various socket functions. Sometimes we pass a pointer to one of these structures to a socket function and it fills in the contents.

We always pass these structures by reference (i.e., we pass a pointer to the structure, not the structure itself), and we always pass the size of the structure as another argument.

When a socket function fills in a structure, the length is also passed by reference, so that its value can be updated by the function. We call these value-result arguments.

Always, set the structure variables to NULL (i.e., '\0') by using memset() for bzero() functions, otherwise it may get unexpected junk values in your structure.

Socket - Ports and Services

When a client process wants to a connect a server, the client must have a way of identifying the server that it wants to connect. If the client knows the 32-bit Internet address of the host on which the server resides, it can contact that host. But how does the client identify the particular server process running on that host?

To resolve the problem of identifying a particular server process running on a host, both TCP and UDP have defined a group of well-known ports.

For our purpose, a port will be defined as an integer number between 1024 and 65535. This is because all port numbers smaller than 1024 are considered *well-known* -- for example, telnet uses port 23, http uses 80, ftp uses 21, and so on.

The port assignments to network services can be found in the file /etc/services. If you are writing your own server then care must be taken to assign a port to your server. You should make sure that this port should not be assigned to any other server.

Normally it is a practice to assign any port number more than 5000. But there are many organizations who have written servers having port numbers more than 5000. For example, Yahoo Messenger runs on 5050, SIP Server runs on 5060, etc.

Example Ports and Services

Here is a small list of services and associated ports. You can find the most updated list of internet ports and associated service at IANA - TCP/IP Port Assignments.

Service	Port Number	Service Description
echo	7	UDP/TCP sends back what it receives.
discard	9	UDP/TCP throws away input.
daytime	13	UDP/TCP returns ASCII time.
chargen	19	UDP/TCP returns characters.
ftp	21	TCP file transfer.
telnet	23	TCP remote login.
smtp	25	TCP email.
daytime	37	UDP/TCP returns binary time.
tftp	69	UDP trivial file transfer.
finger	79	TCP info on users.
http	80	TCP World Wide Web.
login	513	TCP remote login.
who	513	UDP different info on users.
Xserver	6000	TCP X windows (N.B. >1023).

Port and Service Functions

Unix provides the following functions to fetch service name from the /etc/services file.

- **struct servent *getservbyname(char *name, char *proto)** − This call takes service name and protocol name, and returns the corresponding port number for that service.
- **struct servent *getservbyport(int port, char *proto)** − This call takes port number and protocol name, and returns the corresponding service name.

The return value for each function is a pointer to a structure with the following form −

```
struct servent {
   char  *s_name;
   char  **s_aliases;
   int   s_port;
   char  *s_proto;
};
```

Here is the description of the member fields −

Attribute	Values	Description
s_name	http	It is the official name of the service. For example, SMTP, FTP POP3, etc.
s_aliases	ALIAS	It holds the list of service aliases. Most of the time, it will be set to NULL.
s_port	80	It will have the associated port number. For example, for HTTP, it will be 80.
s_proto	TCP UDP	It is set to the protocol used. Internet services are provided using either TCP or UDP.

Socket - Network Byte Orders

Unfortunately, not all computers store the bytes that comprise a multibyte value in the same order. Consider a 16-bit internet that is made up of 2 bytes. There are two ways to store this value.

- **Little Endian** − In this scheme, low-order byte is stored on the starting address (A) and high-order byte is stored on the next address (A + 1).
- **Big Endian** − In this scheme, high-order byte is stored on the starting address (A) and low-order byte is stored on the next address (A + 1).

To allow machines with different byte order conventions communicate with each other, the Internet protocols specify a canonical byte order convention for data transmitted over the network. This is known as Network Byte Order.

While establishing an Internet socket connection, you must make sure that the data in the sin_port and sin_addr members of the sockaddr_in structure are represented in Network Byte Order.

Byte Ordering Functions

Routines for converting data between a host's internal representation and Network Byte Order are as follows −

Function	Description
htons()	Host to Network Short
htonl()	Host to Network Long
ntohl()	Network to Host Long
ntohs()	Network to Host Short

Listed below are some more detail about these functions −

- **unsigned short htons(unsigned short hostshort)** − This function converts 16-bit (2-byte) quantities from host byte order to network byte order.
- **unsigned long htonl(unsigned long hostlong)** − This function converts 32-bit (4-byte) quantities from host byte order to network byte order.
- **unsigned short ntohs(unsigned short netshort)** − This function converts 16-bit (2-byte) quantities from network byte order to host byte order.
- **unsigned long ntohl(unsigned long netlong)** − This function converts 32-bit quantities from network byte order to host byte order.

These functions are macros and result in the insertion of conversion source code into the calling program. On little-endian machines, the code will change the values around to network byte order. On big-endian machines, no code is inserted since none is needed; the functions are defined as null.

Program to Determine Host Byte Order

Keep the following code in a file *byteorder.c* and then compile it and run it over your machine.

In this example, we store the two-byte value 0x0102 in the short integer and then look at the two consecutive bytes, c[0] (the address A) and c[1] (the address A + 1) to determine the byte order.

Socket - IP Address Functions

Unix provides various function calls to help you manipulate IP addresses. These functions convert Internet addresses between ASCII strings (what humans prefer to use) and network byte ordered binary values (values that are stored in socket address structures).

The following three function calls are used for IPv4 addressing −

- int inet_aton(const char *strptr, struct in_addr *addrptr)
- in_addr_t inet_addr(const char *strptr)
- char *inet_ntoa(struct in_addr inaddr)

int inet_aton(const char *strptr, struct in_addr *addrptr)

This function call converts the specified string in the Internet standard dot notation to a network address, and stores the address in the structure provided. The converted address will be in Network Byte Order (bytes ordered from left to right). It returns 1 if the string was valid and 0 on error.

Following is the usage example −

in_addr_t inet_addr(const char *strptr)

This function call converts the specified string in the Internet standard dot notation to an integer value suitable for use as an Internet address. The converted address will be in Network Byte Order (bytes ordered from left to right). It returns a 32-bit binary network byte ordered IPv4 address and INADDR_NONE on error.

Following is the usage example −

char *inet_ntoa(struct in_addr inaddr)

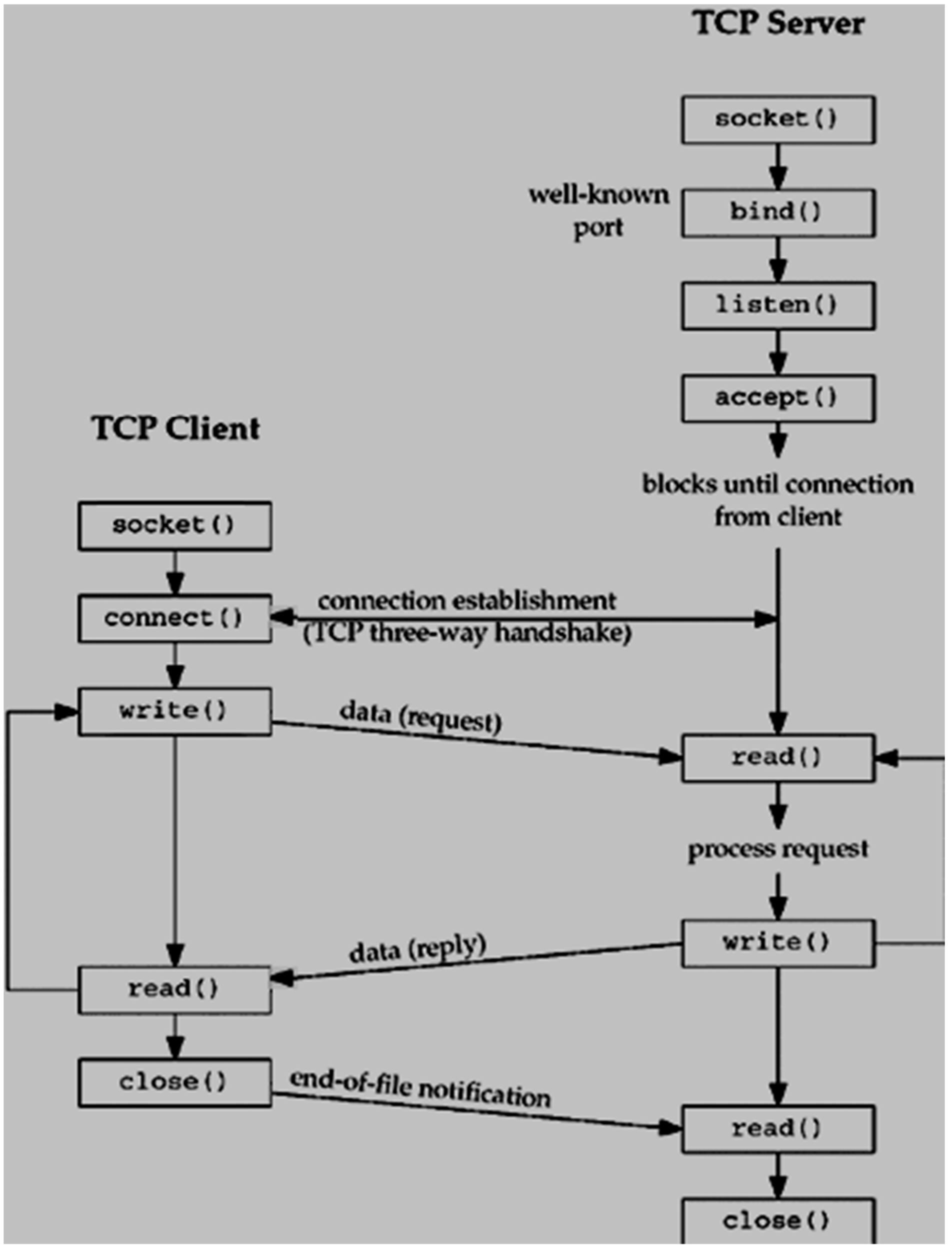
This function call converts the specified Internet host address to a string in the Internet standard dot notation.

Following is the usage example −

Socket - Core Functions

This chapter describes the core socket functions required to write a complete TCP client and server.

The following diagram shows the complete Client and Server interaction −



The socket Function

To perform network I/O, the first thing a process must do is, call the socket function, specifying the type of communication protocol desired and protocol family, etc.

```
#include <sys/types.h>
#include <sys/socket.h>

int socket (int family, int type, int protocol);
```

This call returns a socket descriptor that you can use in later system calls or -1 on error.

Parameters

family − It specifies the protocol family and is one of the constants shown below −

Family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing Sockets
AF_KEY	Ket socket

This chapter does not cover other protocols except IPv4.

type − It specifies the kind of socket you want. It can take one of the following values −

Type	Description
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_SEQPACKET	Sequenced packet socket
SOCK_RAW	Raw socket

protocol − The argument should be set to the specific protocol type given below, or 0 to select the system's default for the given combination of family and type −

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

The *connect* Function

The *connect* function is used by a TCP client to establish a connection with a TCP server.

```
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

This call returns 0 if it successfully connects to the server, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **serv_addr** − It is a pointer to struct sockaddr that contains destination IP address and port.
- **addrlen** − Set it to sizeof(struct sockaddr).

The *bind* Function

The *bind* function assigns a local protocol address to a socket. With the Internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number. This function is called by TCP server only.

```
int bind(int sockfd, struct sockaddr *my_addr,int addrlen);
```

This call returns 0 if it successfully binds to the address, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.

- **my_addr** − It is a pointer to struct sockaddr that contains the local IP address and port.
- **addrlen** − Set it to sizeof(struct sockaddr).

You can put your IP address and your port automatically

A 0 value for port number means that the system will choose a random port, and *INADDR_ANY* value for IP address means the server's IP address will be assigned automatically.

```
server.sin_port = 0;
server.sin_addr.s_addr = INADDR_ANY;
```

NOTE − All ports below 1024 are reserved. You can set a port above 1024 and below 65535 unless they are the ones being used by other programs.

The *listen* Function

The *listen* function is called only by a TCP server and it performs two actions −

- The listen function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.
- The second argument to this function specifies the maximum number of connections the kernel should queue for this socket.

```
int listen(int sockfd,int backlog);
```

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **backlog** − It is the number of allowed connections.

The *accept* Function

The *accept* function is called by a TCP server to return the next completed connection from the front of the completed connection queue. The signature of the call is as follows −

```
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

This call returns a non-negative descriptor on success, otherwise it returns -1 on error. The returned descriptor is assumed to be a client socket descriptor and all read-write operations will be done on this descriptor to communicate with the client.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.

- **cliaddr** − It is a pointer to struct sockaddr that contains client IP address and port.
- **addrlen** − Set it to sizeof(struct sockaddr).

The *send* Function

The *send* function is used to send data over stream sockets or CONNECTED datagram sockets. If you want to send data over UNCONNECTED datagram sockets, you must use sendto() function.

You can use *write()* system call to send data. Its signature is as follows −

```
int send(int sockfd, const void *msg, int len, int flags);
```

This call returns the number of bytes sent out, otherwise it will return -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **msg** − It is a pointer to the data you want to send.
- **len** − It is the length of the data you want to send (in bytes).
- **flags** − It is set to 0.

The *recv* Function

The *recv* function is used to receive data over stream sockets or CONNECTED datagram sockets. If you want to receive data over UNCONNECTED datagram sockets you must use recvfrom().

You can use *read()* system call to read the data. This call is explained in helper functions chapter.

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

This call returns the number of bytes read into the buffer, otherwise it will return -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **buf** − It is the buffer to read the information into.
- **len** − It is the maximum length of the buffer.
- **flags** − It is set to 0.

The *sendto* Function

The *sendto* function is used to send data over UNCONNECTED datagram sockets. Its signature is as follows −

int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);

This call returns the number of bytes sent, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **msg** − It is a pointer to the data you want to send.
- **len** − It is the length of the data you want to send (in bytes).
- **flags** − It is set to 0.
- **to** − It is a pointer to struct sockaddr for the host where data has to be sent.
- **tolen** − It is set it to sizeof(struct sockaddr).

The *recvfrom* Function

The *recvfrom* function is used to receive data from UNCONNECTED datagram sockets.

int recvfrom(int sockfd, void *buf, int len, unsigned int flags struct sockaddr *from, int *fromlen);

This call returns the number of bytes read into the buffer, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **buf** − It is the buffer to read the information into.
- **len** − It is the maximum length of the buffer.
- **flags** − It is set to 0.
- **from** − It is a pointer to struct sockaddr for the host where data has to be read.
- **fromlen** − It is set it to sizeof(struct sockaddr).

The *close* Function

The *close* function is used to close the communication between the client and the server. Its syntax is as follows −

int close(int sockfd);

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.

The *shutdown* Function

The *shutdown* function is used to gracefully close the communication between the client and the server. This function gives more control in comparison to the *close* function. Given below is the syntax of *shutdown* −

```
int shutdown(int sockfd, int how);
```

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **how** − Put one of the numbers −
    - **0** − indicates that receiving is not allowed,
    - **1** − indicates that sending is not allowed, and
    - **2** − indicates that both sending and receiving are not allowed. When *how* is set to 2, it's the same thing as close().

The *select* Function

The *select* function indicates which of the specified file descriptors is ready for reading, ready for writing, or has an error condition pending.

When an application calls *recv or recvfrom*, it is blocked until data arrives for that socket. An application could be doing other useful processing while the incoming data stream is empty. Another situation is when an application receives data from multiple sockets.

Calling *recv or recvfrom* on a socket that has no data in its input queue prevents immediate reception of data from other sockets. The select function call solves this problem by allowing the program to poll all the socket handles to see if they are available for non-blocking reading and writing operations.

Given below is the syntax of *select* −

```
int select(int  nfds, fd_set  *readfds, fd_set  *writefds, fd_set *errorfds, struct timeval
*timeout);
```

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

- **nfds** − It specifies the range of file descriptors to be tested. The select() function tests file descriptors in the range of 0 to nfds-1

- **readfds** − It points to an object of type *fd_set* that on input, specifies the file descriptors to be checked for being ready to read, and on output, indicates which file descriptors are ready to read. It can be NULL to indicate an empty set.
- **writefds** − It points to an object of type *fd_set* that on input, specifies the file descriptors to be checked for being ready to write, and on output, indicates which file descriptors are ready to write. It can be NULL to indicate an empty set.
- **exceptfds** − It points to an object of type *fd_set* that on input, specifies the file descriptors to be checked for error conditions pending, and on output indicates, which file descriptors have error conditions pending. It can be NULL to indicate an empty set.
- **timeout** − It points to a timeval struct that specifies how long the select call should poll the descriptors for an available I/O operation. If the timeout value is 0, then select will return immediately. If the timeout argument is NULL, then select will block until at least one file/socket handle is ready for an available I/O operation. Otherwise *select* will return after the amount of time in the timeout has elapsed OR when at least one file/socket descriptor is ready for an I/O operation.

The return value from select is the number of handles specified in the file descriptor sets that are ready for I/O. If the time limit specified by the timeout field is reached, select return 0. The following macros exist for manipulating a file descriptor set −

- **FD_CLR(fd, &fdset)** − Clears the bit for the file descriptor fd in the file descriptor set *fdset.*
- **FD_ISSET(fd, &fdset)** − Returns a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.
- **FD_SET(fd, &fdset)** − Sets the bit for the file descriptor fd in the file descriptor set fdset.
- **FD_ZERO(&fdset)** − Initializes the file descriptor set fdset to have zero bits for all file descriptors.

The behavior of these macros is undefined if the fd argument is less than 0 or greater than or equal to FD_SETSIZE.

Socket - Helper Functions

This chapter describes all the helper functions, which are used while doing socket programming. Other helper functions are described in the chapters −**Ports and Services**, and Network **Byte Orders**.

The *write* Function

The *write* function attempts to write nbyte bytes from the buffer pointed by *buf* to the file associated with the open file descriptor, *fildes*.

You can also use *send()* function to send data to another process.

```
#include <unistd.h>

int write(int fildes, const void *buf, int nbyte);
```

Upon successful completion, write() returns the number of bytes actually written to the file associated with fildes. This number is never greater than nbyte. Otherwise, -1 is returned.

Parameters

- **fildes** − It is a socket descriptor returned by the socket function.
- **buf** − It is a pointer to the data you want to send.
- **nbyte** − It is the number of bytes to be written. If nbyte is 0, write() will return 0 and have no other results if the file is a regular file; otherwise, the results are unspecified.

The *read* Function

The *read* function attempts to read nbyte bytes from the file associated with the buffer, fildes, into the buffer pointed to by buf.

You can also use *recv()* function to read data to another process.

```
int read(int fildes, const void *buf, int nbyte);
```

Upon successful completion, write() returns the number of bytes actually written to the file associated with fildes. This number is never greater than nbyte. Otherwise, -1 is returned.

Parameters

- **fildes** − It is a socket descriptor returned by the socket function.
- **buf** − It is the buffer to read the information into.
- **nbyte** − It is the number of bytes to read.

The *fork* Function

The *fork* function creates a new process. The new process called the child process will be an exact copy of the calling process (parent process). The child process inherits many attributes from the parent process.

```
int fork(void);
```

Upon successful completion, fork() returns 0 to the child process and the process ID of the child process to the parent process. Otherwise -1 is returned to the parent process, no child process is created and errno is set to indicate the error.

Parameters

- **void** − It means no parameter is required.

The *bzero* Function

The *bzero* function places *nbyte* null bytes in the string *s*. This function is used to set all the socket structures with null values.

void bzero(void *s, int nbyte);

This function does not return anything.

Parameters

- **s** − It specifies the string which has to be filled with null bytes. This will be a point to socket structure variable.
- **nbyte** − It specifies the number of bytes to be filled with null values. This will be the size of the socket structure.

The *bcmp* Function

The *bcmp* function compares byte string s1 against byte string s2. Both strings are assumed to be nbyte bytes long.

```
int bcmp(const void *s1, const void *s2, int nbyte);
```

This function returns 0 if both strings are identical, 1 otherwise. The bcmp() function always returns 0 when nbyte is 0.

Parameters

- **s1** − It specifies the first string to be compared.
- **s2** − It specifies the second string to be compared.
- **nbyte** − It specifies the number of bytes to be compared.

The *bcopy* Function

The *bcopy* function copies nbyte bytes from string s1 to the string s2. Overlapping strings are handled correctly.

```
void bcopy(const void *s1, void *s2, int nbyte);
```

This function does not return anything.

Parameters

- **s1** − It specifies the source string.
- **s2v** − It specifies the destination string.

- **nbyte** − It specifies the number of bytes to be copied.

The *memset* Function

The *memset* function is also used to set structure variables in the same way as **bzero**. Take a look at its syntax, given below.

void *memset(void *s, int c, int nbyte);

This function returns a pointer to void; in fact, a pointer to the set memory and you need to caste it accordingly.

Parameters

- **s** − It specifies the source to be set.
- **c** − It specifies the character to set on nbyte places.
- **nbyte** − It specifies the number of bytes to be set.

Unix Socket - Server Examples

To make a process a TCP server, you need to follow the steps given below −

- Create a socket with the *socket()* system call.
- Bind the socket to an address using the *bind()* system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the *listen()* system call.
- Accept a connection with the *accept()* system call. This call typically blocks until a client connects with the server.
- Send and receive data using the *read()* and *write()* system calls.

Now let us put these steps in the form of source code. Put this code into the file *server.c* and compile it with *gcc* compiler.

```
#include <stdio.h>
#include <stdlib.h>

#include <netdb.h>
#include <netinet/in.h>

#include <string.h>

int main( int argc, char *argv[] ) {
   int sockfd, newsockfd, portno, clilen;
   char buffer[256];
   struct sockaddr_in serv_addr, cli_addr;
   int  n;
```

```
/* First call to socket() function */
sockfd = socket(AF_INET, SOCK_STREAM, 0);

if (sockfd < 0) {
   perror("ERROR opening socket");
   exit(1);
}

/* Initialize socket structure */
bzero((char *) &serv_addr, sizeof(serv_addr));
portno = 5001;

serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);

/* Now bind the host address using bind() call.*/
if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
   perror("ERROR on binding");
   exit(1);
}

/* Now start listening for the clients, here process will
   * go in sleep mode and will wait for the incoming connection
*/

listen(sockfd,5);
clilen = sizeof(cli_addr);

/* Accept actual connection from the client */
newsockfd = accept(sockfd, (struct sockaddr *)&cli_addr, &clilen);

if (newsockfd < 0) {
   perror("ERROR on accept");
   exit(1);
}

/* If connection is established then start communicating */
bzero(buffer,256);
n = read( newsockfd,buffer,255 );

if (n < 0) {
   perror("ERROR reading from socket");
   exit(1);
}

printf("Here is the message: %s\n",buffer);

/* Write a response to the client */
```

```
   n = write(newsockfd,"I got your message",18);

   if (n < 0) {
      perror("ERROR writing to socket");
      exit(1);
   }

   return 0;
}
```

Handle Multiple Connections

To allow the server to handle multiple simultaneous connections, we make the following changes in the above code −

- Put the *accept* statement and the following code in an infinite loop.
- After a connection is established, call *fork()* to create a new process.
- The child process will close *sockfd* and call *doprocessing* function, passing the new socket file descriptor as an argument. When the two processes have completed their conversation, as indicated by *doprocessing()* returning, this process simply exits.
- The parent process closes *newsockfd*. As all of this code is in an infinite loop, it will return to the accept statement to wait for the next connection.

```
#include <stdio.h>
#include <stdlib.h>

#include <netdb.h>
#include <netinet/in.h>

#include <string.h>

void doprocessing (int sock);

int main( int argc, char *argv[] ) {
   int sockfd, newsockfd, portno, clilen;
   char buffer[256];
   struct sockaddr_in serv_addr, cli_addr;
   int n, pid;

   /* First call to socket() function */
   sockfd = socket(AF_INET, SOCK_STREAM, 0);

   if (sockfd < 0) {
      perror("ERROR opening socket");
      exit(1);
```

```
}

/* Initialize socket structure */
bzero((char *) &serv_addr, sizeof(serv_addr));
portno = 5001;

serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);

/* Now bind the host address using bind() call.*/
if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
  perror("ERROR on binding");
  exit(1);
}

/* Now start listening for the clients, here
  * process will go in sleep mode and will wait
  * for the incoming connection
*/

listen(sockfd,5);
clilen = sizeof(cli_addr);

while (1) {
  newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);

  if (newsockfd < 0) {
    perror("ERROR on accept");
    exit(1);
  }

  /* Create child process */
  pid = fork();

  if (pid < 0) {
    perror("ERROR on fork");
    exit(1);
  }

  if (pid == 0) {
    /* This is the client process */
    close(sockfd);
    doprocessing(newsockfd);
    exit(0);
  }
  else {
    close(newsockfd);
  }
```

```
   } /* end of while */
}
```

Socket - Client Examples

To make a process a TCP client, you need to follow the steps given below &minus ;

- Create a socket with the *socket()* system call.
- Connect the socket to the address of the server using the *connect()* system call.
- Send and receive data. There are a number of ways to do this, but the simplest way is to use the *read()* and *write()* system calls.

Now let us put these steps in the form of source code. Put this code into the file **client.c** and compile it with **gcc** compiler.

Run this program and pass *hostname* and *port number* of the server, to connect to the server, which you already must have run in another Unix window.

```
#include <stdio.h>
#include <stdlib.h>

#include <netdb.h>
#include <netinet/in.h>

#include <string.h>

int main(int argc, char *argv[]) {
   int sockfd, portno, n;
   struct sockaddr_in serv_addr;
   struct hostent *server;

   char buffer[256];

   if (argc < 3) {
      fprintf(stderr,"usage %s hostname port\n", argv[0]);
      exit(0);
   }

   portno = atoi(argv[2]);

   /* Create a socket point */
   sockfd = socket(AF_INET, SOCK_STREAM, 0);

   if (sockfd < 0) {
      perror("ERROR opening socket");
      exit(1);
   }
   server = gethostbyname(argv[1]);

   if (server == NULL) {
      fprintf(stderr,"ERROR, no such host\n");
      exit(0);
   }

   bzero((char *) &serv_addr, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;
   bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr, server->h_length);
   serv_addr.sin_port = htons(portno);

   /* Now connect to the server */
   if (connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
      perror("ERROR connecting");
      exit(1);
   }

   /* Now ask for a message from the user, this message
      * will be read by server
   */

   printf("Please enter the message: ");
   bzero(buffer,256);
   fgets(buffer,255,stdin);

   /* Send message to the server */
   n = write(sockfd, buffer, strlen(buffer));

   if (n < 0) {
      perror("ERROR writing to socket");
      exit(1);
   }

   /* Now read server response */
   bzero(buffer,256);
   n = read(sockfd, buffer, 255);

   if (n < 0) {
      perror("ERROR reading from socket");
      exit(1);
   }

   printf("%s\n",buffer);
   return 0;
}
```

Socket - Summary

Here is a list of all the functions related to socket programming.

Port and Service Functions

Unix provides the following functions to fetch service name from the /etc/services file.

- **struct servent *getservbyname(char *name, char *proto)** − This call takes a service name and a protocol name and returns the corresponding port number for that service.
- **struct servent *getservbyport(int port, char *proto)** − This call takes a port number and a protocol name and returns the corresponding service name.

Byte Ordering Functions

- **unsigned short htons (unsigned short hostshort)** − This function converts 16-bit (2-byte) quantities from host byte order to network byte order.
- **unsigned long htonl (unsigned long hostlong)** − This function converts 32-bit (4-byte) quantities from host byte order to network byte order.
- **unsigned short ntohs (unsigned short netshort)** − This function converts 16-bit (2-byte) quantities from network byte order to host byte order.
- **unsigned long ntohl (unsigned long netlong)** − This function converts 32-bit quantities from network byte order to host byte order.

IP Address Functions

- **int inet_aton (const char *strptr, struct in_addr *addrptr)** − This function call converts the specified string, in the Internet standard dot notation, to a network address, and stores the address in the structure provided. The converted address will be in Network Byte Order (bytes ordered from left to right). It returns 1 if the string is valid and 0 on error.
- **in_addr_t inet_addr (const char *strptr)** − This function call converts the specified string, in the Internet standard dot notation, to an integer value suitable for use as an Internet address. The converted address will be in Network Byte Order (bytes ordered from left to right). It returns a 32-bit binary network byte ordered IPv4 address and INADDR_NONE on error.
- **char *inet_ntoa (struct in_addr inaddr)** − This function call converts the specified Internet host address to a string in the Internet standard dot notation.

Socket Core Functions

- **int socket (int family, int type, int protocol)** − This call returns a socket descriptor that you can use in later system calls or it gives you -1 on error.
- **int connect (int sockfd, struct sockaddr *serv_addr, int addrlen)** − The connect function is used by a TCP client to establish a connection with a TCP server. This call returns 0 if it successfully connects to the server, otherwise it returns -1.
- **int bind(int sockfd, struct sockaddr *my_addr,int addrlen)** − The bind function assigns a local protocol address to a socket. This call returns 0 if it successfully binds to the address, otherwise it returns -1.
- **int listen(int sockfd, int backlog)** − The listen function is called only by a TCP server to listen for the client request. This call returns 0 on success, otherwise it returns -1.
- **int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen)** − The accept function is called by a TCP server to accept client requests and to establish actual connection. This call returns a non-negative descriptor on success, otherwise it returns -1.
- **int send(int sockfd, const void *msg, int len, int flags)** − The send function is used to send data over stream sockets or CONNECTED datagram sockets. This call returns the number of bytes sent out, otherwise it returns -1.
- **int recv (int sockfd, void *buf, int len, unsigned int flags)** − The recv function is used to receive data over stream sockets or CONNECTED datagram sockets. This call returns the number of bytes read into the buffer, otherwise it returns -1 on error.
- **int sendto (int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen)** − The sendto function is used to send data over UNCONNECTED datagram sockets. This call returns the number of bytes sent, otherwise it returns -1 on error.
- **int recvfrom (int sockfd, void *buf, int len, unsigned int flags struct sockaddr *from, int *fromlen)** − The recvfrom function is used to receive data from UNCONNECTED datagram sockets. This call returns the number of bytes read into the buffer, otherwise it returns -1 on error.
- **int close (int sockfd)** − The close function is used to close a communication between the client and the server. This call returns 0 on success, otherwise it returns -1.
- **int shutdown (int sockfd, int how)** − The shutdown function is used to gracefully close a communication between the client and the server. This function gives more control in comparison to close function. It returns 0 on success, -1 otherwise.
- **int select (int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout)** − This function is used to read or write multiple sockets.

Socket Helper Functions

- **int write (int fildes, const void *buf, int nbyte)** − The write function attempts to write nbyte bytes from the buffer pointed to by buf to the file associated with the open file descriptor, fildes. Upon successful completion, write() returns the number of bytes actually written to the file associated with fildes. This number is never greater than nbyte. Otherwise, -1 is returned.
- **int read (int fildes, const void *buf, int nbyte)** − The read function attempts to read nbyte bytes from the file associated with the open file descriptor, fildes, into the buffer pointed to by buf. Upon successful completion, write() returns the number of bytes actually written to the file associated with fildes. This number is never greater than nbyte. Otherwise, -1 is returned.
- **int fork (void)** − The fork function creates a new process. The new process, called the child process, will be an exact copy of the calling process (parent process).
- **void bzero (void *s, int nbyte)** − The bzero function places nbyte null bytes in the string s. This function will be used to set all the socket structures with null values.
- **int bcmp (const void *s1, const void *s2, int nbyte)** − The bcmp function compares the byte string s1 against the byte string s2. Both the strings are assumed to be nbyte bytes long.
- **void bcopy (const void *s1, void *s2, int nbyte)** − The bcopy function copies nbyte bytes from the string s1 to the string s2. Overlapping strings are handled correctly.
- **void *memset(void *s, int c, int nbyte)** − The memset function is also used to set structure variables in the same way as bzero. as bzero.

Shared memory IPC mechanism

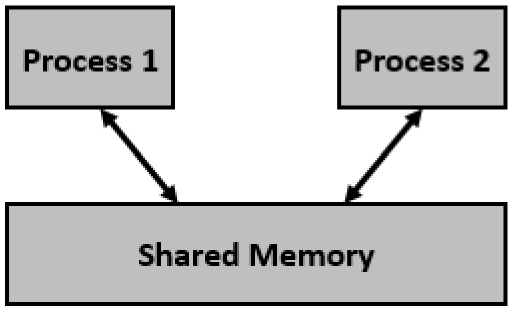
Shared memory is a memory shared between two or more processes. However, why do we need to share memory or some other means of communication?

Each process has its own address space, if any process wants to communicate with some information from its own address space to other processes, then it is only possible with IPC (inter process communication) techniques. As we are already aware, communication can be between related or unrelated processes.

Usually, inter-related process communication is performed using Pipes or Named Pipes. Unrelated processes (say one process running in one terminal and another process in another terminal) communication can be performed using Named Pipes or through popular IPC techniques of Shared Memory and Message Queues.

We have seen the IPC techniques of Pipes and Named pipes and now it is time to know the remaining IPC techniques viz., Shared Memory, Message Queues, Semaphores, Signals, and Memory Mapping.

In this chapter, we will know all about shared memory.



We know that to communicate between two or more processes, we use shared memory but before using the shared memory what needs to be done with the system calls, let us see this −

- Create the shared memory segment or use an already created shared memory segment (shmget())
- Attach the process to the already created shared memory segment (shmat())
- Detach the process from the already attached shared memory segment (shmdt())
- Control operations on the shared memory segment (shmctl())

Let us look at a few details of the system calls related to shared memory.

```
int shmget(key_t key, size_t size, int shmflg)
```

The above system call creates or allocates a System V shared memory segment. The arguments that need to be passed are as follows −

The **first argument, key,** recognizes the shared memory segment. The key can be either an arbitrary value or one that can be derived from the library function ftok(). The key can also be IPC_PRIVATE, means, running processes as server and client (parent and child relationship) i.e., inter-related process communiation. If the client wants to use shared memory with this key, then it must be a child process of the server. Also, the child process needs to be created after the parent has obtained a shared memory.

The **second argument, size,** is the size of the shared memory segment rounded to multiple of PAGE_SIZE.

The **third argument, shmflg,** specifies the required shared memory flag/s such as IPC_CREAT (creating new segment) or IPC_EXCL (Used with IPC_CREAT to create new segment and the call fails, if the segment already exists). Need to pass the permissions as well.

Note − Refer earlier sections for details on permissions.

This call would return a valid shared memory identifier (used for further calls of shared memory) on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
void * shmat(int shmid, const void *shmaddr, int shmflg)
```

The above system call performs shared memory operation for System V shared memory segment i.e., attaching a shared memory segment to the address space of the calling process. The arguments that need to be passed are as follows −

The first argument, shmid, is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

The second argument, shmaddr, is to specify the attaching address. If shmaddr is NULL, the system by default chooses the suitable address to attach the segment. If shmaddr is not NULL and SHM_RND is specified in shmflg, the attach is equal to the address of the nearest multiple of SHMLBA (Lower Boundary Address). Otherwise, shmaddr must be a page aligned address at which the shared memory attachment occurs/starts.

The third argument, shmflg, specifies the required shared memory flag/s such as SHM_RND (rounding off address to SHMLBA) or SHM_EXEC (allows the contents of segment to be executed) or SHM_RDONLY (attaches the segment for read-only purpose, by default it is read-write) or SHM_REMAP (replaces the existing mapping in the range specified by shmaddr and continuing till the end of segment).

This call would return the address of attached shared memory segment on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
int shmdt(const void *shmaddr)
```

The above system call performs shared memory operation for System V shared memory segment of detaching the shared memory segment from the address space of the calling process. The argument that needs to be passed is −

The argument, shmaddr, is the address of shared memory segment to be detached. The to-be-detached segment must be the address returned by the shmat() system call.

This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

The above system call performs control operation for a System V shared memory segment. The following arguments needs to be passed −

The first argument, shmid, is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

The second argument, cmd, is the command to perform the required control operation on the shared memory segment.

Valid values for cmd are −

- **IPC_STAT** − Copies the information of the current values of each member of struct shmid_ds to the passed structure pointed by buf. This command requires read permission to the shared memory segment.
- **IPC_SET** − Sets the user ID, group ID of the owner, permissions, etc. pointed to by structure buf.
- **IPC_RMID** − Marks the segment to be destroyed. The segment is destroyed only after the last process has detached it.
- **IPC_INFO** − Returns the information about the shared memory limits and parameters in the structure pointed by buf.
- **SHM_INFO** − Returns a shm_info structure containing information about the consumed system resources by the shared memory.

The third argument, buf, is a pointer to the shared memory structure named struct shmid_ds. The values of this structure would be used for either set or get as per cmd.

This call returns the value depending upon the passed command. Upon success of IPC_INFO and SHM_INFO or SHM_STAT returns the index or identifier of the shared memory segment or 0 for other operations and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

Let us consider the following sample program.

- Create two processes, one is for writing into the shared memory (shm_write.c) and another is for reading from the shared memory (shm_read.c)
- The program performs writing into the shared memory by write process (shm_write.c) and reading from the shared memory by reading process (shm_read.c)
- In the shared memory, the writing process, creates a shared memory of size 1K (and flags) and attaches the shared memory
- The write process writes 5 times the Alphabets from 'A' to 'E' each of 1023 bytes into the shared memory. Last byte signifies the end of buffer
- Read process would read from the shared memory and write to the standard output
- Reading and writing process actions are performed simultaneously
- After completion of writing, the write process updates to indicate completion of writing into the shared memory (with complete variable in struct shmseg)
- Reading process performs reading from the shared memory and displays on the output until it gets indication of write process completion (complete variable in struct shmseg)
- Performs reading and writing process for a few times for simplication and also in order to avoid infinite loops and complicating the program

Following is the code for write process (Writing into Shared Memory – File: shm_write.c)

```
/* Filename: shm_write.c */
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<string.h>
#include<errno.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>

#define BUF_SIZE 1024
#define SHM_KEY 0x1234

struct shmseg {
   int cnt;
   int complete;
   char buf[BUF_SIZE];
};
int fill_buffer(char * bufptr, int size);

int main(int argc, char *argv[]) {
   int shmid, numtimes;
   struct shmseg *shmp;
   char *bufptr;
   int spaceavailable;
   shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT);
   if (shmid == -1) {
      perror("Shared memory");
      return 1;
   }

   // Attach to the segment to get a pointer to it.
   shmp = shmat(shmid, NULL, 0);
   if (shmp == (void *) -1) {
      perror("Shared memory attach");
      return 1;
   }

   /* Transfer blocks of data from buffer to shared memory */
   bufptr = shmp->buf;
   spaceavailable = BUF_SIZE;
   for (numtimes = 0; numtimes < 5; numtimes++) {
      shmp->cnt = fill_buffer(bufptr, spaceavailable);
      shmp->complete = 0;
      printf("Writing Process: Shared Memory Write: Wrote %d bytes\n", shmp->cnt);
      bufptr = shmp->buf;
      spaceavailable = BUF_SIZE;
      sleep(3);
   }
   printf("Writing Process: Wrote %d times\n", numtimes);
   shmp->complete = 1;

   if (shmdt(shmp) == -1) {
      perror("shmdt");
      return 1;
   }

   if (shmctl(shmid, IPC_RMID, 0) == -1) {
      perror("shmctl");
      return 1;
   }
   printf("Writing Process: Complete\n");
   return 0;
}

int fill_buffer(char * bufptr, int size) {
   static char ch = 'A';
   int filled_count;

   //printf("size is %d\n", size);
   memset(bufptr, ch, size - 1);
   bufptr[size-1] = '\0';
   if (ch > 122)
   ch = 65;
   if ( (ch >= 65) && (ch <= 122) ) {
      if ( (ch >= 91) && (ch <= 96) ) {
         ch = 65;
      }
   }
   filled_count = strlen(bufptr);

   //printf("buffer count is: %d\n", filled_count);
   //printf("buffer filled is:%s\n", bufptr);
   ch++;
   return filled_count;
}
```

Following is the code for read process (Reading from the Shared Memory and writing to the standard output – File: shm_read.c)

```
/* Filename: shm_read.c */
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<string.h>
```

```
#include<errno.h>
#include<stdlib.h>

#define BUF_SIZE 1024
#define SHM_KEY 0x1234

struct shmseg {
  int cnt;
  int complete;
  char buf[BUF_SIZE];
};

int main(int argc, char *argv[]) {
  int shmid;
  struct shmseg *shmp;
  shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT);
  if (shmid == -1) {
    perror("Shared memory");
    return 1;
  }

  // Attach to the segment to get a pointer to it.
  shmp = shmat(shmid, NULL, 0);
  if (shmp == (void *) -1) {
    perror("Shared memory attach");
    return 1;
  }

  /* Transfer blocks of data from shared memory to stdout*/
  while (shmp->complete != 1) {
    printf("segment contains : \n\"%s\"\n", shmp->buf);
    if (shmp->cnt == -1) {
      perror("read");
      return 1;
    }
    printf("Reading Process: Shared Memory: Read %d bytes\n", shmp->cnt);
    sleep(3);
  }
  printf("Reading Process: Reading Done, Detaching Shared Memory\n");
  if (shmdt(shmp) == -1) {
    perror("shmdt");
    return 1;
  }
  printf("Reading Process: Complete\n");
  return 0;
}
```

Message queue IPC mechanism

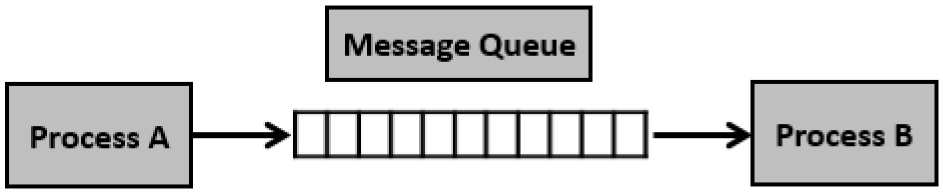
Why do we need message queues when we already have the shared memory? It would be for multiple reasons, let us try to break this into multiple points for simplification −

- As understood, once the message is received by a process it would be no longer available for any other process. Whereas in shared memory, the data is available for multiple processes to access.
- If we want to communicate with small message formats.
- Shared memory data need to be protected with synchronization when multiple processes communicating at the same time.
- Frequency of writing and reading using the shared memory is high, then it would be very complex to implement the functionality. Not worth with regard to utilization in this kind of cases.
- What if all the processes do not need to access the shared memory but very few processes only need it, it would be better to implement with message queues.
- If we want to communicate with different data packets, say process A is sending message type 1 to process B, message type 10 to process C, and message type 20 to process D. In this case, it is simplier to implement with message queues. To simplify the given message type as 1, 10, 20, it can be either 0 or +ve or –ve as discussed below.
- Ofcourse, the order of message queue is FIFO (First In First Out). The first message inserted in the queue is the first one to be retrieved.

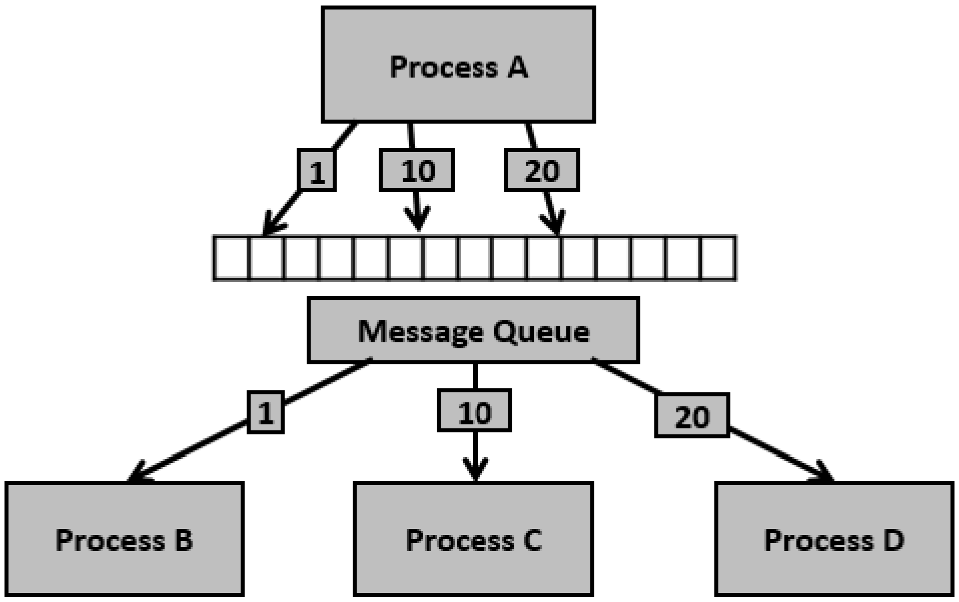
Using Shared Memory or Message Queues depends on the need of the application and how effectively it can be utilized.

Communication using message queues can happen in the following ways −

- Writing into the shared memory by one process and reading from the shared memory by another process. As we are aware, reading can be done with multiple processes as well.



- Writing into the shared memory by one process with different data packets and reading from it by multiple processes, i.e., as per message type.



Having seen certain information on message queues, now it is time to check for the system call (System V) which supports the message queues.

To perform communication using message queues, following are the steps −

Step 1 − Create a message queue or connect to an already existing message queue (msgget())

Step 2 − Write into message queue (msgsnd())

Step 3 − Read from the message queue (msgrcv())

Step 4 − Perform control operations on the message queue (msgctl())

Now, let us check the syntax and certain information on the above calls.

```
int msgget(key_t key, int msgflg)
```

This system call creates or allocates a System V message queue. Following arguments need to be passed −

- The first argument, key, recognizes the message queue. The key can be either an arbitrary value or one that can be derived from the library function ftok().
- The second argument, shmflg, specifies the required message queue flag/s such as IPC_CREAT (creating message queue if not exists) or IPC_EXCL (Used with IPC_CREAT to create the message queue and the call fails, if the message queue already exists). Need to pass the permissions as well.

Note − Refer earlier sections for details on permissions.

This call would return a valid message queue identifier (used for further calls of message queue) on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

Various errors with respect to this call are EACCESS (permission denied), EEXIST (queue already exists can't create), ENOENT (queue doesn't exist), ENOMEM (not enough memory to create the queue), etc.

```
int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg)
```

This system call sends/appends a message into the message queue (System V). Following arguments need to be passed −

- The first argument, msgid, recognizes the message queue i.e., message queue identifier. The identifier value is received upon the success of msgget()
- The second argument, msgp, is the pointer to the message, sent to the caller, defined in the structure of the following form −

```
struct msgbuf {
  long mtype;
  char mtext[1];
};
```

The variable mtype is used for communicating with different message types, explained in detail in msgrcv() call. The variable mtext is an array or other structure whose size is specified by msgsz (positive value). If the mtext field is not mentioned, then it is considered as zero size message, which is permitted.

- The third argument, msgsz, is the size of message (the message should end with a null character)
- The fourth argument, msgflg, indicates certain flags such as IPC_NOWAIT (returns immediately when no message is found in queue or MSG_NOERROR (truncates message text, if more than msgsz bytes)

This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
int msgrcv(int msgid, const void *msgp, size_t msgsz, long msgtype, int msgflg)
```

This system call retrieves the message from the message queue (System V). Following arguments need to be passed −

- The first argument, msgid, recognizes the message queue i.e., the message queue identifier. The identifier value is received upon the success of msgget()
- The second argument, msgp, is the pointer of the message received from the caller. It is defined in the structure of the following form −

```
struct msgbuf {
  long mtype;
  char mtext[1];
};
```

The variable mtype is used for communicating with different message types. The variable mtext is an array or other structure whose size is specified by msgsz (positive value). If the mtext field is not mentioned, then it is considered as zero size message, which is permitted.

- The third argument, msgsz, is the size of the message received (message should end with a null character)
- The fouth argument, msgtype, indicates the type of message −
  - **If msgtype is 0** − Reads the first received message in the queue
  - **If msgtype is +ve** − Reads the first message in the queue of type msgtype (if msgtype is 10, then reads only the first message of type 10 even though other types may be in the queue at the beginning)
  - **If msgtype is –ve** − Reads the first message of lowest type less than or equal to the absolute value of message type (say, if msgtype is -5, then it reads first message of type less than 5 i.e., message type from 1 to 5)
- The fifth argument, msgflg, indicates certain flags such as IPC_NOWAIT (returns immediately when no message is found in the queue or MSG_NOERROR (truncates the message text if more than msgsz bytes)

This call would return the number of bytes actually received in mtext array on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
int msgctl(int msgid, int cmd, struct msqid_ds *buf)
```

This system call performs control operations of the message queue (System V). Following arguments need to be passed −

- The first argument, msgid, recognizes the message queue i.e., the message queue identifier. The identifier value is received upon the success of msgget()
- The second argument, cmd, is the command to perform the required control operation on the message queue. Valid values for cmd are −

IPC_STAT − Copies information of the current values of each member of struct msqid_ds to the passed structure pointed by buf. This command requires read permission on the message queue.

IPC_SET − Sets the user ID, group ID of the owner, permissions etc pointed to by structure buf.

IPC_RMID − Removes the message queue immediately.

IPC_INFO − Returns information about the message queue limits and parameters in the structure pointed by buf, which is of type struct msginfo

MSG_INFO − Returns an msginfo structure containing information about the consumed system resources by the message queue.

- The third argument, buf, is a pointer to the message queue structure named struct msqid_ds. The values of this structure would be used for either set or get as per cmd.

This call would return the value depending on the passed command. Success of IPC_INFO and MSG_INFO or MSG_STAT returns the index or identifier of the message queue or 0 for other operations and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

Having seen the basic information and system calls with regard to message queues, now it is time to check with a program.

Let us see the description before looking at the program −

Step 1 − Create two processes, one is for sending into message queue (msgq_send.c) and another is for retrieving from the message queue (msgq_recv.c)

Step 2 − Creating the key, using ftok() function. For this, initially file msgq.txt is created to get a unique key.

Step 3 − The sending process performs the following.

- Reads the string input from the user
- Removes the new line, if it exists
- Sends into message queue
- Repeats the process until the end of input (CTRL + D)
- Once the end of input is received, sends the message "end" to signify the end of the process

Step 4 − In the receiving process, performs the following.

- Reads the message from the queue
- Displays the output
- If the received message is "end", finishes the process and exits

To simplify, we are not using the message type for this sample. Also, one process is writing into the queue and another process is reading from the queue. This can be extended as needed i.e., ideally one process would write into the queue and multiple processes read from the queue.

Now, let us check the process (message sending into queue) – File: msgq_send.c

```
/* Filename: msgq_send.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
```

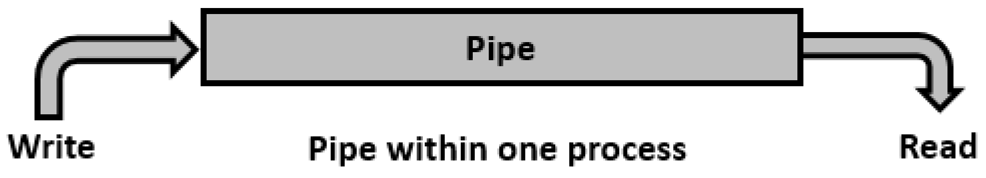
```
#include <sys/msg.h>

#define PERMS 0644
struct my_msgbuf {
   long mtype;
   char mtext[200];
};

int main(void) {
   struct my_msgbuf buf;
   int msqid;
   int len;
   key_t key;
   system("touch msgq.txt");

   if ((key = ftok("msgq.txt", 'B')) == -1) {
      perror("ftok");
      exit(1);
   }

   if ((msqid = msgget(key, PERMS | IPC_CREAT)) == -1) {
      perror("msgget");
      exit(1);
   }
   printf("message queue: ready to send messages.\n");
   printf("Enter lines of text, ^D to quit:\n");
   buf.mtype = 1; /* we don't really care in this case */

   while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {
      len = strlen(buf.mtext);
      /* remove newline at end, if it exists */
      if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';
      if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
      perror("msgsnd");
   }
   strcpy(buf.mtext, "end");
   len = strlen(buf.mtext);
   if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
   perror("msgsnd");

   if (msgctl(msqid, IPC_RMID, NULL) == -1) {
      perror("msgctl");
      exit(1);
   }
   printf("message queue: done sending messages.\n");
   return 0;
}

/* Filename: msgq_recv.c */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define PERMS 0644
struct my_msgbuf {
   long mtype;
   char mtext[200];
};

int main(void) {
   struct my_msgbuf buf;
   int msqid;
   int toend;
   key_t key;

   if ((key = ftok("msgq.txt", 'B')) == -1) {
      perror("ftok");
      exit(1);
   }

   if ((msqid = msgget(key, PERMS)) == -1) { /* connect to the queue */
      perror("msgget");
      exit(1);
   }
   printf("message queue: ready to receive messages.\n");

   for(;;) { /* normally receiving never ends but just to make conclusion
             /* this program ends wuth string of end */
      if (msgrcv(msqid, &buf, sizeof(buf.mtext), 0, 0) == -1) {
         perror("msgrcv");
         exit(1);
      }
      printf("recvd: \"%s\"\n", buf.mtext);
      toend = strcmp(buf.mtext,"end");
      if (toend == 0)
      break;
   }
   printf("message queue: done receiving messages.\n");
   system("rm msgq.txt");
   return 0;
}
```

Pipe IPC mechanisms

Pipe is a communication medium between two or more related or interrelated processes. It can be either within one process or a communication between the child and the parent processes. Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc. Communication is achieved by one process writing into the pipe and other reading from the pipe. To achieve the pipe system call, create two files, one to write into the file and another to read from the file.

Pipe mechanism can be viewed with a real-time scenario such as filling water with the pipe into some container, say a bucket, and someone retrieving it, say with a mug. The filling process is nothing but writing into the pipe and the reading process is nothing but retrieving from the pipe. This implies that one output (water) is input for the other (bucket).



```
#include<unistd.h>

int pipe(int pipedes[2]);
```

This system call would create a pipe for one-way communication i.e., it creates two descriptors, first one is connected to read from the pipe and other one is connected to write into the pipe.

Descriptor pipedes[0] is for reading and pipedes[1] is for writing. Whatever is written into pipedes[1] can be read from pipedes[0].

This call would return zero on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Even though the basic operations for file are read and write, it is essential to open the file before performing the operations and closing the file after completion of the required operations. Usually, by default, 3 descriptors opened for every process, which are used for input (standard input – stdin), output (standard output – stdout) and error (standard error – stderr) having file descriptors 0, 1 and 2 respectively.

This system call would return a file descriptor used for further file operations of read/write/seek (lseek). Usually file descriptors start from 3 and increase by one number as the number of files open.

The arguments passed to open system call are pathname (relative or absolute path), flags mentioning the purpose of opening file (say, opening for read, O_RDONLY, to write, O_WRONLY, to read and write, O_RDWR, to append to the existing file O_APPEND, to create file, if not exists with O_CREAT and so on) and the required mode providing permissions of read/write/execute for user or owner/group/others. Mode can be mentioned with symbols.

Read – 4, Write – 2 and Execute – 1.

For example: Octal value (starts with 0), 0764 implies owner has read, write and execute permissions, group has read and write permissions, other has read permissions. This can also be represented as S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH, which implies or operation of 0700|0040|0020|0004 → 0764.

This system call, on success, returns the new file descriptor id and -1 in case of error. The cause of error can be identified with errno variable or perror() function.

```
int close(int fd)
```

The above system call closing already opened file descriptor. This implies the file is no longer in use and resources associated can be reused by any other process. This system call returns zero on success and -1 in case of error. The cause of error can be identified with errno variable or perror() function.

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

The above system call is to read from the specified file with arguments of file descriptor fd, proper buffer with allocated memory (either static or dynamic) and the size of buffer.

The file descriptor id is to identify the respective file, which is returned after calling open() or pipe() system call. The file needs to be opened before reading from the file. It automatically opens in case of calling pipe() system call.

This call would return the number of bytes read (or zero in case of encountering the end of the file) on success and -1 in case of failure. The return bytes can be smaller than the number of bytes requested, just in case no data is available or file is closed. Proper error number is set in case of failure.

To know the cause of failure, check with errno variable or perror() function.

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

The above system call is to write to the specified file with arguments of the file descriptor fd, a proper buffer with allocated memory (either static or dynamic) and the size of buffer.

The file descriptor id is to identify the respective file, which is returned after calling open() or pipe() system call.

The file needs to be opened before writing to the file. It automatically opens in case of calling pipe() system call.

This call would return the number of bytes written (or zero in case nothing is written) on success and -1 in case of failure. Proper error number is set in case of failure.

To know the cause of failure, check with errno variable or perror() function.

Example Programs

Following are some example programs.

Example program 1 − Program to write and read two messages using pipe.

Algorithm

Step 1 − Create a pipe.

Step 2 − Send a message to the pipe.

Step 3 − Retrieve the message from the pipe and write it to the standard output.

Step 4 − Send another message to the pipe.

Step 5 − Retrieve the message from the pipe and write it to the standard output.

Note − Retrieving messages can also be done after sending all messages.

Source Code: simplepipe.c

```
#include<stdio.h>
#include<unistd.h>

int main() {
   int pipefds[2];
   int returnstatus;
   char writemessages[2][20]={"Hi", "Hello"};
   char readmessage[20];
   returnstatus = pipe(pipefds);

   if (returnstatus == -1) {
      printf("Unable to create pipe\n");
      return 1;
   }

   printf("Writing to pipe - Message 1 is %s\n", writemessages[0]);
   write(pipefds[1], writemessages[0], sizeof(writemessages[0]));
   read(pipefds[0], readmessage, sizeof(readmessage));
   printf("Reading from pipe – Message 1 is %s\n", readmessage);
   printf("Writing to pipe - Message 2 is %s\n", writemessages[0]);
   write(pipefds[1], writemessages[1], sizeof(writemessages[0]));
   read(pipefds[0], readmessage, sizeof(readmessage));
   printf("Reading from pipe – Message 2 is %s\n", readmessage);
   return 0;
}
```

Note − Ideally, return status needs to be checked for every system call. To simplify the process, checks are not done for all the calls.

Execution Steps

Compilation

Example program 2 − Program to write and read two messages through the pipe using the parent and the child processes.

Algorithm

Step 1 − Create a pipe.

Step 2 − Create a child process.

Step 3 − Parent process writes to the pipe.

Step 4 − Child process retrieves the message from the pipe and writes it to the standard output.

Step 5 − Repeat step 3 and step 4 once again.

Source Code: pipewithprocesses.c

```
#include<stdio.h>
#include<unistd.h>

int main() {
   int pipefds[2];
   int returnstatus;
   int pid;
   char writemessages[2][20]={"Hi", "Hello"};
   char readmessage[20];
   returnstatus = pipe(pipefds);
   if (returnstatus == -1) {
      printf("Unable to create pipe\n");
      return 1;
   }
   pid = fork();

   // Child process
   if (pid == 0) {
      read(pipefds[0], readmessage, sizeof(readmessage));
      printf("Child Process - Reading from pipe – Message 1 is %s\n", readmessage);
      read(pipefds[0], readmessage, sizeof(readmessage));
      printf("Child Process - Reading from pipe – Message 2 is %s\n", readmessage);
   } else { //Parent process
      printf("Parent Process - Writing to pipe - Message 1 is %s\n", writemessages[0]);
      write(pipefds[1], writemessages[0], sizeof(writemessages[0]));
      printf("Parent Process - Writing to pipe - Message 2 is %s\n", writemessages[1]);
      write(pipefds[1], writemessages[1], sizeof(writemessages[1]));
   }
   return 0;
}
```

Two-way Communication Using Pipes

Pipe communication is viewed as only one-way communication i.e., either the parent process writes and the child process reads or vice-versa but not both. However, what if both the parent and the child needs to write and read from the pipes simultaneously, the solution is a two-way communication using pipes. Two pipes are required to establish two-way communication.

Following are the steps to achieve two-way communication −

Step 1 − Create two pipes. First one is for the parent to write and child to read, say as pipe1. Second one is for the child to write and parent to read, say as pipe2.

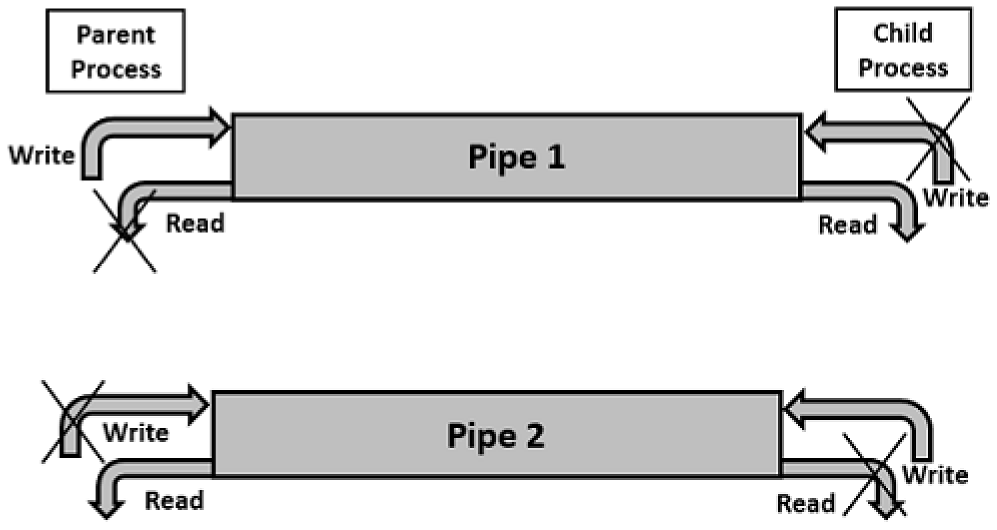
Step 2 − Create a child process.

Step 3 − Close unwanted ends as only one end is needed for each communication.

Step 4 − Close unwanted ends in the parent process, read end of pipe1 and write end of pipe2.

Step 5 − Close the unwanted ends in the child process, write end of pipe1 and read end of pipe2.

Step 6 − Perform the communication as required.



Sample Programs

Sample program 1 − Achieving two-way communication using pipes.

Algorithm

Step 1 − Create pipe1 for the parent process to write and the child process to read.

Step 2 − Create pipe2 for the child process to write and the parent process to read.

Step 3 − Close the unwanted ends of the pipe from the parent and child side.

Step 4 − Parent process to write a message and child process to read and display on the screen.

Step 5 − Child process to write a message and parent process to read and display on the screen.

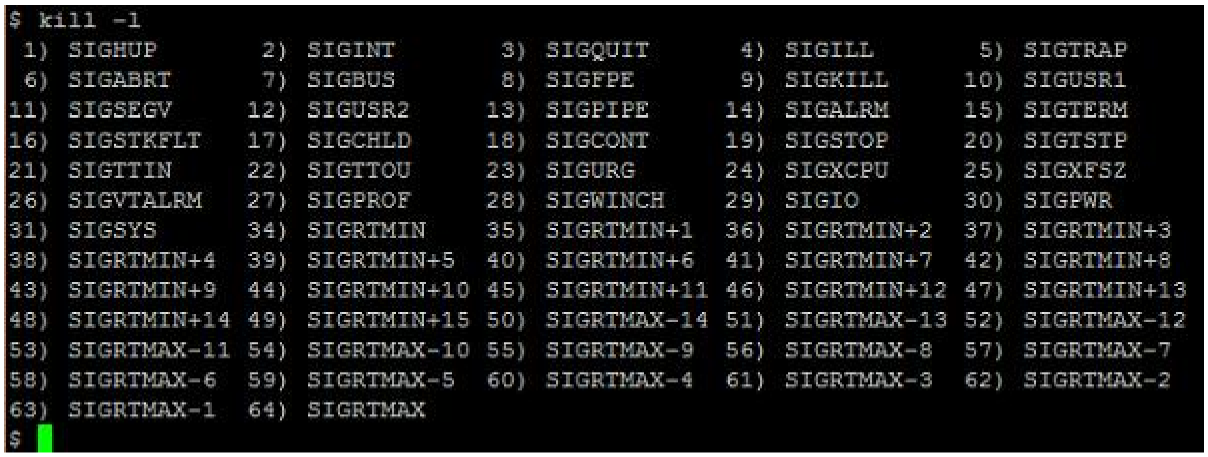
Source Code: twowayspipe.c

```
#include<stdio.h>
#include<unistd.h>

int main() {
   int pipefds1[2], pipefds2[2];
   int returnstatus1, returnstatus2;
   int pid;
   char pipe1writemessage[20] = "Hi";
   char pipe2writemessage[20] = "Hello";
   char readmessage[20];
   returnstatus1 = pipe(pipefds1);

   if (returnstatus1 == -1) {
      printf("Unable to create pipe 1 \n");
      return 1;
   }
   returnstatus2 = pipe(pipefds2);

   if (returnstatus2 == -1) {
      printf("Unable to create pipe 2 \n");
      return 1;
   }
   pid = fork();

   if (pid != 0) // Parent process {
      close(pipefds1[0]); // Close the unwanted pipe1 read side
      close(pipefds2[1]); // Close the unwanted pipe2 write side
      printf("In Parent: Writing to pipe 1 – Message is %s\n", pipe1writemessage);
      write(pipefds1[1], pipe1writemessage, sizeof(pipe1writemessage));
      read(pipefds2[0], readmessage, sizeof(readmessage));
      printf("In Parent: Reading from pipe 2 – Message is %s\n", readmessage);
   } else { //child process
      close(pipefds1[1]); // Close the unwanted pipe1 write side
      close(pipefds2[0]); // Close the unwanted pipe2 read side
      read(pipefds1[0], readmessage, sizeof(readmessage));
      printf("In Child: Reading from pipe 1 – Message is %s\n", readmessage);
      printf("In Child: Writing to pipe 2 – Message is %s\n", pipe2writemessage);
      write(pipefds2[1], pipe2writemessage, sizeof(pipe2writemessage));
   }
   return 0;
}
```

Signal

signal is a notification to a process indicating the occurrence of an event. Signal is also called **software interrupt** and is not predictable to know its occurrence, hence it is also called an **asynchronous event**.

Signal can be specified with a number or a name, usually signal names start with SIG. The available signals can be checked with the command kill –l (l for Listing signal names), which is as follows −



```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
$
```

Whenever a signal is raised (either programmatically or system generated signal), a default action is performed. What if you don't want to perform the default action but wish to perform your own actions on receiving the signal? Is this possible for all the signals? Yes, it is possible to handle the signal but not for all the signals. What if you want to ignore the signals, is this possible? Yes, it is possible to ignore the signal. Ignoring the signal implies neither performing the default action nor handling the signal. It is possible to ignore or handle almost all the signals. The signals which can't be either ignored or handled/caught are SIGSTOP and SIGKILL.

In summary, the actions performed for the signals are as follows −

- Default Action
- Handle the signal
- Ignore the signal

As discussed the signal can be handled altering the execution of default action. Signal handling can be done in either of the two ways i.e., through system calls, signal() and sigaction().

```
#include <signal.h>

typedef void (*sighandler_t) (int);
```

```
sighandler_t signal(int signum, sighandler_t handler);
```

The system call signal() would call the registered handler upon generation of signal as mentioned in signum. The handler can be either one of the SIG_IGN (Ignoring the Signal), SIG_DFL (Setting signal back to default mechanism) or user-defined signal handler or function address.

This system call on success returns the address of a function that takes an integer argument and has no return value. This call returns SIG_ERR in case of error.

Though with signal() the respective signal handler as registered by the user can be called, fine tuning such as masking the signals that should be blocked, modifying the behavior of a signal, and other functionalities are not possible. This are possible using sigaction() system call.

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)
```

This system call is used to either examine or change a signal action. If the act is not null, the new action for signal signum is installed from the act. If oldact is not null, the previous action is saved in oldact.

The sigaction structure contains the following fields −

Field 1 − Handler mentioned either in sa_handler or sa_sigaction.

```
void (*sa_handler)(int);
void (*sa_sigaction)(int, siginfo_t *, void *);
```

The handler for sa_handler specifies the action to be performed based on the signum and with SIG_DFL indicating default action or SIG_IGN to ignore the signal or pointer to a signal handling function.

The handler for sa_sigaction specifies the signal number as the first argument, pointer to siginfo_t structure as the second argument and pointer to user context (check getcontext() or setcontext() for further details) as the third argument.

The structure siginfo_t contains signal information such as the signal number to be delivered, signal value, process id, real user id of sending process, etc.

Field 2 − Set of signals to be blocked.

```
int sa_mask;
```

This variable specifies the mask of signals that should be blocked during the execution of signal handler.

Field 3 − Special flags.

```
int sa_flags;
```

This field specifies a set of flags which modify the behavior of the signal.

Field 4 − Restore handler.

void (*sa_restorer) (void);

This system call returns 0 on success and -1 in case of failure.

Let us consider a few sample programs.

First, let us start with a sample program, which generates exception. In this program, we are trying to perform divide by zero operation, which makes the system generate an exception.

```
/* signal_fpe.c */
#include<stdio.h>

int main() {
   int result;
   int v1, v2;
   v1 = 121;
   v2 = 0;
   result = v1/v2;
   printf("Result of Divide by Zero is %d\n", result);
   return 0;
}
```

Compilation and Execution Steps

Floating point exception (core dumped)

Thus, when we are trying to perform an arithmetic operation, the system has generated a floating point exception with core dump, which is the default action of the signal.

Now, let us modify the code to handle this particular signal using signal() system call.

```
/* signal_fpe_handler.c */
#include<stdio.h>
#include<signal.h>
#include<stdlib.h>

void handler_dividebyzero(int signum);

int main() {
   int result;
   int v1, v2;
   void (*sigHandlerReturn)(int);
   sigHandlerReturn = signal(SIGFPE, handler_dividebyzero);
   if (sigHandlerReturn == SIG_ERR) {
      perror("Signal Error: ");
      return 1;
   }
   v1 = 121;
   v2 = 0;
   result = v1/v2;
   printf("Result of Divide by Zero is %d\n", result);
   return 0;
}

void handler_dividebyzero(int signum) {
   if (signum == SIGFPE) {
      printf("Received SIGFPE, Divide by Zero Exception\n");
      exit (0);
   }
   else
      printf("Received %d Signal\n", signum);
      return;
}
```

As discussed, signals are generated by the system (upon performing certain operations such as divide by zero, etc.) or the user can also generate the signal programmatically. If you want to generate signal programmatically, use the library function raise().

Now, let us take another program to demonstrate handling and ignoring the signal.

Assume that we have raised a signal using raise(), what happens then? After raising the signal, the execution of the current process is stopped. Then what happens to the stopped process? There can be two scenarios – First, continue the execution whenever required. Second, terminate (with kill command) the process.

To continue the execution of the stopped process, send SIGCONT to that particular process. You can also issue fg (foreground) or bg (background) commands to continue the execution. Here, the commands would only re-start the execution of the last process. If more than one process is stopped, then only the last process is resumed. If you want to resume the previously stopped process, then resume the jobs (using fg/bg) along with job number.

The following program is used to raise the signal SIGSTOP using raise() function. Signal SIGSTOP can also be generated by the user press of CTRL + Z (Control + Z) key. After issuing this signal, the program will stop executing. Send the signal (SIGCONT) to continue the execution.

In the following example, we are resuming the stopped process with command fg.

```
/* signal_raising.c */
#include<stdio.h>
#include<signal.h>
#include<stdlib.h>

int main() {
   printf("Testing SIGSTOP\n");
   raise(SIGSTOP);
   return 0;
}
```

Now, enhance the previous program to continue the execution of the stopped process by issuing SIGCONT from another terminal.

```
/* signal_stop_continue.c */
#include<stdio.h>
#include<signal.h>
#include <sys/types.h>
#include <unistd.h>

void handler_sigtstp(int signum);

int main() {
   pid_t pid;
   printf("Testing SIGSTOP\n");
   pid = getpid();
   printf("Open Another Terminal and issue following command\n");
   printf("kill -SIGCONT %d or kill -CONT %d or kill -18 %d\n", pid, pid, pid);
   raise(SIGSTOP);
   printf("Received signal SIGCONT\n");
   return 0;
}
```

Compilation and Execution Steps

```
Testing SIGSTOP
Open Another Terminal and issue following command
kill -SIGCONT 30379 or kill -CONT 30379 or kill -18 30379
[1]+ Stopped ./a.out

Received signal SIGCONT
[1]+ Done ./a.out
```

In another terminal

```
kill -SIGCONT 30379
```

So far, we have seen the program which handles the signal generated by the system. Now, let us see the signal generated through program (using raise() function or through kill command). This program generates signal SIGTSTP (terminal stop), whose default action is to stop the execution. However, since we are handling the signal now instead of default action, it will come to the defined handler. In this case, we are just printing the message and exiting.

```
/* signal_raising_handling.c */
#include<stdio.h>
#include<signal.h>
#include<stdlib.h>
```

```
void handler_sigtstp(int signum);

int main() {
   void (*sigHandlerReturn)(int);
   sigHandlerReturn = signal(SIGTSTP, handler_sigtstp);
   if (sigHandlerReturn == SIG_ERR) {
      perror("Signal Error: ");
      return 1;
   }
   printf("Testing SIGTSTP\n");
   raise(SIGTSTP);
   return 0;
}

void handler_sigtstp(int signum) {
   if (signum == SIGTSTP) {
      printf("Received SIGTSTP\n");
      exit(0);
   }
   else
      printf("Received %d Signal\n", signum);
      return;
}
```

Compilation and Execution Steps

```
Testing SIGTSTP
Received SIGTSTP
```

We have seen the instances of performing default action or handling the signal. Now, it is time to ignore the signal. Here, in this sample program, we are registering the signal SIGTSTP to ignore through SIG_IGN and then we are raising the signal SIGTSTP (terminal stop). When the signal SIGTSTP is being generated that would be ignored.

```
/* signal_raising_ignoring.c */
#include<stdio.h>
#include<signal.h>
#include<stdlib.h>

void handler_sigtstp(int signum);

int main() {
   void (*sigHandlerReturn)(int);
   sigHandlerReturn = signal(SIGTSTP, SIG_IGN);
   if (sigHandlerReturn == SIG_ERR) {
      perror("Signal Error: ");
      return 1;
   }
   printf("Testing SIGTSTP\n");
   raise(SIGTSTP);
   printf("Signal SIGTSTP is ignored\n");
   return 0;
}
```

Compilation and Execution Steps

```
Testing SIGTSTP
Signal SIGTSTP is ignored
```

So far, we have observed that we have one signal handler to handle one signal. Can we have a single handler to handle multiple signals? The answer is Yes. Let us consider this with a program.

The following program does the following −

Step 1 − Registers a handler (handleSignals) to catch or handle signals SIGINT (CTRL + C) or SIGQUIT (CTRL + \)

Step 2 − If the user generates signal SIGQUIT (either through kill command or keyboard control with CTRL + \), the handler simply prints the message as return.

Step 3 − If the user generates signal SIGINT (either through kill command or keyboard control with CTRL + C) first time, then it modifies the signal to perform default action (with SIG_DFL) from next time.

Step 4 − If the user generates signal SIGINT second time, it performs a default action, which is the termination of program.

```
/* Filename: sigHandler.c */
#include<stdio.h>
#include<unistd.h>
#include<signal.h>

void handleSignals(int signum);

int main(void) {
   void (*sigHandlerInterrupt)(int);
   void (*sigHandlerQuit)(int);
   void (*sigHandlerReturn)(int);
   sigHandlerInterrupt = sigHandlerQuit = handleSignals;
   sigHandlerReturn = signal(SIGINT, sigHandlerInterrupt);
   if (sigHandlerReturn == SIG_ERR) {
      perror("signal error: ");
      return 1;
   }
   sigHandlerReturn = signal(SIGQUIT, sigHandlerQuit);

   if (sigHandlerReturn == SIG_ERR) {
      perror("signal error: ");
      return 1;
   }
   while (1) {
      printf("\nTo terminate this program, perform the following: \n");
      printf("1. Open another terminal\n");
      printf("2. Issue command: kill %d or issue CTRL+C 2 times (second time it
terminates)\n", getpid());
      sleep(10);
   }
   return 0;
}

void handleSignals(int signum) {
   switch(signum) {
      case SIGINT:
      printf("\nYou pressed CTRL+C \n");
      printf("Now reverting SIGINT signal to default action\n");
      signal(SIGINT, SIG_DFL);
      break;
      case SIGQUIT:
      printf("\nYou pressed CTRL+\\ \n");
      break;
      default:
      printf("\nReceived signal number %d\n", signum);
      break;
   }
   return;
}
```

Compilation and Execution Steps

```
To terminate this program, perform the following:
1. Open another terminal
2. Issue command: kill 74 or issue CTRL+C 2 times (second time it terminates)
^C
You pressed CTRL+C
Now reverting SIGINT signal to default action

To terminate this program, perform the following:
1. Open another terminal
2. Issue command: kill 74 or issue CTRL+C 2 times (second time it terminates)
^\You pressed CTRL+\
To terminate this program, perform the following:
1. Open another terminal
2. Issue command: kill 120
Terminated
```

Another Terminal

```
kill 71
```

Second Method

```
To terminate this program, perform the following:
1. Open another terminal
2. Issue command: kill 71 or issue CTRL+C 2 times (second time it terminates)
^C
You pressed CTRL+C
Now reverting SIGINT signal to default action

To terminate this program, perform the following:
1. Open another terminal
2. Issue command: kill 71 or issue CTRL+C 2 times (second time it terminates)
^C
```

We know that to handle a signal, we have two system calls i.e., either signal() or sigaction(). Till now we have seen with signal() system call, now it is time for sigaction() system call. Let us modify the above program to perform using sigaction() as follows −

```c
/* Filename: sigHandlerSigAction.c */
#include<stdio.h>
#include<unistd.h>
#include<signal.h>

void handleSignals(int signum);

int main(void) {
  void (*sigHandlerReturn)(int);
  struct sigaction mysigaction;
  mysigaction.sa_handler = handleSignals;
  sigemptyset(&mysigaction.sa_mask);
  mysigaction.sa_flags = 0;
  sigaction(SIGINT, &mysigaction, NULL);

  if (mysigaction.sa_handler == SIG_ERR) {
    perror("signal error: ");
    return 1;
  }
  mysigaction.sa_handler = handleSignals;
  sigemptyset(&mysigaction.sa_mask);
  mysigaction.sa_flags = 0;
  sigaction(SIGQUIT, &mysigaction, NULL);

  if (mysigaction.sa_handler == SIG_ERR) {
    perror("signal error: ");
```

```
      return 1;
   }
   while (-1) {
      printf("\nTo terminate this program, perform either of the following: \n");
      printf("1. Open another terminal and issue command: kill %d\n", getpid());
      printf("2. Issue CTRL+C 2 times (second time it terminates)\n");
      sleep(10);
   }
   return 0;
}

void handleSignals(int signum) {
   switch(signum) {
      case SIGINT:
      printf("\nYou have entered CTRL+C \n");
      printf("Now reverting SIGINT signal to perform default action\n");
      signal(SIGINT, SIG_DFL);
      break;
      case SIGQUIT:
      printf("\nYou have entered CTRL+\\ \n");
      break;
      default:
      printf("\nReceived signal number %d\n", signum);
      break;
   }
   return;
}
```

Let us see the compilation and execution process. In the execution process, let us see issue CTRL+C twice, remaining checks/ways (as above) you can try for this program as well.

Compilation and Execution Steps

```
To terminate this program, perform either of the following:
1. Open another terminal and issue command: kill 3199
2. Issue CTRL+C 2 times (second time it terminates)
^C
You have entered CTRL+C
Now reverting SIGINT signal to perform default action
To terminate this program, perform either of the following:
1. Open another terminal and issue command: kill 3199
2. Issue CTRL+C 2 times (second time it terminates)
```

Semaphores IPC mechanisms

The first question that comes to mind is, why do we need semaphores? A simple answer, to protect the critical/common region shared among multiple processes.

Let us assume, multiple processes are using the same region of code and if all want to access parallelly then the outcome is overlapped. Say, for example, multiple users are using one printer only (common/critical section), say 3 users, given 3 jobs at same time, if all the jobs start parallelly, then one user output is overlapped with another. So, we need to protect that using semaphores i.e., locking the critical section when one process is running and unlocking when it is done. This would be repeated for each user/process so that one job is not overlapped with another job.

Basically semaphores are classified into two types −

Binary Semaphores − Only two states 0 & 1, i.e., locked/unlocked or available/unavailable, Mutex implementation.

Counting Semaphores − Semaphores which allow arbitrary resource count are called counting semaphores.

Assume that we have 5 printers (to understand assume that 1 printer only accepts 1 job) and we got 3 jobs to print. Now 3 jobs would be given for 3 printers (1 each). Again 4 jobs came while this is in progress. Now, out of 2 printers available, 2 jobs have been scheduled and we are left with 2 more jobs, which would be completed only after one of the resource/printer is available. This kind of scheduling as per resource availability can be viewed as counting semaphores.

To perform synchronization using semaphores, following are the steps −

Step 1 − Create a semaphore or connect to an already existing semaphore (semget())

Step 2 − Perform operations on the semaphore i.e., allocate or release or wait for the resources (semop())

Step 3 − Perform control operations on the message queue (semctl())

Now, let us check this with the system calls we have.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg)
```

This system call creates or allocates a System V semaphore set. The following arguments need to be passed −

- The first argument, key, recognizes the message queue. The key can be either an arbitrary value or one that can be derived from the library function ftok().

- The second argument, nsems, specifies the number of semaphores. If binary then it is 1, implies need of 1 semaphore set, otherwise as per the required count of number of semaphore sets.
- The third argument, semflg, specifies the required semaphore flag/s such as IPC_CREAT (creating semaphore if it does not exist) or IPC_EXCL (used with IPC_CREAT to create semaphore and the call fails, if a semaphore already exists). Need to pass the permissions as well.

Note − Refer earlier sections for details on permissions.

This call would return valid semaphore identifier (used for further calls of semaphores) on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

Various errors with respect to this call are EACCESS (permission denied), EEXIST (queue already exists can't create), ENOENT (queue doesn't exist), ENOMEM (not enough memory to create the queue), ENOSPC (maximum sets limit exceeded), etc.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *semops, size_t nsemops)
```

This system call performs the operations on the System V semaphore sets viz., allocating resources, waiting for the resources or freeing the resources. Following arguments need to be passed −

- The first argument, semid, indicates semaphore set identifier created by semget().
- The second argument, semops, is the pointer to an array of operations to be performed on the semaphore set. The structure is as follows −

```
struct sembuf {
  unsigned short sem_num; /* Semaphore set num */
  short sem_op; /* Semaphore operation */
  short sem_flg; /* Operation flags, IPC_NOWAIT, SEM_UNDO */
};
```

Element, sem_op, in the above structure, indicates the operation that needs to be performed −

- If sem_op is –ve, allocate or obtain resources. Blocks the calling process until enough resources have been freed by other processes, so that this process can allocate.
- If sem_op is zero, the calling process waits or sleeps until semaphore value reaches 0.
- If sem_op is +ve, release resources.

For example −

struct sembuf sem_lock = { 0, -1, SEM_UNDO };

struct sembuf sem_unlock = {0, 1, SEM_UNDO };

- The third argument, nsemops, is the number of operations in that array.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, …)
```

This system call performs control operation for a System V semaphore. The following arguments need to be passed −

- The first argument, semid, is the identifier of the semaphore. This id is the semaphore identifier, which is the return value of semget() system call.
- The second argument, semnum, is the number of semaphore. The semaphores are numbered from 0.
- The third argument, cmd, is the command to perform the required control operation on the semaphore.
- The fourth argument, of type, union semun, depends on the cmd. For few cases, the fourth argument is not applicable.

Let us check the union semun −

```
union semun {
  int val; /* val for SETVAL */
  struct semid_ds *buf; /* Buffer for IPC_STAT and IPC_SET */
  unsigned short *array; /* Buffer for GETALL and SETALL */
  struct seminfo *__buf; /* Buffer for IPC_INFO and SEM_INFO*/
};
```

The semid_ds data structure which is defined in sys/sem.h is as follows −

```
struct semid_ds {
  struct ipc_perm sem_perm; /* Permissions */
  time_t sem_otime; /* Last semop time */
  time_t sem_ctime; /* Last change time */
  unsigned long sem_nsems; /* Number of semaphores in the set */
};
```

Note − Please refer manual pages for other data structures.

union semun arg; Valid values for cmd are −

- **IPC_STAT** − Copies the information of the current values of each member of struct semid_ds to the passed structure pointed by arg.buf. This command requires read permission to the semaphore.

- **IPC_SET** − Sets the user ID, group ID of the owner, permissions, etc. pointed to by the structure semid_ds.
- **IPC_RMID** − Removes the semaphores set.
- **IPC_INFO** − Returns the information about the semaphore limits and parameters in the structure semid_ds pointed by arg.__buf.
- **SEM_INFO** − Returns a seminfo structure containing information about the consumed system resources by the semaphore.

This call would return value (non-negative value) depending upon the passed command. Upon success, IPC_INFO and SEM_INFO or SEM_STAT returns the index or identifier of the highest used entry as per Semaphore or the value of semncnt for GETNCNT or the value of sempid for GETPID or the value of semval for GETVAL 0 for other operations on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

Before looking at the code, let us understand its implementation −

- Create two processes say, child and parent.
- Create shared memory mainly needed to store the counter and other flags to indicate end of read/write process into the shared memory.
- The counter is incremented by count by both parent and child processes. The count is either passed as a command line argument or taken as default (if not passed as command line argument or the value is less than 10000). Called with certain sleep time to ensure both parent and child accesses the shared memory at the same time i.e., in parallel.
- Since, the counter is incremented in steps of 1 by both parent and child, the final value should be double the counter. Since, both parent and child processes performing the operations at same time, the counter is not incremented as required. Hence, we need to ensure the completeness of one process completion followed by other process.
- All the above implementations are performed in the file shm_write_cntr.c
- Check if the counter value is implemented in file shm_read_cntr.c
- To ensure completion, the semaphore program is implemented in file shm_write_cntr_with_sem.c. Remove the semaphore after completion of the entire process (after read is done from other program)
- Since, we have separate files to read the value of counter in the shared memory and don’t have any effect from writing, the reading program remains the same (shm_read_cntr.c)
- It is always better to execute the writing program in one terminal and reading program from another terminal. Since, the program completes execution only after the writing and reading process is complete, it is ok to run the program after

completely executing the write program. The write program would wait until the read program is run and only finishes after it is done.

Remote procedure call (RPC)

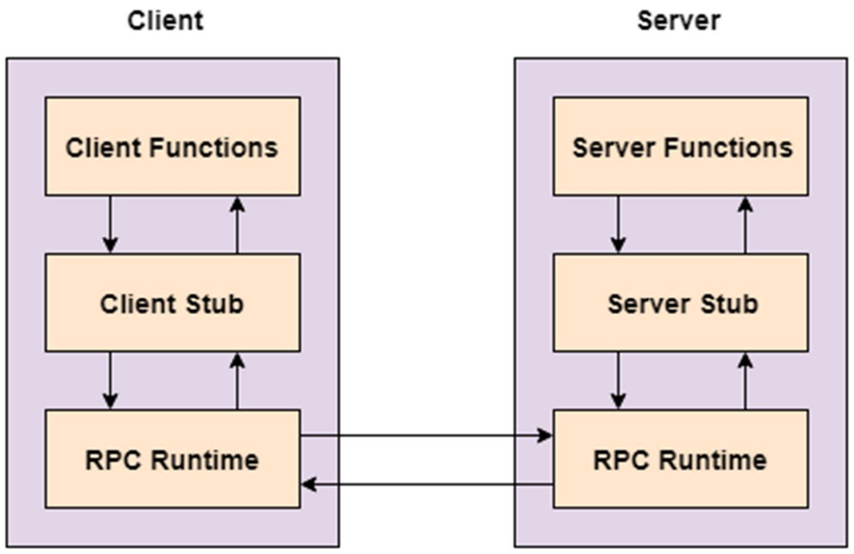
A remote procedure call is an interprocess communication technique that is used for client-server based applications. It is also known as a subroutine call or a function call.

A client has a request message that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client. The client is blocked while the server is processing the call and only resumed execution after the server is finished.

The sequence of events in a remote procedure call are given as follows −

- The client stub is called by the client.
- The client stub makes a system call to send the message to the server and puts the parameters in the message.
- The message is sent from the client to the server by the client's operating system.
- The message is passed to the server stub by the server operating system.
- The parameters are removed from the message by the server stub.
- Then, the server procedure is called by the server stub.

A diagram that demonstrates this is as follows −



Advantages of Remote Procedure Call

Some of the advantages of RPC are as follows −

- Remote procedure calls support process oriented and thread oriented models.
- The internal message passing mechanism of RPC is hidden from the user.
- The effort to re-write and re-develop the code is minimum in remote procedure calls.
- Remote procedure calls can be used in distributed environment as well as the local environment.
- Many of the protocol layers are omitted by RPC to improve performance.

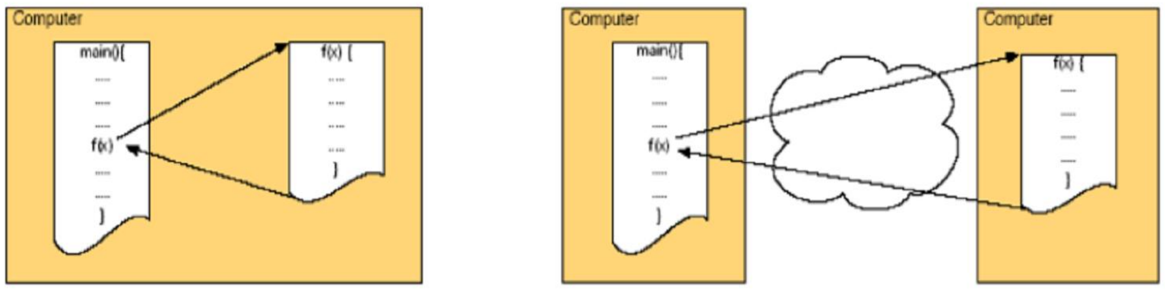
Disadvantages of Remote Procedure Call

Some of the disadvantages of RPC are as follows −

- The remote procedure call is a concept that can be implemented in different ways. It is not a standard.
- There is no flexibility in RPC for hardware architecture. It is only interaction based.
- There is an increase in costs because of remote procedure call.

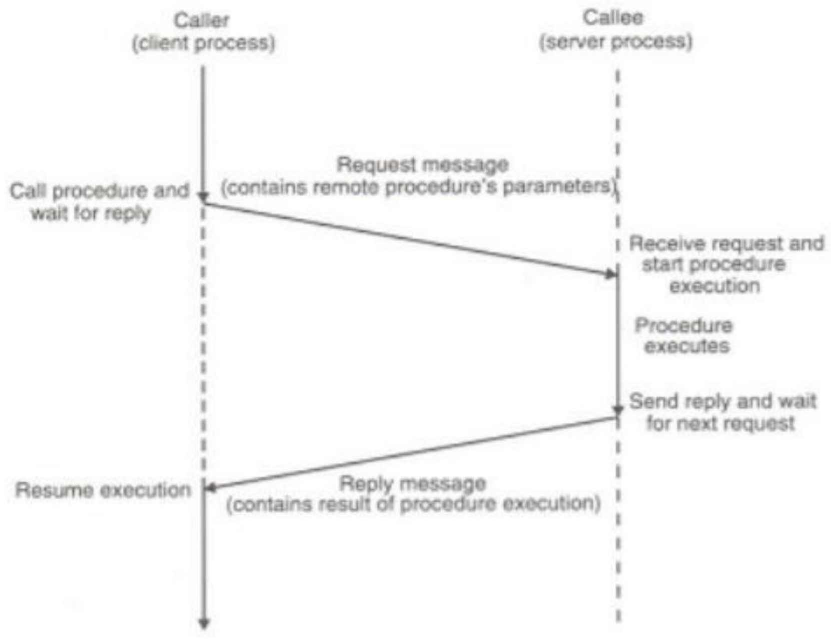
<u>RPC Model</u>

- ✓ Similar to "Procedure Call" model except now we invoke procedures on a remote system.
- ✓ Local Procedure Call – "Caller and Callee are within a single process on a given host"
- ✓ Intent of RPC is to make it appear to the programmer that a local procedure call is taking place



Local Procedure Call　　　　Remote Procedure Call

Typical workflow of RPC


Caller
(client process)
Callee
(server process)
Request message
(contains remote procedure's parameters)
Call procedure and wait for reply
Receive request and start procedure execution
Procedure executes
Send reply and wait for next request
Resume execution
Reply message
(contains result of procedure execution)

1. Caller issue remote procedure request
2. Callee (remote processing system) starts procedure execution and sends result process back to caller

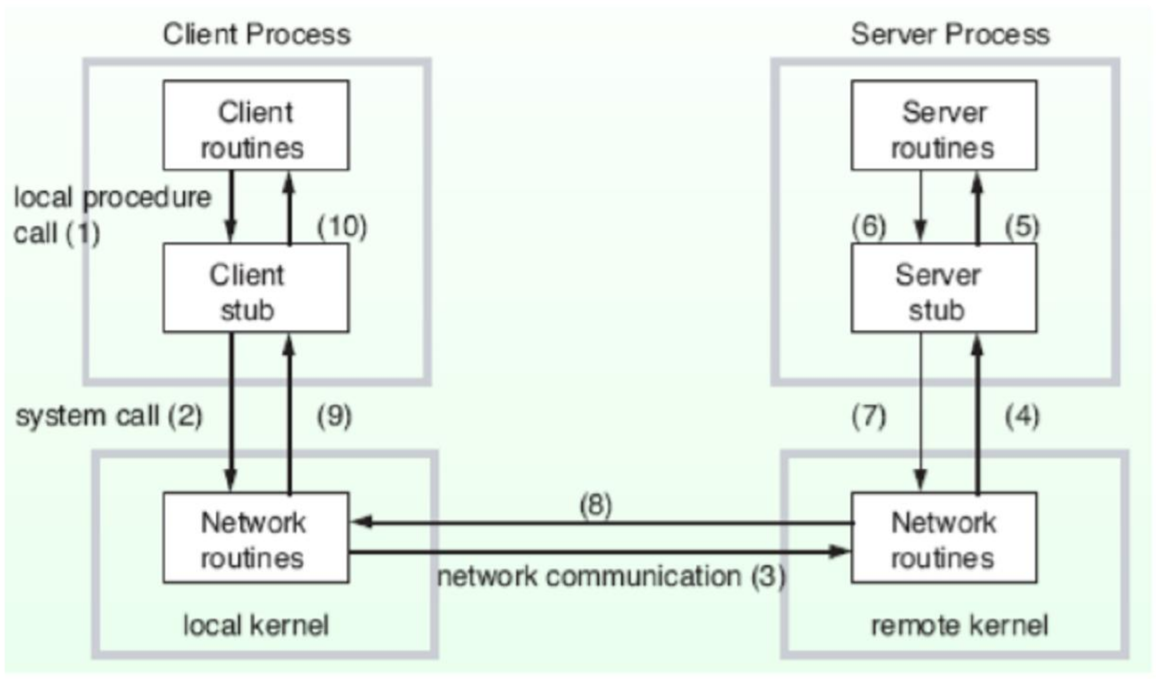
Implementing RPC Mechanism

- ✓ Similar to networking stacks, we have layers as 'stubs' in our RPC protocol.
- ✓ Stubs provide a perfectly normal(local) procedure call abstraction
- ✓ Implementation involves five elements:
  - ➢ Client
  - ➢ client stub
  - ➢ RPC Runtime
  - ➢ server stub
  - ➢ server

Stubs

Client & sever stubs are generated from interface definition of server routines by development tools, similar to class definition in C++ & Java.

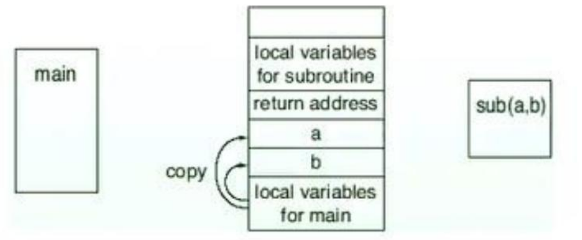
Diagram example



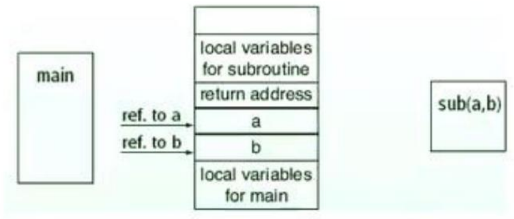
Parameter Passing Mechanisms

When a procedure is called, parameters are passed to the procedure as arguments. There are 3 methods of passing parameters.

1. Call-by-value – supplying an actual value
   The calling procedure may modify value, but modifications do not affect the original value at the calling side

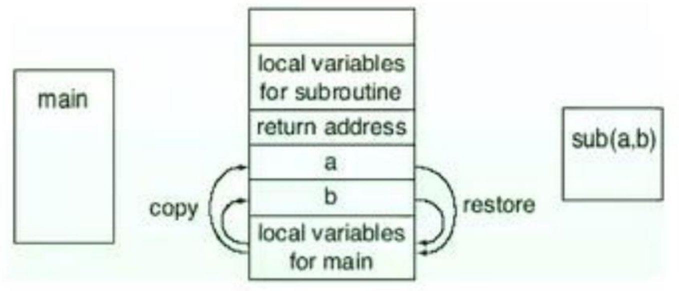


2. Call-by-reference – supplying the memory address of a variable, enabling the calling procedure to manipulate the original values at the calling side



3. Call-by-copy/restore

- ✓ Values of arguments are copied to the stack and passed to the calling procedure.
- ✓ When the processing of procedure completes, the values are copied back to the original values at the calling side.

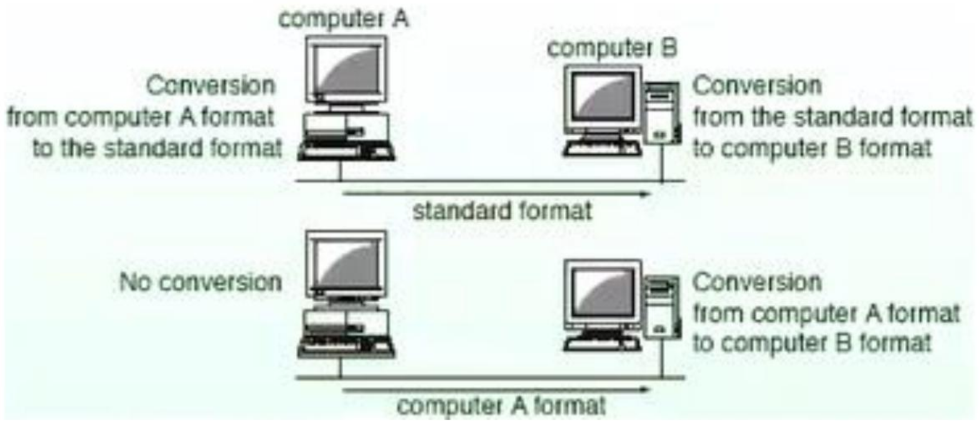


Parameter Passing in RPC

- ✓ We can implement all 3 mechanisms shown above.
- ✓ Usually call-by-value and call-by-copy/restore are used
- ✓ Call-by-reference is difficult to implement. all data which may be referenced mu be copied to the remote host and the reference to the copied data is used.

Question: Do we need to convert the values of parameter arguments into a standa format to transmit over the network?

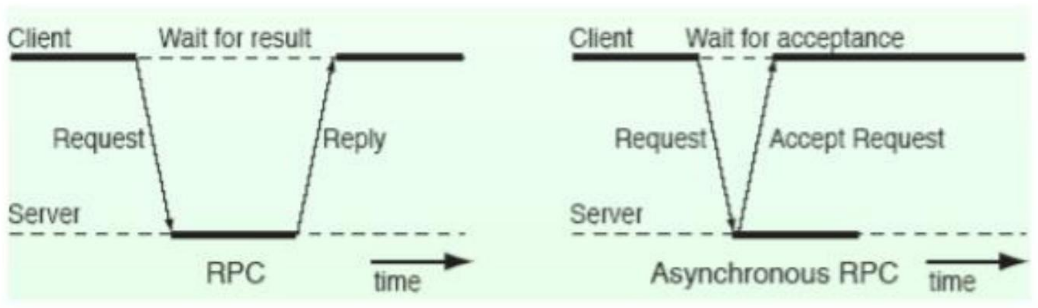
- ✓ Yes, many machines use different character codes, e.g. IBM mainframes use EBCDIC, while PCs use ASCII
- ✓ If a standard format is not used, two message conversions are necessary
- ✓ If format info is attached to a message, only one conversion at receiver will suffice, however, receiver must be able to handle many different formats.


computer A
computer B
Conversion from computer A format to the standard format
Conversion from the standard format to computer B format
standard format
No conversion
Conversion from computer A format to computer B format
computer A format

Asynchronous RPC

- ✓ For synchronous RPC, when a client requests a remote procedure, the client wait until a reply comes back
- ✓ If no result is returned, there will be unnecessary wait time overhead
- ✓ In asynchronous RPC, the server immediately sends accept message when it receives a request

Synchronous vs Asynchronous RPC Diagram



Interprocess Communications in windows OS

The Windows operating system provides mechanisms for facilitating communications and data sharing between applications. Collectively, the activities enabled by these mechanisms are called *interprocess communications* (IPC). Some forms of IPC facilitate the division of labor among several specialized processes. Other forms of IPC facilitate the division of labor among computers on a network.

Typically, applications can use IPC categorized as clients or servers. A *client* is an application or a process that requests a service from some other application or process. A *server* is an application or a process that responds to a client request. Many applications act as both a client and a server, depending on the situation. For example, a word processing application might act as a client in requesting a summary table of manufacturing costs from a spreadsheet application acting as a server. The spreadsheet application, in turn, might act as a client in requesting the latest inventory levels from an automated inventory control application.

After you decide that your application would benefit from IPC, you must decide which of the available IPC methods to use. It is likely that an application will use several IPC mechanisms. The answers to these questions determine whether an application can benefit by using one or more IPC mechanisms.

- Should the application be able to communicate with other applications running on other computers on a network, or is it sufficient for the application to communicate only with applications on the local computer?
- Should the application be able to communicate with applications running on other computers that may be running under different operating systems (such as 16-bit Windows or UNIX)?
- Should the user of the application have to choose the other applications with which the application communicates, or can the application implicitly find its cooperating partners?
- Should the application communicate with many different applications in a general way, such as allowing cut-and-paste operations with any other application, or should its communications requirements be limited to a restricted set of interactions with specific other applications?
- Is performance a critical aspect of the application? All IPC mechanisms include some amount of overhead.
- Should the application be a GUI application or a console application? Some IPC mechanisms require a GUI application.

The following IPC mechanisms are supported by Windows:

- Clipboard
- COM
- Data Copy
- DDE
- File Mapping
- Mailslots
- Pipes
- RPC
- Windows Sockets

Using the Clipboard for IPC

The clipboard acts as a central depository for data sharing among applications. When a user performs a cut or copy operation in an application, the application puts the selected data on the clipboard in one or more standard or application-defined formats. Any other application can then retrieve the data from the clipboard, choosing from the available formats that it understands. The clipboard is a very loosely coupled exchange medium, where applications need only agree on the data format. The applications can reside on the same computer or on different computers on a network.

Key Point: All applications should support the clipboard for those data formats that they understand. For example, a text editor or word processor should at least be able to produce and accept clipboard data in pure text format. For more information, see Clipboard.

Using COM for IPC

Applications that use OLE manage *compound documents*—that is, documents made up of data from a variety of different applications. OLE provides services that make it easy for applications to call on other applications for data editing. For example, a word processor that uses OLE could embed a graph from a spreadsheet. The user could start the spreadsheet automatically from within the word processor by choosing the embedded chart for editing. OLE takes care of starting the spreadsheet and presenting the graph for editing. When the user quit the spreadsheet, the graph would be updated in the original word processor document. The spreadsheet appears to be an extension of the word processor.

The foundation of OLE is the Component Object Model (COM). A software component that uses COM can communicate with a wide variety of other components, even those that have not yet been written. The components interact as objects and clients. Distributed COM extends the COM programming model so that it works across a network.

Key Point: OLE supports compound documents and enables an application to include embedded or linked data that, when chosen, automatically starts another application for data editing. This enables the application to be extended by any other application that uses OLE. COM objects provide access to an object's data through one or more sets of related functions, known as *interfaces*. For more information, see COM and ActiveX Object Services.

Using Data Copy for IPC

Data copy enables an application to send information to another application using the **WM_COPYDATA** message. This method requires cooperation between the sending application and the receiving application. The receiving application must know the format of the information and be able to identify the sender. The sending application cannot modify the memory referenced by any pointers.

Key Point: Data copy can be used to quickly send information to another application using Windows messaging. For more information, see Data Copy.

Using DDE for IPC

DDE is a protocol that enables applications to exchange data in a variety of formats. Applications can use DDE for one-time data exchanges or for ongoing exchanges in which the applications update one another as new data becomes available.

The data formats used by DDE are the same as those used by the clipboard. DDE can be thought of as an extension of the clipboard mechanism. The clipboard is almost always used for a one-time response to a user command, such as choosing the Paste command from a menu. DDE is also usually initiated by a user command, but it often continues to function without further user interaction. You can also define custom DDE data formats for special-purpose IPC between applications with more tightly coupled communications requirements.

DDE exchanges can occur between applications running on the same computer or on different computers on a network.

Key Point: DDE is not as efficient as newer technologies. However, you can still use DDE if other IPC mechanisms are not suitable or if you must interface with an existing application that only supports DDE. For more information, see Dynamic Data Exchange and Dynamic Data Exchange Management Library.

Using a File Mapping for IPC

File mapping enables a process to treat the contents of a file as if they were a block of memory in the process's address space. The process can use simple pointer operations to examine and modify the contents of the file. When two or more processes access the same file mapping, each process receives a pointer to memory in its own address space that it can use to read or modify the contents of the file. The processes must use a synchronization object, such as a semaphore, to prevent data corruption in a multitasking environment.

You can use a special case of file mapping to provide *named shared memory* between processes. If you specify the system swapping file when creating a file-mapping object, the file-mapping object is treated as a shared memory block. Other processes can access the same block of memory by opening the same file-mapping object.

File mapping is quite efficient and also provides operating-system–supported security attributes that can help prevent unauthorized data corruption. File mapping can be used only between processes on a local computer; it cannot be used over a network.

Key Point: File mapping is an efficient way for two or more processes on the same computer to share data, but you must provide synchronization between the processes. For more information, see File Mapping and Synchronization.

Using a Mailslot for IPC

Mailslots provide one-way communication. Any process that creates a mailslot is a *mailslot server*. Other processes, called *mailslot clients*, send messages to the mailslot server by writing a message to its mailslot. Incoming messages are always appended to the mailslot. The mailslot saves the messages until the mailslot server has read them. A process can be both a mailslot server and a mailslot client, so two-way communication is possible using multiple mailslots.

A mailslot client can send a message to a mailslot on its local computer, to a mailslot on another computer, or to all mailslots with the same name on all computers in a specified network domain. Messages broadcast to all mailslots on a domain can be no longer than 400 bytes, whereas messages sent to a single mailslot are limited only by the maximum message size specified by the mailslot server when it created the mailslot.

Key Point: Mailslots offer an easy way for applications to send and receive short messages. They also provide the ability to broadcast messages across all computers in a network domain. For more information, see Mailslots.

Using Pipes for IPC

There are two types of pipes for two-way communication: anonymous pipes and named pipes. *Anonymous pipes* enable related processes to transfer information to each other. Typically, an anonymous pipe is used for redirecting the standard input or output of a child process so that it can exchange data with its parent process. To exchange data in both directions (duplex operation), you must create two anonymous pipes. The parent process writes data to one pipe using its write handle, while the child process reads the data from that pipe using its read handle. Similarly, the child process writes data to the other pipe and the parent process reads from it. Anonymous pipes cannot be used over a network, nor can they be used between unrelated processes.

Named pipes are used to transfer data between processes that are not related processes and between processes on different computers. Typically, a named-pipe server process creates a named pipe with a well-known name or a name that is to be communicated to its clients. A named-pipe client process that knows the name of the pipe can open its other end, subject to access restrictions specified by named-pipe server process. After both the server and client have connected to the pipe, they can exchange data by performing read and write operations on the pipe.

Key Point: Anonymous pipes provide an efficient way to redirect standard input or output to child processes on the same computer. Named pipes provide a simple programming interface for transferring data between two processes, whether they reside on the same computer or over a network. For more information, see Pipes.

Using RPC for IPC

RPC enables applications to call functions remotely. Therefore, RPC makes IPC as easy as calling a function. RPC operates between processes on a single computer or on different computers on a network.

The RPC provided by Windows is compliant with the Open Software Foundation (OSF) Distributed Computing Environment (DCE). This means that applications that use RPC are able to communicate with applications running with other operating systems that support DCE. RPC automatically supports data conversion to account for different hardware architectures and for byte-ordering between dissimilar environments.

RPC clients and servers are tightly coupled but still maintain high performance. The system makes extensive use of RPC to facilitate a client/server relationship between different parts of the operating system.

Key Point: RPC is a function-level interface, with support for automatic data conversion and for communications with other operating systems. Using RPC, you can create high-performance, tightly coupled distributed applications. For more information, see Microsoft RPC Components.

Using Windows Sockets for IPC

Windows Sockets is a protocol-independent interface. It takes advantage of the communication capabilities of the underlying protocols. In Windows Sockets 2, a socket handle can optionally be used as a file handle with the standard file I/O functions.

Windows Sockets are based on the sockets first popularized by Berkeley Software Distribution (BSD). An application that uses Windows Sockets can communicate with other socket implementation on other types of systems. However, not all transport service providers support all available options.

Key Point: Windows Sockets is a protocol-independent interface capable of supporting current and emerging networking capabilities. For more information, see Windows Sockets 2.

Memory management

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

This tutorial will teach you basic concepts related to Memory Management.

Process Address Space

The process address space is the set of logical addresses that a process references in its code. For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, 2^31 possible numbers, for a total theoretical size of 2 gigabytes.

The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated −

S.N.	Memory Addresses & Description
1	**Symbolic addresses** The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.
2	**Relative addresses** At the time of compilation, a compiler converts symbolic addresses into relative addresses.
3	**Physical addresses** The loader generates these addresses at the time when a program is loaded into main memory.

Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

The set of all logical addresses generated by a program is referred to as a **logical address space**. The set of all physical addresses corresponding to these logical addresses is referred to as a **physical address space.**

The runtime mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process, which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
- The user program deals with virtual addresses; it never sees the real physical addresses.

Static vs Dynamic Loading

The choice between Static or Dynamic Loading is to be made at the time of computer program being developed. If you have to load your program statically, then at the time of compilation, the complete programs will be compiled and linked without leaving any external program or module dependency. The linker combines the object program with other necessary object modules into an absolute program, which also includes logical addresses.

If you are writing a Dynamically loaded program, then your compiler will compile the program and for all the modules which you want to include dynamically, only references will be provided and rest of the work will be done at the time of execution.

At the time of loading, with **static loading**, the absolute program (and data) is loaded into memory in order for execution to start.

If you are using **dynamic loading**, dynamic routines of the library are stored on a disk in relocatable form and are loaded into memory only when they are needed by the program.

Static vs Dynamic Linking

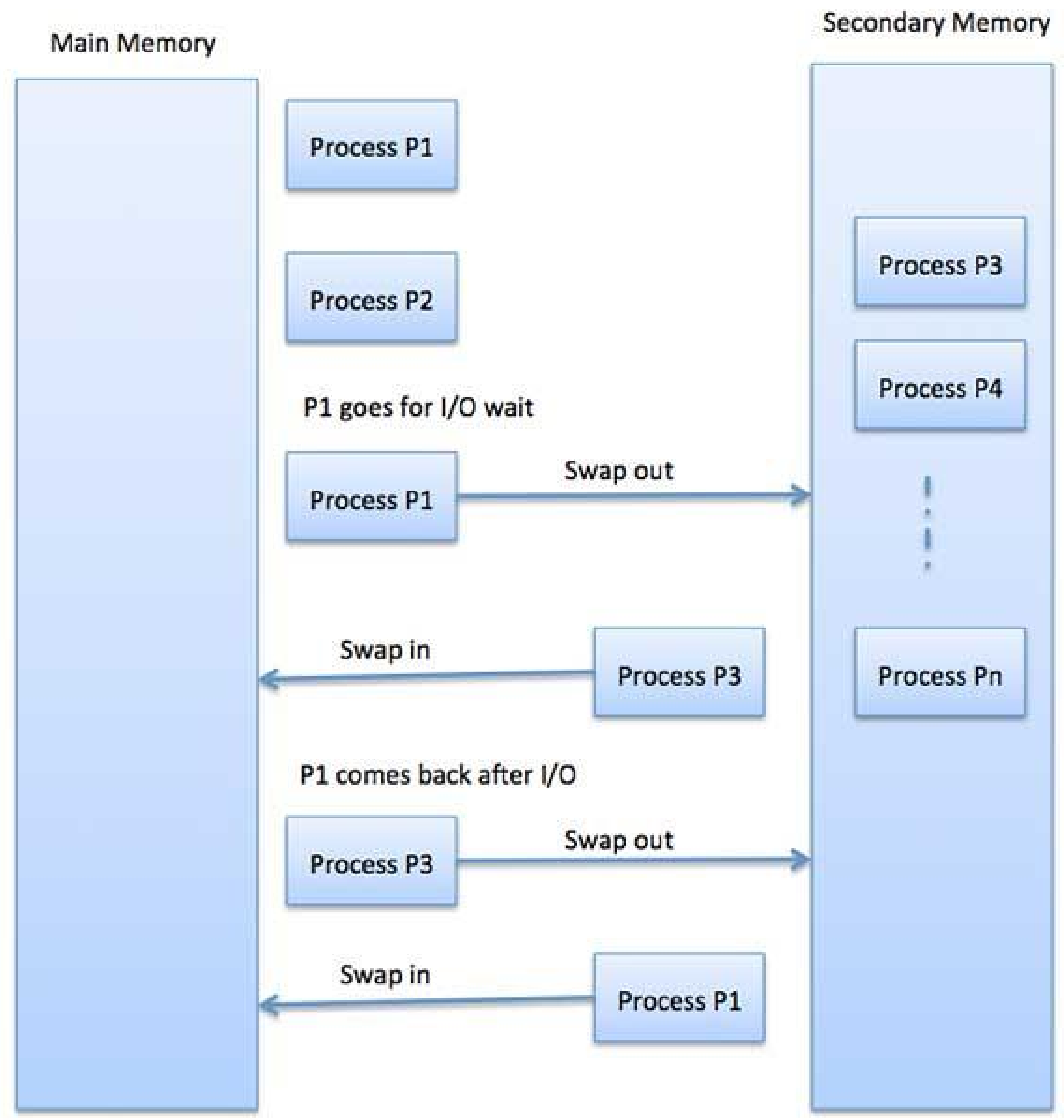
As explained above, when static linking is used, the linker combines all other modules needed by a program into a single executable program to avoid any runtime dependency.

When dynamic linking is used, it is not required to link the actual module or library with the program, rather a reference to the dynamic module is provided at the time of compilation and linking. Dynamic Link Libraries (DLL) in Windows and Shared Objects in Unix are good examples of dynamic libraries.

Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.

Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason **Swapping is also known as a technique for memory compaction**.



The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes to regain main memory.

Let us assume that the user process is of size 2048KB and on a standard hard disk where swapping will take place has a data transfer rate around 1 MB per second. The actual transfer of the 1000K process to or from memory will take

2048KB / 1024KB per second
= 2 seconds
= 2000 milliseconds

Now considering in and out time, it will take complete 4000 milliseconds plus other overhead where the process competes to regain main memory.

Memory Allocation

Main memory usually has two partitions −

- **Low Memory** − Operating system resides in this memory.
- **High Memory** − User processes are held in high memory.

Operating system uses the following memory allocation mechanism.

S.N.	Memory Allocation & Description
1	**Single-partition allocation** In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.
2	**Multiple-partition allocation** In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

Fragmentation

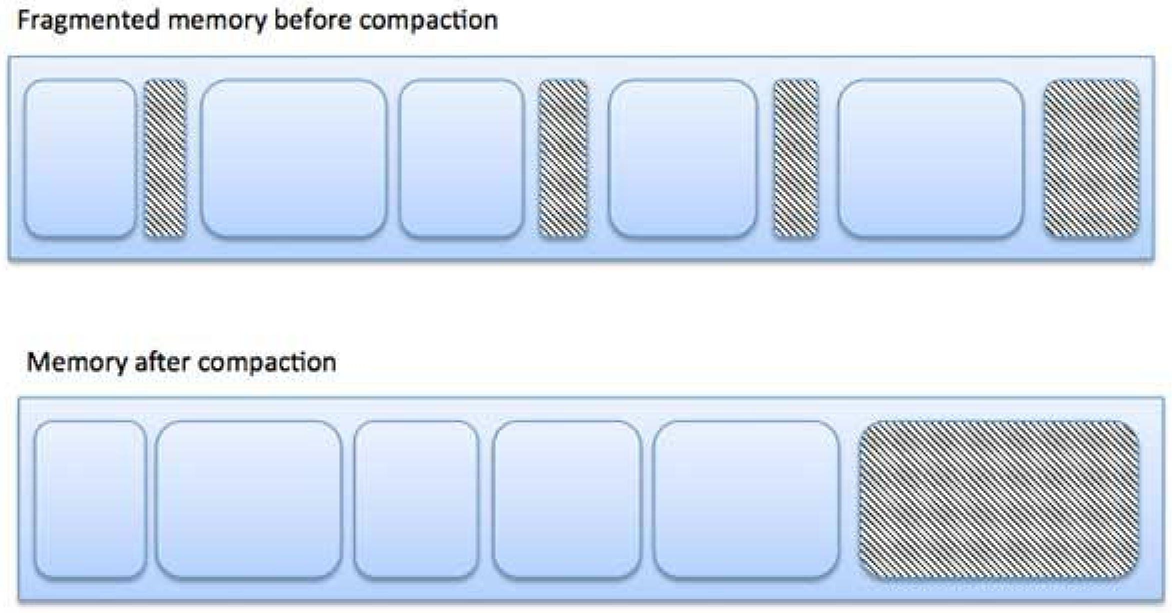
As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types −

S.N.	Fragmentation & Description
1	**External fragmentation** Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.

2	**Internal fragmentation** Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory −



External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.

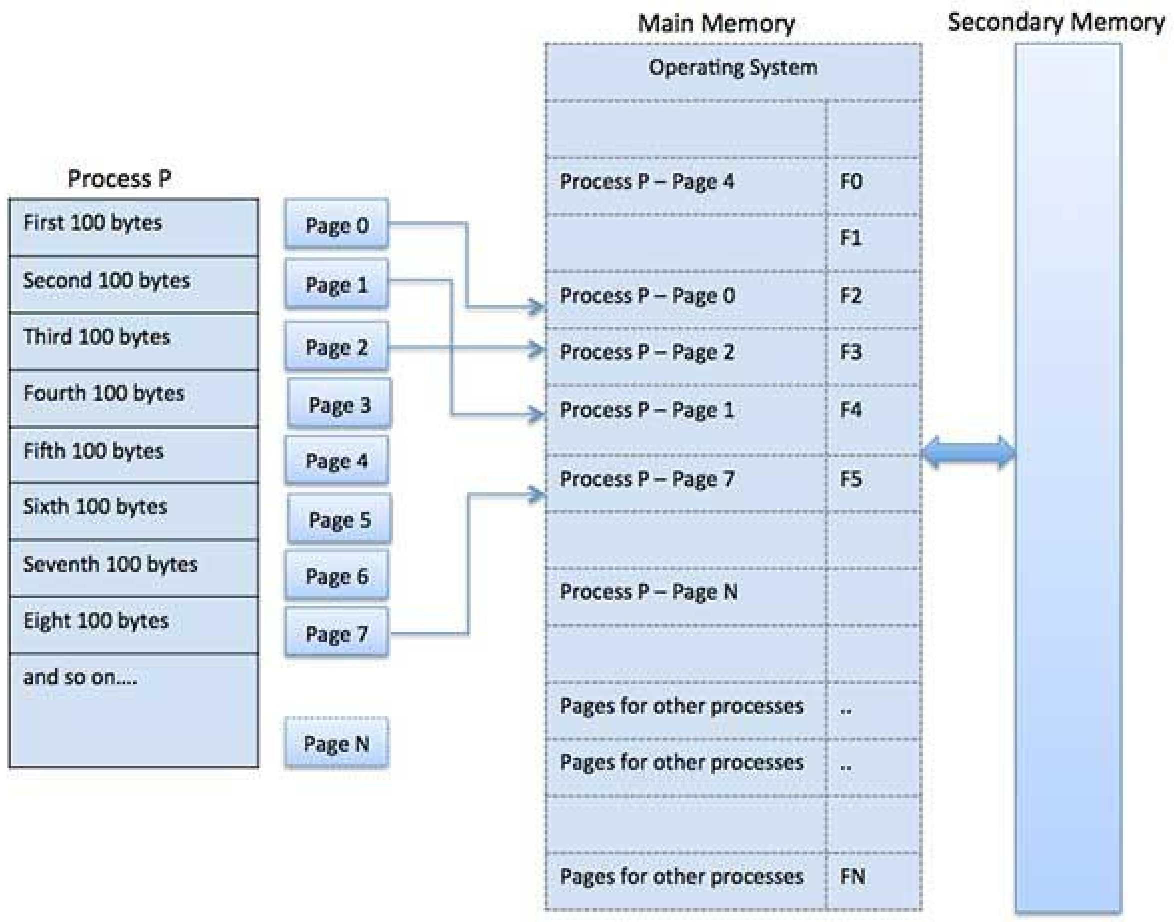
The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

Paging

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.



Address Translation

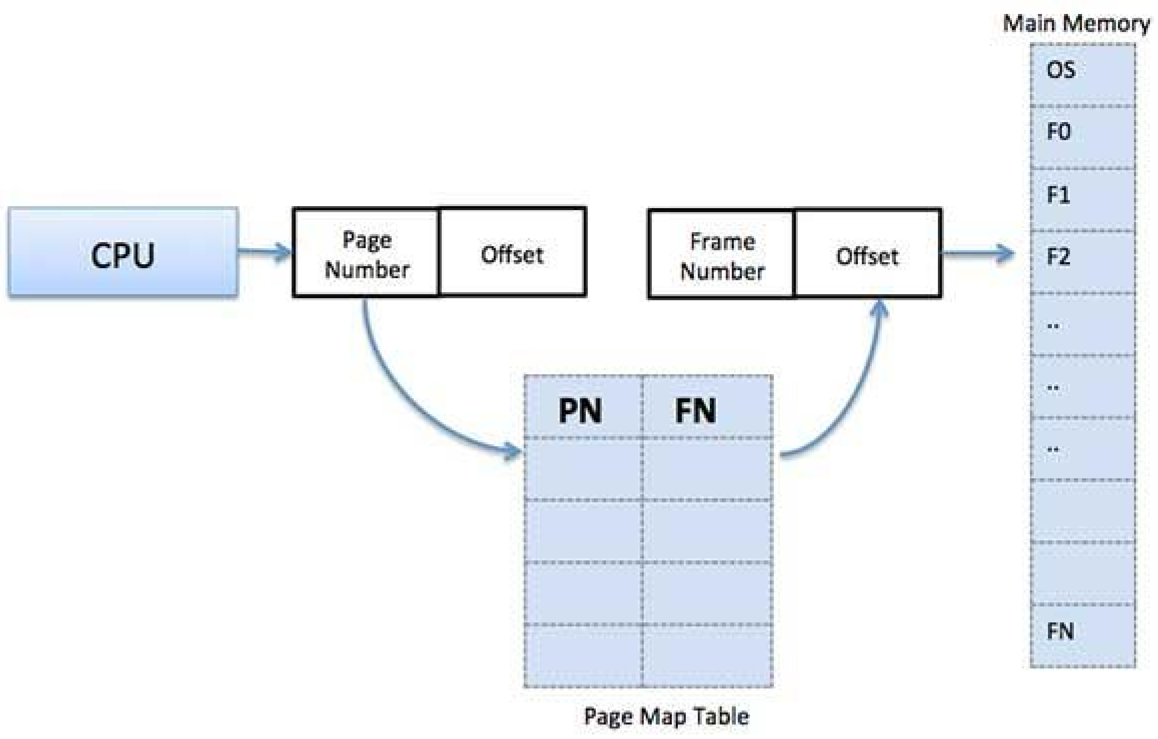
Page address is called **logical address** and represented by **page number** and the **offset**.

Logical Address = Page number + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

Physical Address = Frame number + page offset

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.



When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging −

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

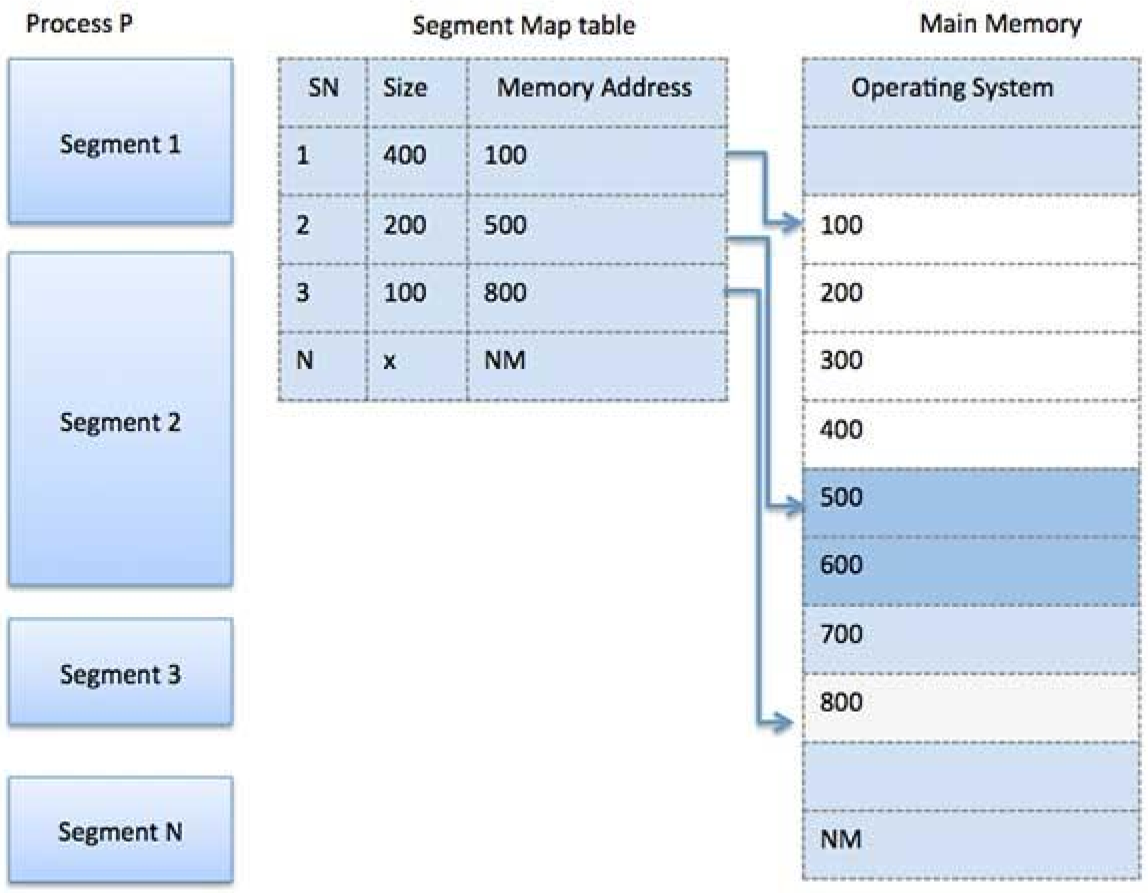
Segmentation

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.

When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.

A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.



Virtual memory

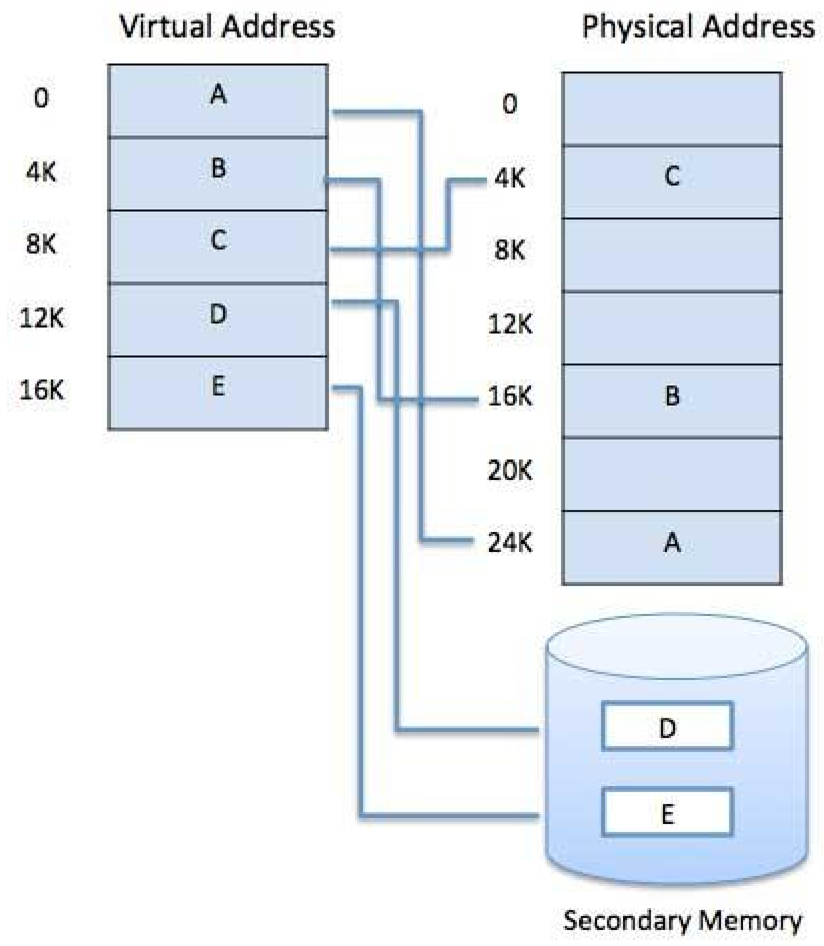
A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.

The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

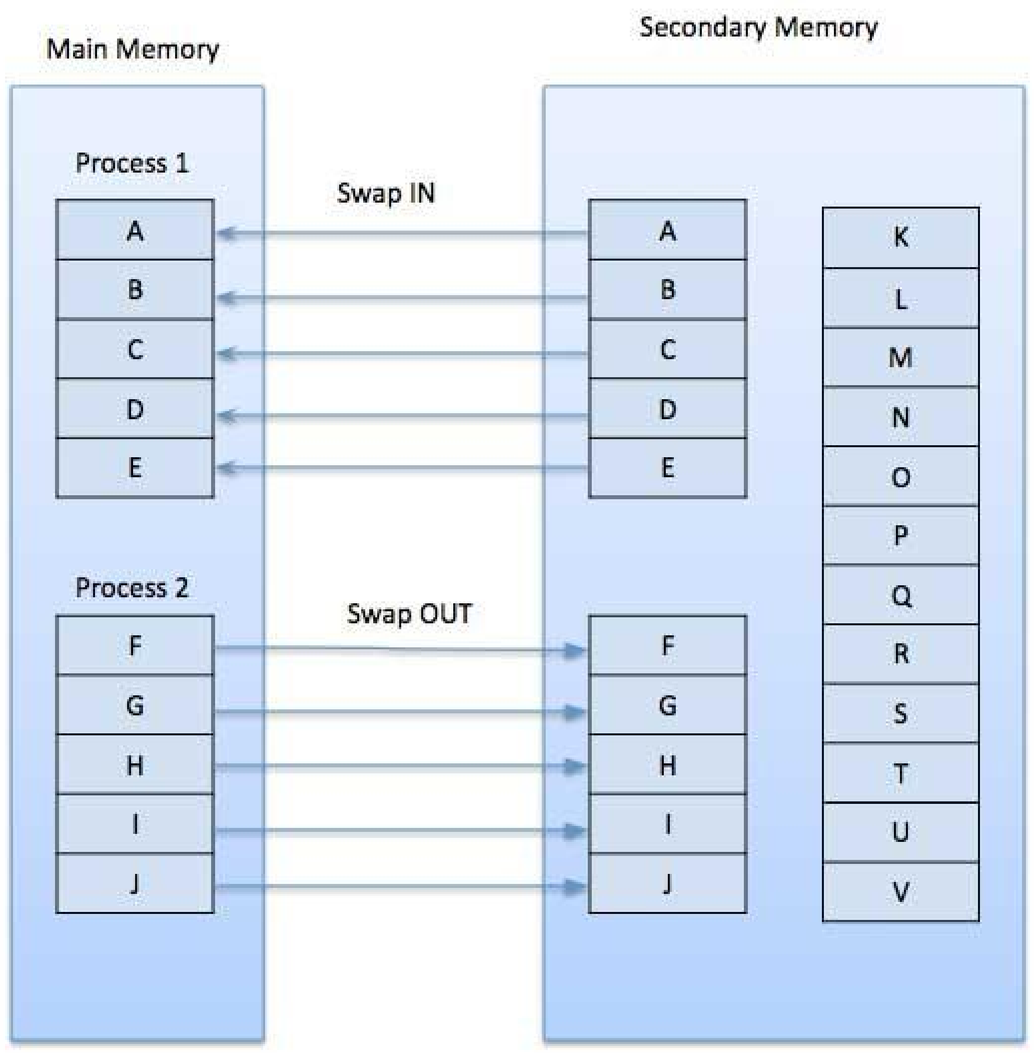
Modern microprocessors intended for general-purpose use, a memory management unit, or MMU, is built into the hardware. The MMU's job is to translate virtual addresses into physical addresses. A basic example is given below −



Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.



While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a **page fault** and transfers control from the program to the operating system to demand the page back into the memory.

Advantages

Following are the advantages of Demand Paging −

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

Disadvantages

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

Page Replacement Algorithm

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

When the page that was selected for replacement and was paged out, is referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

A page replacement algorithm looks at the limited information about accessing the pages provided by hardware, and tries to select which pages should be replaced to minimize the total number of page misses, while balancing it with the costs of primary storage and processor time of the algorithm itself. There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults,

Reference String

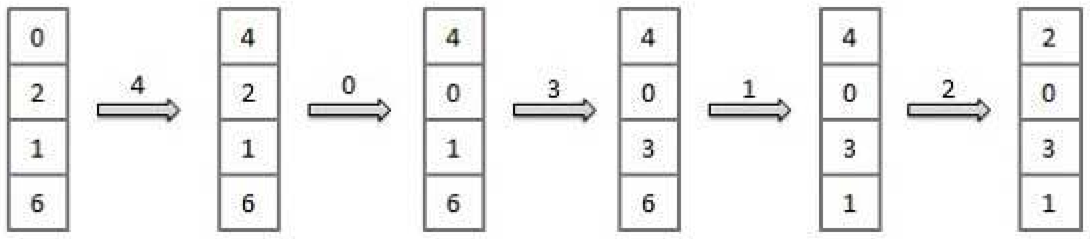
The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data, where we note two things.

- For a given page size, we need to consider only the page number, not the entire address.
- If we have a reference to a page **p**, then any immediately following references to page **p** will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.
- For example, consider the following sequence of addresses − 123,215,600,1234,76,96
- If page size is 100, then the reference string is 1,2,6,12,0,0

First In First Out (FIFO) algorithm

- Oldest page in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages from the tail and add new pages at the head.

```
Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses           :x  x  x  x  x  x      x  x  x
```

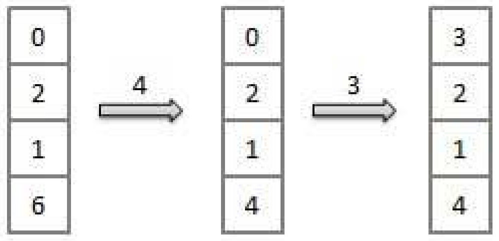


Fault Rate = 9 / 12 = 0.75

Optimal Page algorithm

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN.
- Replace the page that will not be used for the longest period of time. Use the time when a page is to be used.

```
Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses           :x  x  x  x  x       x
```

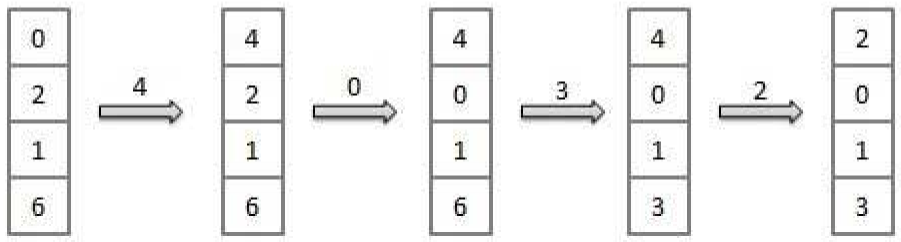


Fault Rate = 6 / 12 = 0.50

Least Recently Used (LRU) algorithm

- Page which has not been used for the longest time in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages by looking back into time.

```
Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses           : x  x  x  x  x  x        x     x
```



Fault Rate = 8 / 12 = 0.67

Page Buffering algorithm

- To get a process start quickly, keep a pool of free frames.
- On page fault, select a page to be replaced.
- Write the new page in the frame of free pool, mark the page table and restart the process.
- Now write the dirty page out of disk and place the frame holding replaced page in free pool.

Least frequently Used(LFU) algorithm

- The page with the smallest count is the one which will be selected for replacement.
- This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

Most frequently Used(MFU) algorithm

- This algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

IO SUB SYSTEM

One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc.

An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories −

- **Block devices** − A block device is one with which the driver communicates by sending entire blocks of data. For example, Hard disks, USB cameras, Disk-On-Key etc.
- **Character devices** − A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc

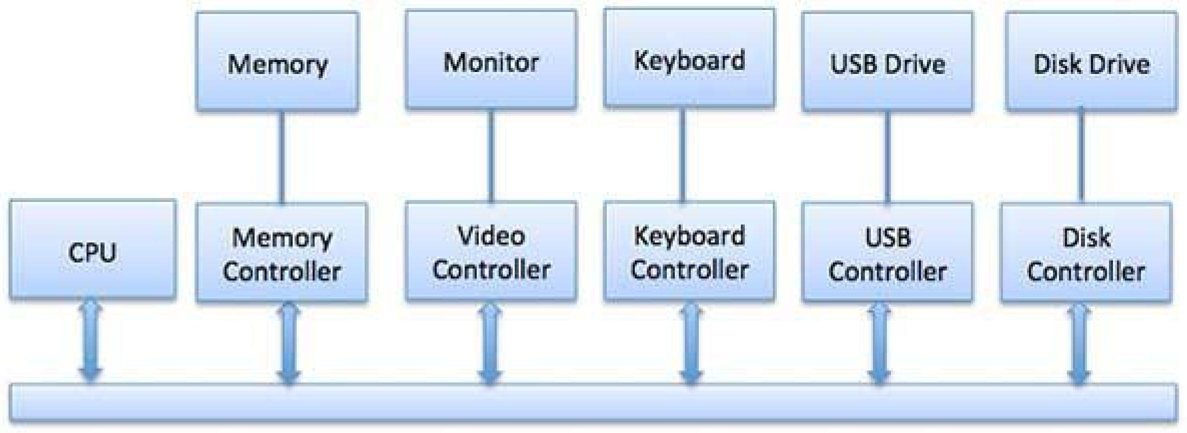
Device Controllers

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices.

The Device Controller works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller.

There is always a device controller and a device driver for each device to communicate with the Operating Systems. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error correction as necessary.

Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller. Following is a model for connecting the CPU, memory, controllers, and I/O devices where CPU and device controllers all use a common bus for communication.



Synchronous vs asynchronous I/O

- **Synchronous I/O** − In this scheme CPU execution waits while I/O proceeds
- **Asynchronous I/O** − I/O proceeds concurrently with CPU execution

Communication to I/O Devices

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

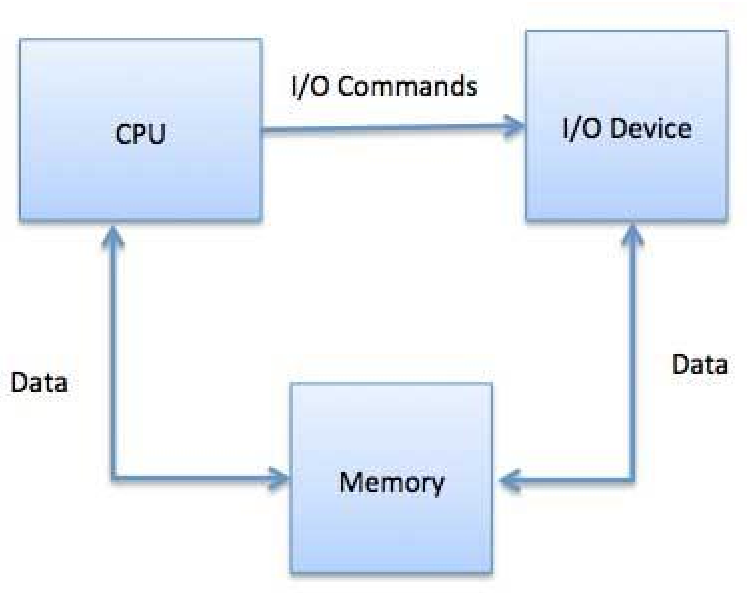
- Special Instruction I/O
- Memory-mapped I/O
- Direct memory access (DMA)

Special Instruction I/O

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

Memory-mapped I/O

When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU.



While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished.

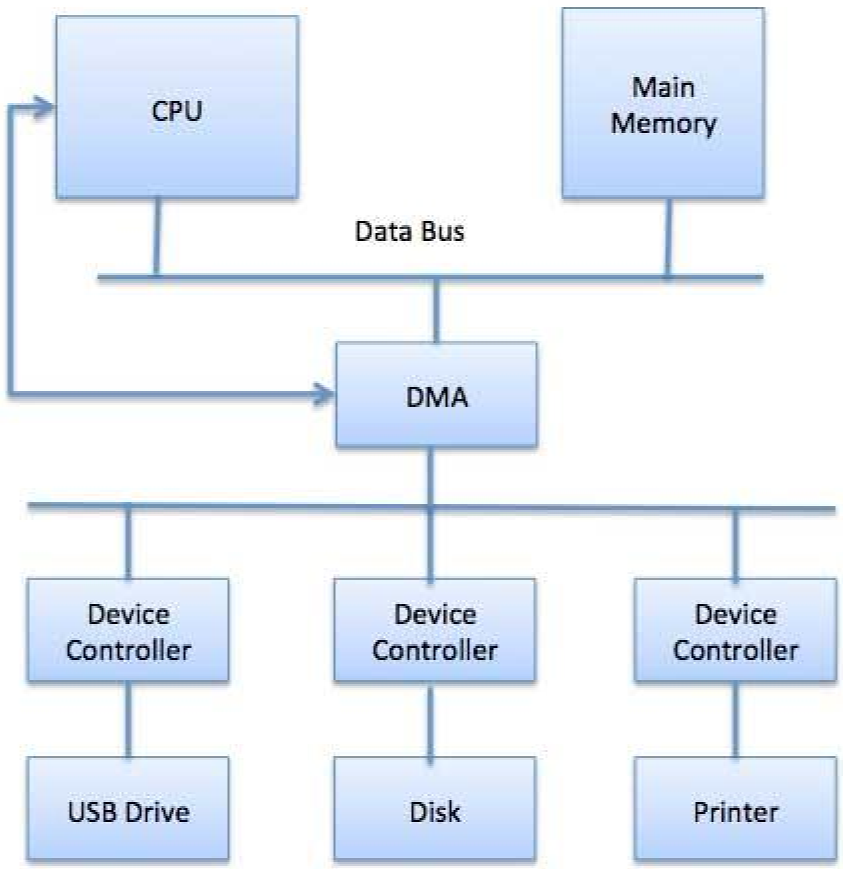
The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

Direct Memory Access (DMA)

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So a typical computer uses direct memory access (DMA) hardware to reduce this overhead.

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.

Direct Memory Access needs a special hardware called DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.



The operating system uses the DMA hardware as follows −

Step	Description
1	Device driver is instructed to transfer disk data to a buffer address X.
2	Device driver then instruct disk controller to transfer data to buffer.
3	Disk controller starts DMA transfer.
4	Disk controller sends each byte to DMA controller.
5	DMA controller transfers bytes to buffer, increases the memory address, decreases the counter C until C becomes zero.

6	When C becomes zero, DMA interrupts CPU to signal transfer completion.

Polling vs Interrupts I/O

A computer must have a way of detecting the arrival of any type of input. There are two ways that this can happen, known as **polling** and **interrupts**. Both of these techniques allow the processor to deal with events that can happen at any time and that are not related to the process it is currently running.

Polling I/O

Polling is the simplest way for an I/O device to communicate with the processor. The process of periodically checking status of the device to see if it is time for the next I/O operation, is called polling. The I/O device simply puts the information in a Status register, and the processor must come and get the information.

Most of the time, devices will not require attention and when one does it will have to wait until it is next interrogated by the polling program. This is an inefficient method and much of the processors time is wasted on unnecessary polls.

Compare this method to a teacher continually asking every student in a class, one after another, if they need help. Obviously the more efficient method would be for a student to inform the teacher whenever they require assistance.

Interrupts I/O

An alternative scheme for dealing with I/O is the interrupt-driven method. An interrupt is a signal to the microprocessor from a device that requires attention.

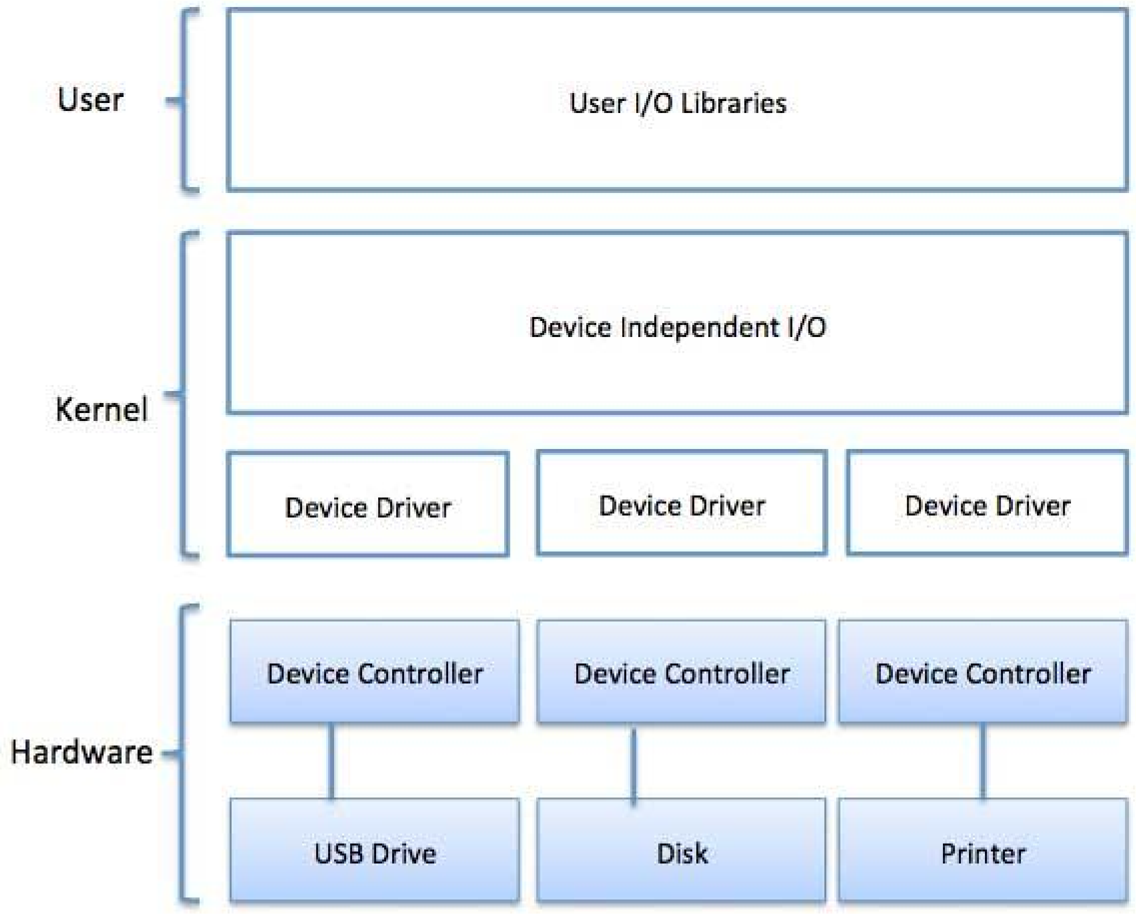
A device controller puts an interrupt signal on the bus when it needs CPU's attention when CPU receives an interrupt, It saves its current state and invokes the appropriate interrupt handler using the interrupt vector (addresses of OS routines to handle various events). When the interrupting device has been dealt with, the CPU continues with its original task as if it had never been interrupted.

IO subsystem software

I/O software is often organized in the following layers −

- **User Level Libraries** − This provides simple interface to the user program to perform input and output. For example, **stdio** is a library provided by C and C++ programming languages.
- **Kernel Level Modules** − This provides device driver to interact with the device controller and device independent I/O modules used by the device drivers.
- **Hardware** − This layer includes actual hardware and hardware controller which interact with the device drivers and makes hardware alive.

A key concept in the design of I/O software is that it should be device independent where it should be possible to write programs that can access any I/O device without having to specify the device in advance. For example, a program that reads a file as input should be able to read a file on a floppy disk, on a hard disk, or on a CD-ROM, without having to modify the program for each different device.



Device Drivers

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices. Device drivers encapsulate device-dependent code and implement a standard interface in such a way that code contains device-specific register reads/writes. Device driver, is generally written by the device's manufacturer and delivered along with the device on a CD-ROM.

A device driver performs the following jobs −

- To accept request from the device independent software above to it.
- Interact with the device controller to take and give I/O and perform required error handling

- Making sure that the request is executed successfully

How a device driver handles a request is as follows: Suppose a request comes to read a block N. If the driver is idle at the time a request arrives, it starts carrying out the request immediately. Otherwise, if the driver is already busy with some other request, it places the new request in the queue of pending requests.

Interrupt handlers

An interrupt handler, also known as an interrupt service routine or ISR, is a piece of software or more specifically a callback function in an operating system or more specifically in a device driver, whose execution is triggered by the reception of an interrupt.

When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt, updates data structures and wakes up process that was waiting for an interrupt to happen.

The interrupt mechanism accepts an address ─ a number that selects a specific interrupt handling routine/function from a small set. In most architectures, this address is an offset stored in a table called the interrupt vector table. This vector contains the memory addresses of specialized interrupt handlers.

Device-Independent I/O Software

The basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software. Though it is difficult to write completely device independent software but we can write some modules which are common among all the devices. Following is a list of functions of device-independent I/O Software −

- Uniform interfacing for device drivers
- Device naming - Mnemonic names mapped to Major and Minor device numbers
- Device protection
- Providing a device-independent block size
- Buffering because data coming off a device cannot be stored in final destination.
- Storage allocation on block devices
- Allocation and releasing dedicated devices
- Error Reporting

User-Space I/O Software

These are the libraries which provide richer and simplified interface to access the functionality of the kernel or ultimately interactive with the device drivers. Most of the user-level I/O software consists of library procedures with some exception like spooling system which is a way of dealing with dedicated I/O devices in a multiprogramming system.

I/O Libraries (e.g., stdio) are in user-space to provide an interface to the OS resident device-independent I/O SW. For example putchar(), getchar(), printf() and scanf() are example of user level I/O library stdio available in C programming.

Kernel I/O Subsystem

Kernel I/O Subsystem is responsible to provide many services related to I/O. Following are some of the services provided.

- **Scheduling** − Kernel schedules a set of I/O requests to determine a good order in which to execute them. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The Kernel I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by the applications.
- **Buffering** − Kernel I/O Subsystem maintains a memory area known as **buffer** that stores data while they are transferred between two devices or between a device with an application operation. Buffering is done to cope with a speed mismatch between the producer and consumer of a data stream or to adapt between devices that have different data transfer sizes.
- **Caching** − Kernel maintains cache memory which is region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original.
- **Spooling and Device Reservation** − A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In other operating systems, it is handled by an in kernel thread.
- **Error Handling** − An operating system that uses protected memory can guard against many kinds of hardware and application errors.

File subsystem

File

A file is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.

File Structure

A File Structure should be according to a required format that the operating system can understand.

- A file has a certain defined structure according to its type.
- A text file is a sequence of characters organized into lines.
- A source file is a sequence of procedures and functions.
- An object file is a sequence of bytes organized into blocks that are understandable by the machine.
- When operating system defines different file structures, it also contains the code to support these file structure. Unix, MS-DOS support minimum number of file structure.

File Type

File type refers to the ability of the operating system to distinguish different types of file such as text files source files and binary files etc. Many operating systems support many types of files. Operating system like MS-DOS and UNIX have the following types of files −

Ordinary files

- These are the files that contain user information.
- These may have text, databases or executable program.
- The user can apply various operations on such files like add, modify, delete or even remove the entire file.

Directory files

- These files contain list of file names and other information related to these files.

Special files

- These files are also known as device files.

- These files represent physical device like disks, terminals, printers, networks, tape drive etc.

These files are of two types −

- **Character special files** − data is handled character by character as in case of terminals or printers.
- **Block special files** − data is handled in blocks as in the case of disks and tapes.

File Access Mechanisms

File access mechanism refers to the manner in which the records of a file may be accessed. There are several ways to access files −

- Sequential access
- Direct/Random access
- Indexed sequential access

Sequential access

A sequential access is that in which the records are accessed in some sequence, i.e., the information in the file is processed in order, one record after the other. This access method is the most primitive one. Example: Compilers usually access files in this fashion.

Direct/Random access

- Random access file organization provides, accessing the records directly.
- Each record has its own address on the file with by the help of which it can be directly accessed for reading or writing.
- The records need not be in any sequence within the file and they need not be in adjacent locations on the storage medium.

Indexed sequential access

- This mechanism is built up on base of sequential access.
- An index is created for each file which contains pointers to various blocks.
- Index is searched sequentially and its pointer is used to access the file directly.

Space Allocation

Files are allocated disk spaces by operating system. Operating systems deploy following three main ways to allocate disk space to files.

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

Contiguous Allocation

- Each file occupies a contiguous address space on disk.
- Assigned disk address is in linear order.
- Easy to implement.
- External fragmentation is a major issue with this type of allocation technique.

Linked Allocation

- Each file carries a list of links to disk blocks.
- Directory contains link / pointer to first block of a file.
- No external fragmentation
- Effectively used in sequential access file.
- Inefficient in case of direct access file.

Indexed Allocation

- Provides solutions to problems of contiguous and linked allocation.
- A index block is created having all pointers to files.
- Each file has its own index block which stores the addresses of disk space occupied by the file.
- Directory contains the addresses of index blocks of files.

Security

Security refers to providing a protection system to computer system resources such as CPU, memory, disk, software programs and most importantly data/information stored in the computer system. If a computer program is run by an unauthorized user, then he/she may cause severe damage to computer or data stored in it. So a computer system must be protected against unauthorized access, malicious access to system memory, viruses, worms etc. We're going to discuss following topics in this chapter.

- Authentication
- One Time passwords
- Program Threats
- System Threats
- Computer Security Classifications

Authentication

Authentication refers to identifying each user of the system and associating the executing programs with those users. It is the responsibility of the Operating System to create a protection system which ensures that a user who is running a particular program is authentic. Operating Systems generally identifies/authenticates users using following three ways −

- **Username / Password** − User need to enter a registered username and password with Operating system to login into the system.
- **User card/key** − User need to punch card in card slot, or enter key generated by key generator in option provided by operating system to login into the system.
- **User attribute - fingerprint/ eye retina pattern/ signature** − User need to pass his/her attribute via designated input device used by operating system to login into the system.

One Time passwords

One-time passwords provide additional security along with normal authentication. In One-Time Password system, a unique password is required every time user tries to login into the system. Once a one-time password is used, then it cannot be used again. One-time password are implemented in various ways.

- **Random numbers** − Users are provided cards having numbers printed along with corresponding alphabets. System asks for numbers corresponding to few alphabets randomly chosen.
- **Secret key** − User are provided a hardware device which can create a secret id mapped with user id. System asks for such secret id which is to be generated every time prior to login.

- **Network password** − Some commercial applications send one-time passwords to user on registered mobile/ email which is required to be entered prior to login.

Program Threats

Operating system's processes and kernel do the designated task as instructed. If a user program made these process do malicious tasks, then it is known as **Program Threats**. One of the common example of program threat is a program installed in a computer which can store and send user credentials via network to some hacker. Following is the list of some well-known program threats.

- **Trojan Horse** − Such program traps user login credentials and stores them to send to malicious user who can later on login to computer and can access system resources.
- **Trap Door** − If a program which is designed to work as required, have a security hole in its code and perform illegal action without knowledge of user then it is called to have a trap door.
- **Logic Bomb** − Logic bomb is a situation when a program misbehaves only when certain conditions met otherwise it works as a genuine program. It is harder to detect.
- **Virus** − Virus as name suggest can replicate themselves on computer system. They are highly dangerous and can modify/delete user files, crash systems. A virus is generatlly a small code embedded in a program. As user accesses the program, the virus starts getting embedded in other files/ programs and can make system unusable for user

System Threats

System threats refers to misuse of system services and network connections to put user in trouble. System threats can be used to launch program threats on a complete network called as program attack. System threats creates such an environment that operating system resources/ user files are misused. Following is the list of some well-known system threats.

- **Worm** − Worm is a process which can choked down a system performance by using system resources to extreme levels. A Worm process generates its multiple copies where each copy uses system resources, prevents all other processes to get required resources. Worms processes can even shut down an entire network.
- **Port Scanning** − Port scanning is a mechanism or means by which a hacker can detects system vulnerabilities to make an attack on the system.
- **Denial of Service** − Denial of service attacks normally prevents user to make legitimate use of the system. For example, a user may not be able to use internet if denial of service attacks browser's content settings.

Computer Security Classifications

As per the U.S. Department of Defense Trusted Computer System's Evaluation Criteria there are four security classifications in computer systems: A, B, C, and D. This is widely used specifications to determine and model the security of systems and of security solutions. Following is the brief description of each classification.

S.N.	Classification Type & Description
1	**Type A** Highest Level. Uses formal design specifications and verification techniques. Grants a high degree of assurance of process security.
2	**Type B** Provides mandatory protection system. Have all the properties of a class C2 system. Attaches a sensitivity label to each object. It is of three types. • **B1** − Maintains the security label of each object in the system. Label is used for making decisions to access control. • **B2** − Extends the sensitivity labels to each system resource, such as storage objects, supports covert channels and auditing of events. • **B3** − Allows creating lists or user groups for access-control to grant access or revoke access to a given named object.
3	**Type C** Provides protection and user accountability using audit capabilities. It is of two types. • **C1** − Incorporates controls so that users can protect their private information and keep other users from accidentally reading / deleting their data. UNIX versions are mostly Cl class. • **C2** − Adds an individual-level access control to the capabilities of a Cl level system.
4	**Type D** Lowest level. Minimum protection. MS-DOS, Window 3.1 fall in this category.