**OPERATING SYSTEMS**

Chapter 1 Topics :
Operating systems Objectives and functions, Gaining the role of operating systems, The evolution of OS,Major achievements , OS Design considerations for multiprocessor and multicore.
Process Concept : Processes PCB, Threads – overview and its benefits, process scheduling : Scheduling queues, Schedulers, Context switch, Operations on processes – Process creation, process termination, understanding fork(), wait(), exit()
Interprocess Communication- Shared memory, message passing, pipe(), . Understanding the need of IPC.
Process synchronization :
Background, critical section problem, understating the race condition and the need for the process synchronization.

# 1. Operating Systems :

**Various Definitions of OS includes**

- An interface between user and hardware
- Operating System - Operating systems are those programs that interface the machine with the applications programs. The main function of these systems is to dynamically allocate the shared system resources to the executing programs.
- A program that controls the execution of application programs

**Main objectives of an OS**

- Convenience - Make the computer system convenient to use
- Efficiency -OS allows the system resources to be used in efficient manner
- Ability to evolve – An OS should be constructed in such a way that as it permits the effective testing and development.

**OS Services/Functions**

Briefly, the OS typically provides services in the following areas:

• **Program development:** The OS provides a variety of facilities and services, such as editors and debuggers, to assist the programmer in creating programs. and are referred to as application program development tools.

• **Program execution:** A number of steps need to be performed to execute a program. Instructions and data must be loaded into main memory, I/O devices and files must be initialized, and other resources must be prepared. The OS handles these scheduling duties for the user.

• **Access to I/O devices**: Each I/O device requires its own peculiar set of instructions or control signals for operation. The OS provides a uniform interface that hides these details so that programmers can access such devices using simple reads and writes.

• **Controlled access to files:** For file access, the OS must reflect a detailed understanding of not only the nature of the I/O device (disk drive, tape drive) but also the structure of the data contained in the files on the storage medium.

• **System access:** For shared or public systems, the OS controls access to the system as a whole and to specific system resources. The access function must provide protection of resources and data from unauthorized users and must resolve conflicts for resource contention.

• **Error detection and response:** A variety of errors can occur while a computer system is running. These include internal and external hardware errors, such as a memory error, or a device failure or malfunction; and various software errors, such as division by zero, attempt to access forbidden memory location, and inability of the OS to grant the request of an application.

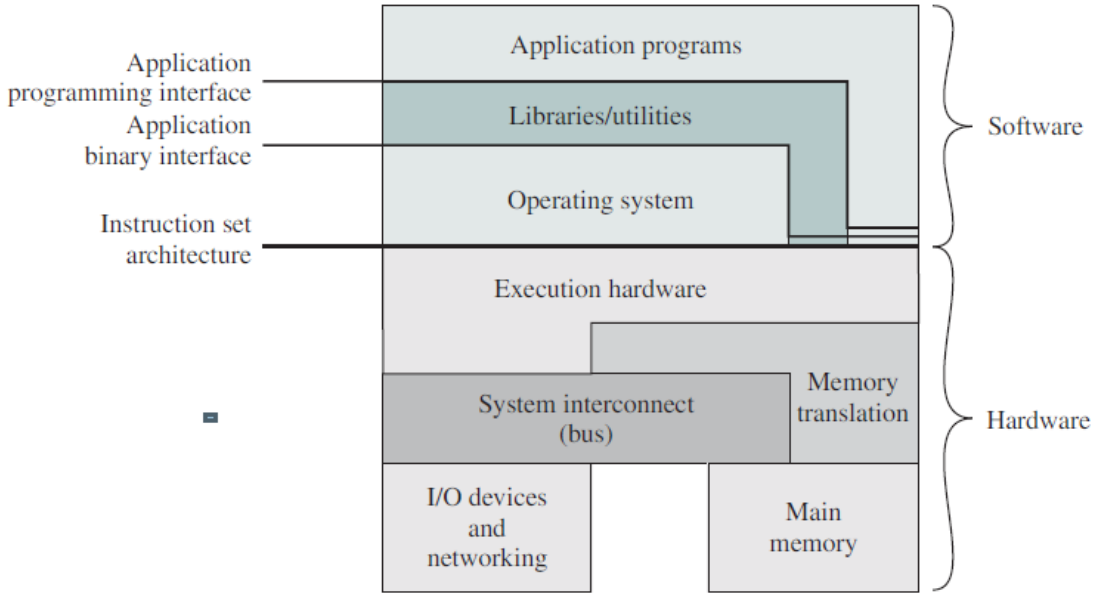
• **Accounting: A good OS will collect usage statistics for various resources and** monitor performance parameters such as response time.

## Role of Operating System :

A computer is a set of resources for the movement, storage, and processing of data and for the control of these functions. **The OS is responsible for managing these resources.**

### a. The Operating System as a User/Computer Interface

The hardware and software used in providing applications to a user can be viewed in a layered or hierarchical fashion, as depicted in Figure . The user of those applications, the end user, generally is not concerned with the details of computer hardware. Thus, the end user views a computer system in terms of a set of applications. An application can be expressed in a programming language and is developed by an application programmer. If one were to develop an application program as a set of machine instructions that is completely responsible for controlling the computer hardware, one would be faced with an overwhelmingly complex undertaking.




The above figure indicates three key interfaces in a typical computer system:

• **Instruction set architecture (ISA) :** The ISA defines the repertoire of machine language instructions that a computer can follow. This interface is the boundary between hardware and software. Note that both application programs and utilities may access the ISA directly. For these programs, a subset of the instruction repertoire is available (user ISA). The OS has access to additional machine language instructions that deal with managing system resources (system ISA).

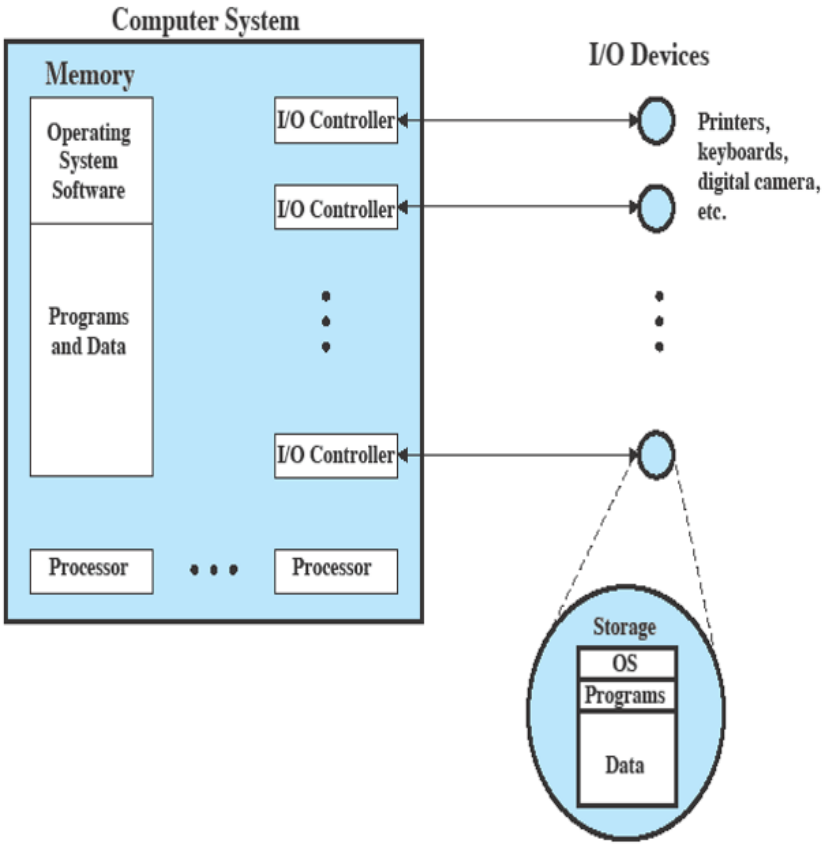
• **Application binary interface (ABI) :** The ABI defines a standard for binary portability across programs. The ABI defines the system call interface to the operating system and the hardware resources and services available in a system through the user ISA.

• **Application programming interface (API) :** The API gives a program access to the hardware resources and services available in a system through the user ISA supplemented with high-level language (HLL) library calls.

### B. Operating system as a resource manager

Figure  suggests the main resources that are managed by the OS. A portion of the OS is in main memory. This includes the **kernel , or nucleus , which contains** the most frequently used

functions in the OS and, at a given time, other portions of the OS currently in use. The remainder of main memory contains user programs and data. The memory management hardware in the processor and the OS jointly control the allocation of main memory, as we shall see. The OS decides when an I/O device can be used by a program in execution and controls access to and use of files.
The processor itself is a resource, and the OS must determine how much processor time is to be devoted to the execution of a particular user program.




## Evolution of Operating Systems

A major OS will evolve over time for a number of reasons:

• **Hardware upgrades plus new types of hardware: T**he use of graphics terminals and page-mode terminals instead of line-at-a time scroll mode terminals affects OS design. For example, a graphics terminal typically allows the user to view several applications at the same time through "windows" on the screen. This requires more sophisticated support in the OS.
• **New services:** In response to user demand or in response to the needs of system managers, the OS expands to offer new services.
• **Fixes: Any OS has faults.** These are discovered over the course of time and fixes are made. Of course, the fix may introduce new faults. The need to change an OS regularly places certain requirements on its design. The OS has a long series in evolution.

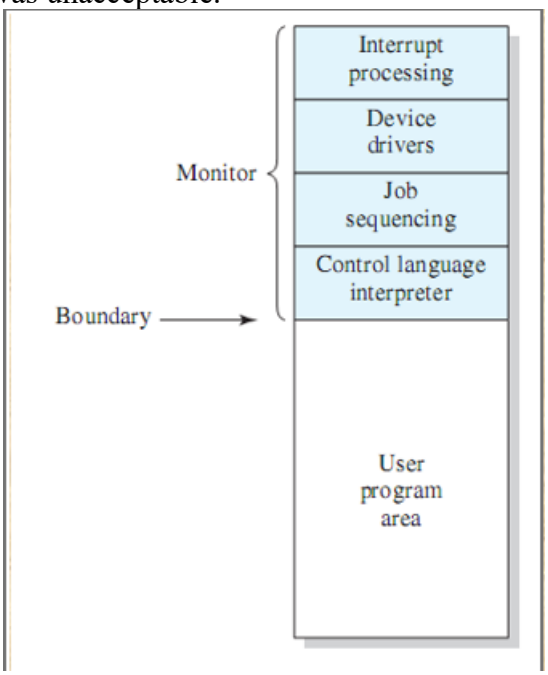
### 1. Serial Processing

With the earliest computers, from the late 1940s to the mid-1950s, the programmer interacted directly with the computer hardware; there was no OS. These computers were run from a console consisting of display lights, toggle switches, some form of input device, and a printer. Programs in machine code were loaded via the input device (e.g., a card reader). If an error halted the program, the error condition was indicated by the lights. If the program proceeded to a normal completion, the output appeared on the printer.
These early systems presented two main problems:
• **Scheduling :** Most installations used a hardcopy sign-up sheet to reserve computer time
• **Setup time:** A single program, called a **job** , could involve loading the compiler plus the high-level language program (source program) into memory, saving the compiled program (object program) and then loading and linking together the object program and common functions. Each

of these steps could involve mounting or dismounting tapes or setting up card decks. If an error occurred, the hapless user typically had to go back to the beginning of the
setup sequence. Thus, a considerable amount of time was spent just in setting up the program to run. This mode of operation could be termed *serial processing , reflecting the fact* that users have access to the computer in series.
.

2. **Simple Batch Processing**
   **Monitor point of view: The monitor controls the sequence of events. For this** to be so, much of the monitor must always be in main memory and available for execution. That portion is referred to as the **resident monitor.** The rest of the monitor consists of utilities and common functions that are loaded as subroutines to the user program at the beginning of any job that requires them. The monitor reads in jobs one at a time from the input device (typically a card reader or magnetic tape drive). As it is read in, the current job is placed in the user program area, and control is passed to this job. When the job is completed, it returns control to the monitor, which immediately reads in the next job. The results of each job are sent to an output device, such as a printer, for delivery to the user.

Early computers were very expensive, and therefore it was important to maximize processor utilization. The wasted time due to scheduling and setup time was unacceptable.
To improve utilization, the concept of a batch OS was developed. It appears that the first batch OS (and the first OS of any kind) was developed in the mid-1950s by General Motors for use on an IBM 701.
The central idea behind the simple batch-processing scheme is the use of a piece of software known as the **monitor .**
**With this type of OS, the user no longer has**
direct access to the processor. Instead, the user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device, for use by the monitor. Each program is constructed to branch back to the monitor when it completes processing, at which point the monitor automatically begins loading the next program.




**Job Control Language**
The monitor performs a scheduling function: A batch of jobs is queued up, and jobs are executed as rapidly as possible, with no intervening idle time. With each job, instructions are included in a primitive form of **job control language (JCL) . This is a special type of programming** language used to provide instructions to the monitor. A simple example is that of a user submitting a program written in the programming language FORTRAN plus some data to be used by the program. All FORTRAN instructions and data are on a separate punched card or a separate record on tape. In addition to FORTRAN and data lines, the job includes job control instructions, which are denoted by the beginning $. The monitor, or batch OS, is simply a computer program.
**Memory protection:** While the user program is executing, it must not alter the memory area containing the monitor.
**Timer: A timer is used to prevent a single job from monopolizing the system.** The timer is set at the beginning of each job. If the timer expires, the user program is stopped, and control returns to the monitor.

**Privileged instructions: Certain machine level instructions are designated privileged** and can be executed only by the monitor. If the processor encounters such an instruction while executing a user program, an error occurs causing control to be transferred to the monitor. Among the privileged instructions are I/O instructions, so that the monitor retains control of all I/O devices.

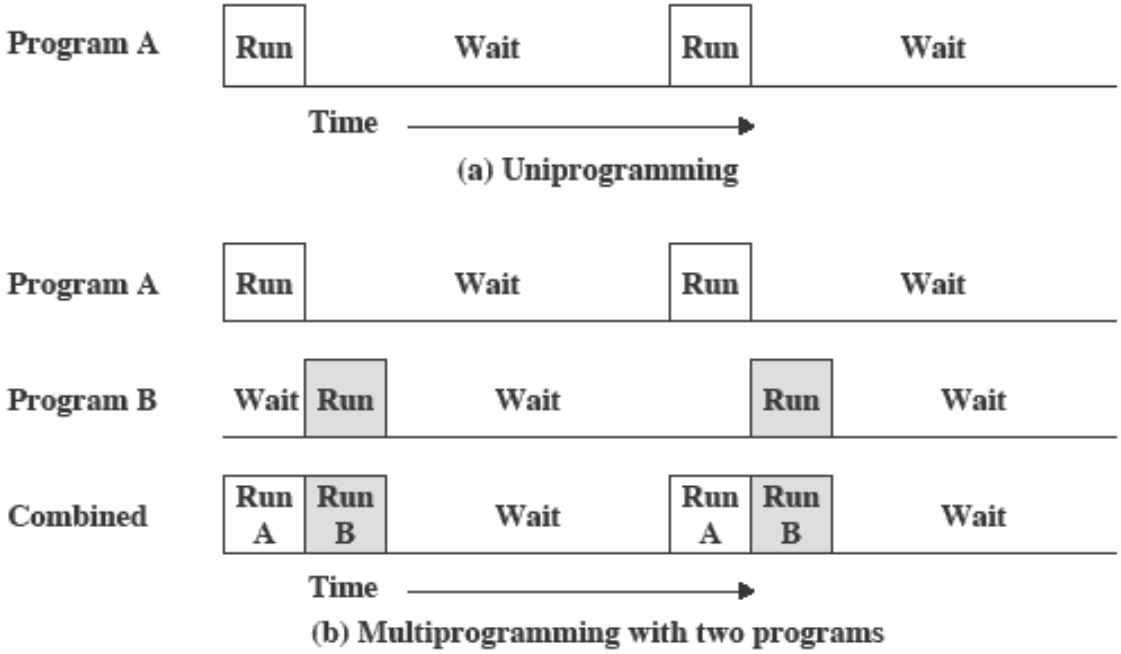
**Interrupts:** Early computer models did not have this capability.

**Modes of Operation**

Considerations of memory protection and privileged instructions lead to the concept of modes of operation. A user program executes in a **user mode , in which** certain areas of memory are protected from the user's use and in which certain instructions may not be executed. The monitor executes in a system mode, or what has come to be called **kernel mode , in which privileged instructions may be executed** and in which protected areas of memory may be accessed.

### 3. Multi programmed systems

Even with the automatic job sequencing provided by a simple batch OS, the processor is often idle. The problem is that I/O devices are slow compared to the processor. The processor spends a certain amount of time executing, until it reaches an I/O instruction; it must then wait until that I/O instruction concludes before proceeding




- There must be enough memory to hold the OS (resident monitor) and one user program
- When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O
- Multiprogramming
    - also known as multitasking
    - memory is expanded to hold three, four, or more programs and switch among all of them

### 4. Time-Sharing Systems

In multiprogrammed batch systems, the user cannot interact with the program during its execution. In time-sharing or multitasking systems, multiple jobs are executed by the CPU switching between them, but the switching occurs so frequently that the user may interact with each program while it is running. Time-sharing operating systems:

- uses CPU scheduling and multiprogramming,

- uses time-slice mechanism,
- allows interactive I/O,
- allows many users to share the computer simultaneously.

### 5. Personal Computer (PC) /Desktop Systems

A computer system dedicated to a single user is referred to as a PC. In the first PCs, the operating system was neither multiuser nor multitasking (eg. MS-DOS). The operating system concepts used in mainframes and minicomputers, today, are also used in PCs (eg. UNIX, Microsoft Windows NT, Macintosh OS).

### 6. Parallel Systems/Multiprocessor systems

Parallel systems have more than one processor. In multiprocessor systems (tightly coupled), processors are in close communication, such as they share computer bus, clock, memory or peripherals.

### 7. Clustered Systems

- Like parallel systems, clustered systems gather together multiple CPUs to accomplish computational work.
- Clustered systems differ from parallel systems, however, in that they are composed of two or more individual systems coupled together.
- The definition of the term clustered is **not concrete;** the general accepted definition is that clustered computers share storage and are closely linked via LAN networking.
- Clustering is usually performed to provide **high availability**.

### 8. Distributed Systems

Distributed systems also have more than one processor distributed in different geographical locations. Each processor has its local memory. Processors communicate through communication lines (eg. Telephone lines, high-speed bus, etc.). Processors are referred to as sites, nodes, computers depending on the context in which they are mentioned. Multicomputer systems are loosely coupled. Example applications are e-mail, web server, etc.

### 9. Real-time Systems

Real-time systems are special purpose operating systems. They are used when there are rigid time requirements on the operation of a processor or the flow of data, and thus it is often used as a control device in a dedicated application (eg. fuel injection systems, weapon systems, industrial control systems, …). It has well defined, fixed time constraints. The processing must be done within the defined constraints, or the system fails.

Two types:

*Hard real-time* systems guarantee that critical tasks complete on time.

In *Soft real-time* systems, a critical real-time task gets priority over other tasks, and the task retains that priority until it completes.

### 10. Handheld Systems

Handheld systems include **Personal Digital Assistants(PDAs)**, such as Palm-Pilots or Cellular Telephones with connectivity to a network such as the Internet. They are usually of limited size due to which most handheld devices have a small amount of memory, include slow processors, and feature small display screens.

## Major Advances

Operating systems are among the most complex pieces of software ever developed. The following provides four major theoretical advances in the development of operating systems:

- Processes
- Memory management
- Information protection and security
- Scheduling and resource management

**Process :**

Three major lines of computer system development created problems in **timing and synchronization** that contributed to the development of the concept of the process: multiprogramming batch operation, time sharing, and real-time transaction systems. As we have seen, multiprogramming was designed to keep the processor and I/O devices, including storage devices, simultaneously busy to achieve maximum efficiency. The key mechanism is this: In response to signals indicating the completion of I/O transactions, the processor is switched among the various programs residing in main memory.

A second line of development was **general-purpose time sharing**. Here, the key design objective is to be responsive to the needs of the individual user and yet, for cost reasons, be able to support many users simultaneously. These goals are compatible because of the relatively slow reaction time of the user

A third important line of development has been **real-time transaction processing** systems. In this case, a number of users are entering queries or updates against a database. An example is an airline reservation system. The key difference between the transaction processing system and the time-sharing system is that the former is limited to one or a few applications, whereas users of a time-sharing system can engage in program development, job execution, and the use of various applications.

**Memory management**

The needs of users can be met best by a computing environment that supports modular programming and the flexible use of data. System managers need efficient and orderly control of storage allocation. The OS, to satisfy these requirements, has five principal storage management responsibilities:

- Process isolation: The OS must prevent independent processes from interfering with each other's memory, both data and instructions.
- Automatic allocation and management: Programs should be dynamically allocated across the memory hierarchy as required. Allocation should be transparent to the programmer. Thus, the programmer is relieved of concerns relating to memory limitations, and the OS can achieve efficiency by assigning memory to jobs only as needed.
- Support of modular programming: Programmers should be able to define program modules, and to create, destroy, and alter the size of modules dynamically.
- Protection and access control: Sharing of memory, at any level of the memory hierarchy, creates the potential for one program to address the memory space of another. This is desirable when sharing is needed by particular applications. At other times, it threatens the integrity of programs and even of the OS itself. The OS must allow portions of memory to be accessible in various ways by various users.
- Long-term storage: Many application programs require means for storing information for extended periods of time, after the computer has been powered down.

**Information Protection and Security**

The growth in the use of time-sharing systems and, more recently, computer networks has brought with it a growth in concern for the protection of information. The nature of the threat that concerns an organization will vary greatly depending on the circumstances. However, there are some general-purpose tools that can be built into computers and operating systems that

support a variety of protection and security mechanisms. In general, we are concerned with the problem of controlling access to computer systems and the information stored in them.
Much of the work in security and protection as it relates to operating systems can be roughly grouped into four categories:
• Availability: Concerned with protecting the system against interruption.
• Confidentiality: Assures that users cannot read data for which access is unauthorized.
• Data integrity: Protection of data from unauthorized modification.
• Authenticity: Concerned with the proper verification of the identity of users and the validity of messages or data.

**Scheduling and resource management**

- Key responsibility of an OS is managing resources
- Resource allocation policies must consider: efficiency, fairness and differential responsiveness

## Multiprocessor and multicore design considerations:

In an SMP system, the kernel can execute on any processor, and typically each processor does self-scheduling from the pool of available processes or threads. The kernel can be constructed as multiple processes or multiple threads, allowing portions of the kernel to execute in parallel. The SMP approach complicates the OS. The OS designer must deal with the complexity due to sharing resources (like data structures) and coordinating actions (like accessing devices) from multiple parts of the OS executing at the same time. Techniques must be employed to resolve and synchronize claims to resources.
An SMP operating system manages processor and other computer resources so that the user may view the system in the same fashion as a multiprogramming uniprocessor system. A user may construct applications that use multiple processes or multiple threads within processes without regard to whether a single processor or multiple processors will be available. Thus, a multiprocessor OS must provide all the functionality of a multiprogramming system plus additional features to accommodate multiple processors. The key design issues include the following:
• **Simultaneous concurrent processes or threads:** Kernel routines need to be reentrant to allow several processors to execute the same kernel code simultaneously. With multiple processors executing the same or different parts of the kernel, kernel tables and management structures must be managed properly to avoid data corruption or invalid operations.
• **Scheduling:** Any processor may perform scheduling, which complicates the task of enforcing a scheduling policy and assuring that corruption of the scheduler data structures is avoided. If kernel-level multithreading is used, then the opportunity exists to schedule multiple threads from the same process simultaneously on multiple processors.
• **Synchronization:** With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective synchronization. Synchronization is a facility that enforces mutual exclusion and event ordering.
• **Memory management:** In addition to basic memeory management need, the OS needs to exploit the available hardware parallelism to achieve the best performance.
**Reliability and fault tolerance:** The OS should provide graceful degradation in the face of processor failure. Because multiprocessor OS design issues generally involve extensions to solutions to multiprogramming uniprocessor design problems

**Multicore OS Considerations**

Current multicore vendors offer systems with up to eight cores on a single chip. With each succeeding processor technology generation, the number of cores and the amount of shared and dedicated cache memory increases, so that we are now entering the era of "many-core" systems.

The design challenge for a many-core multicore system is to efficiently harness the multicore processing power and intelligently manage the substantial on-chip resources efficiently. A central concern is how to match the inherent parallelism of a many-core system with the performance requirements of applications. The potential for **parallelism** in fact exists at three levels in contemporary multicore system.
First, there is **hardware parallelism** within each core processor, known as instruction level parallelism, which may or may not be exploited by application programmers and compilers. Second, there is the potential for **multiprogramming and multithreaded execution** within each processor. Finally, there is the potential for a **single application** to execute in **concurrent** processes or threads across multiple cores.

In essence, then, since the advent of multicore technology, OS designers have been struggling with the problem of how best to extract parallelism from computing workloads.

# Process

## Introduction

A process can be thought of as a program in execution, A process will need certain resources—such as CPU time, memory, files, and I/O devices —to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.

- A process is the unit of work in most systems.
- Systems consist of a collection of processes:
  - o Operating-system processes execute system code, and
  - o user processes execute user code.
- All these processes may execute concurrently. Although traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads.

## Processes

Early computer systems allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, current-day computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a process, **which is a program in executio**n.

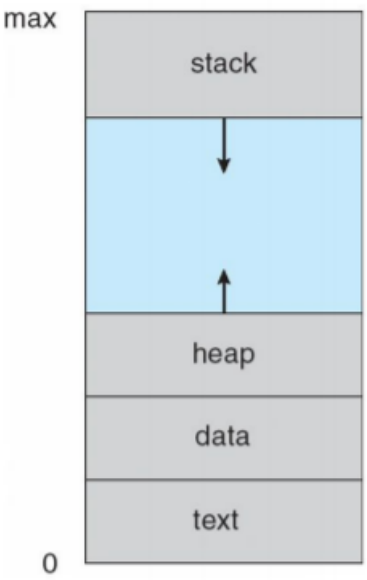
A process is the unit of work in a modern time-sharing system. The more complex the operating system is, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself A system therefore consists of a collection of processes: operating system processes executing system code and user processes executing user code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them.By switching the CPU between processes, the operating system can make the computer more productive.

An operating system executes a variety of programs:

o Batch system – jobs

o Time-shared systems – user programs or tasks

We use the terms job and process almost interchangeably.




Process – a program in execution; process execution must progress in sequential fashion. A process is a program in execution. A process is more than the program code, which is sometimes known as the **text section**. A process also includes: **program counter, program stack, data section.** The process stack contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables. A process may also include a **heap**, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure.

We emphasize that a program by itself is not a process; a program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file). Whereas a **process is an active entity**, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. It is also common to have a process that spawns many processes as it runs.

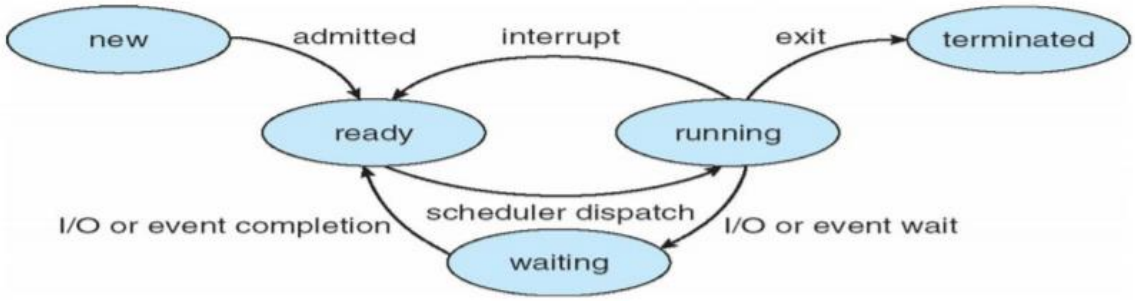
**Process State**

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

o New : The process is being created.

o Running : Instructions are being executed.

o Waiting : The process is waiting for some event to occur (such as an I/O completion or reception of a signal).




o Ready : The process is waiting to be assigned to a processor.

o Terminated : The process has finished execution.

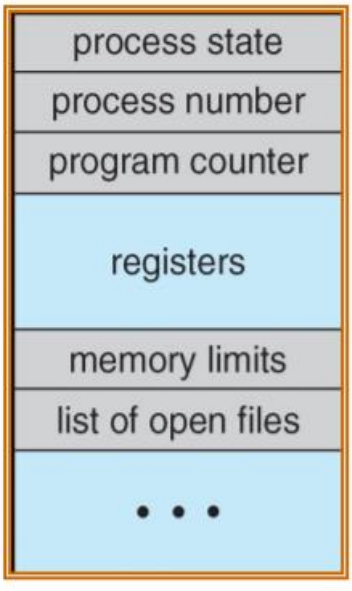
These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however.

**Process Control Block (PCB)**

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. A PCB is shown in Figure.

It contains many pieces of information associated with a specific process: i. Process state ii. Program counter iii. CPU registers iv. CPU scheduling information v. Memory-management information vi. Accounting information vii. I/O status information

- Process state: - The state may be new, ready, running, waiting, halted, and so on.
- Program counter: The counter indicates the address of the next instruction to be executed for this process.
- CPU registers: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
- CPU-scheduling information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters
- Memory-management information: This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.




**Process Control Block (PCB)**

- Accounting information: This information includes the amount of CPU and real time used, time limits, account members, job or process numbers, and so on.
- I/O status information: This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for any information that may vary from process to process

**Context switch**

As we know interrupts cause the operating system to change a CPU from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, **the system needs to save the current context of the process currently running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it**. The context is represented in the PCB of the process; it includes the value of the CPU registers, the process state , and memory-management information. Generically, we perform a state save of the current state of the CPU, be it in kernel or user mode, and then a state restore to resume operations. **Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process is known as a context switch. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching.**

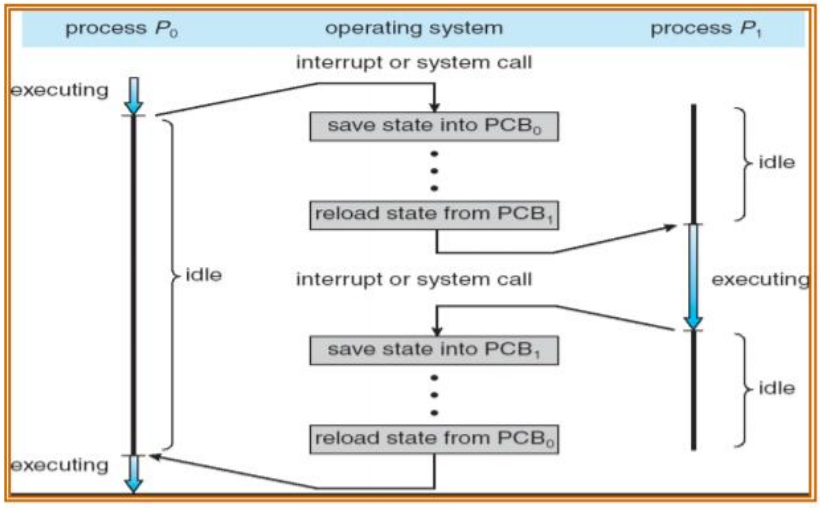
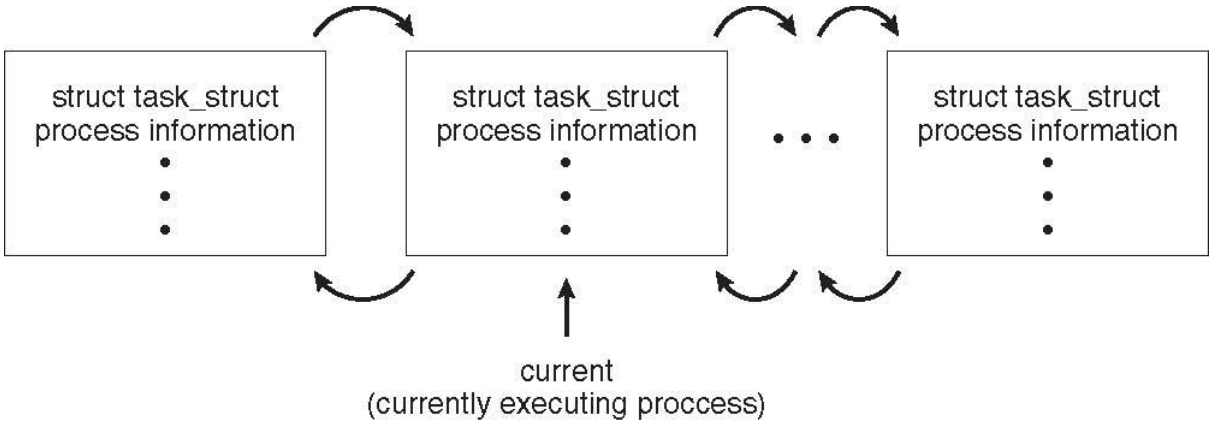



**Diagram showing CPU Switch From Process to Process**

**Process representation in Linux**

Represented by the C structure **task_struct .** This maintains PCB like data structure.

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



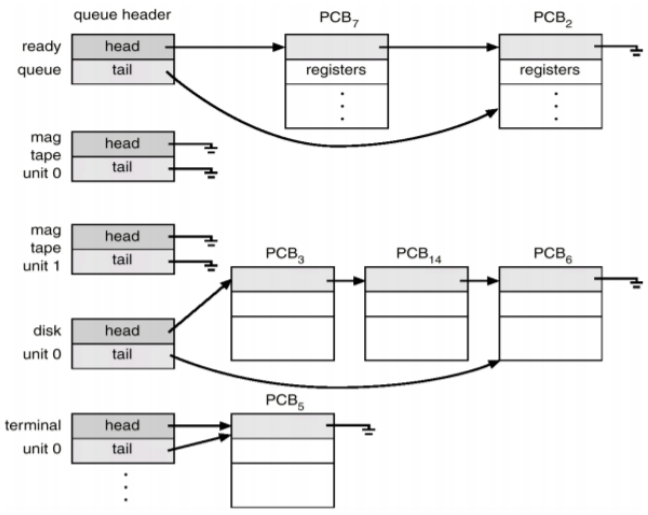

## Process Scheduling

A uniprocessor system can have only one running process. If more process exists, the rest must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.

**Process Scheduling Queues**

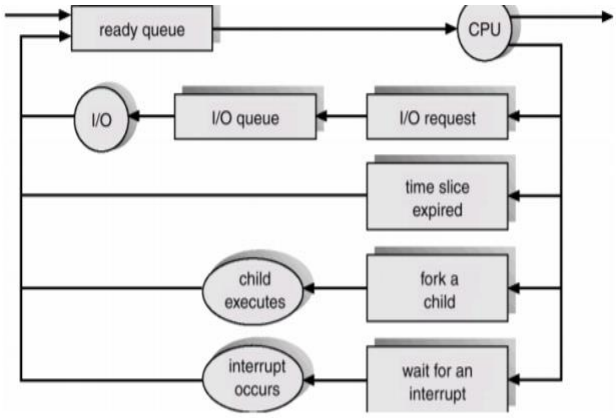
Process migration between the various queues: i. Job queue ii. Ready queue iii. Device queues
As processes enter the system, they are put into a job queue, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list and each PCB includes a pointer field that points to the next PCB in the ready queue.

The Operating system also includes other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a dedicated tape drive, or to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue. A common representation for a discussion of process scheduling is a queuing diagram.




**Ready Queue And Various I/O Device Queues**

Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system. A new process is initially put in the ready queue. It waits there until it is selected for execution, or is dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:




Queuing-diagram representation of Process Scheduling

- o The process could issue an I/O request and then be placed in an I/O queue.
- o The process could create a new subprocess and wait for the subprocess's termination.
- o The process could be removed forcibly from the CPU, as a result of an
- o Interrupt, and be put back in the ready queue.

A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated

## Schedulers

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler. Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution.

o The long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution.

o The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them.

The primary distinction between these two schedulers lies in frequency of execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast.

The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the degree of multiprogramming (the number of processes in memory). Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution. It is important that the long-term scheduler make a careful selection.

In general, most processes can be described as either I/O bound or CPU bound.

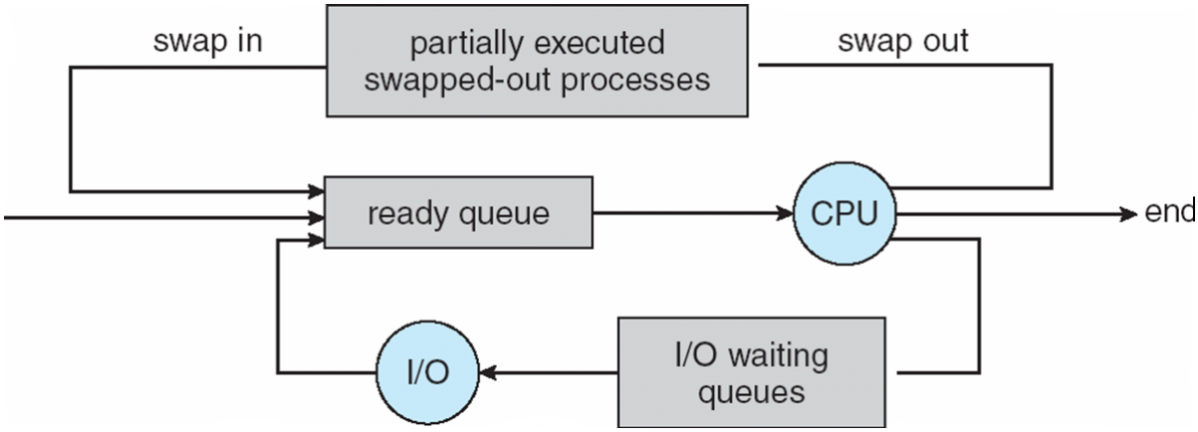
o An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.

o A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.

It is important that the long-term scheduler select a good process mix of I/O-bound and CPU-bound processes.

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. The Medium-term scheduler is diagrammed in Figure. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of

multiprogramming. At sometime later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

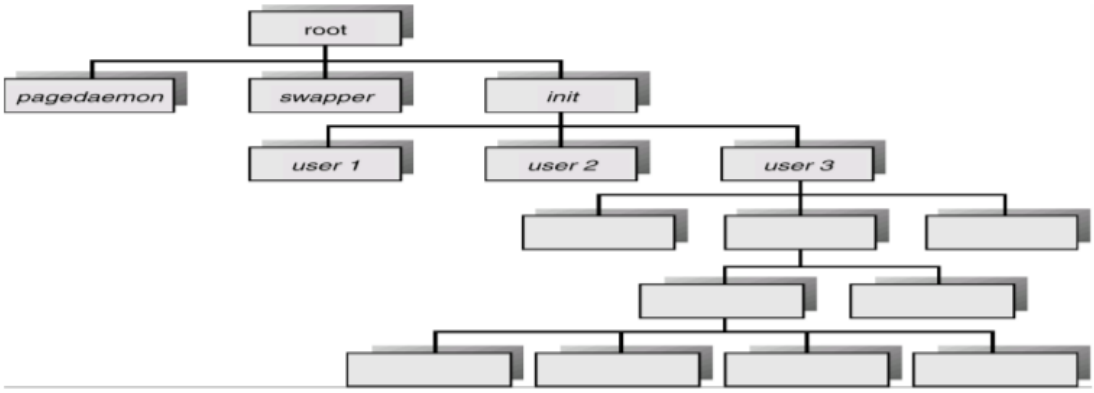



## Operations on Processes :

The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus, the operating system must provide a mechanism (or facility) for process creation and termination

- Process Creation
  Parent process creates children processes, which, in turn create other processes, forming a tree of processes.
  Resource sharing
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.
- Execution
  -Parent and children execute concurrently.
  - Parent waits until children terminate.

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a parent process, and the new processes are called the children of that process.




**A Tree of Processes On A Typical UNIX System**

Each of these new processes may in turn create other processes, forming a tree of processes. In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.

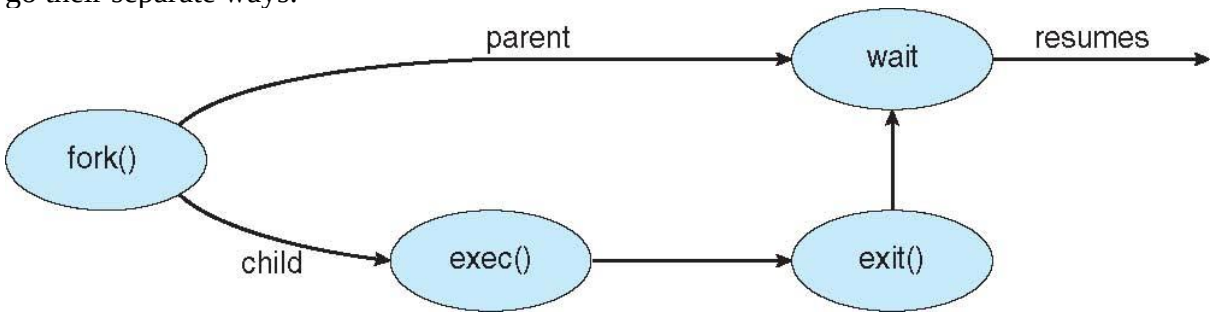
Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses. In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process. For example, consider a process whose function is to display the contents of a file—say, img.jpg—on the screen of a terminal. When it is created, it will get, as an input from its parent process, the name of the file img.jpg, and it will use that file name, open the file, and write the contents out. It may also get the name of the output device.Some operating systems pass resources to child processes. On such a system, the new process may get two open files, img.jpg and the terminal device, and may simply transfer the datum between the two.

When a process creates a new process, two possibilities exist in terms of execution:
o The parent continues to execute concurrently with its children.
o The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:
o The child process is a duplicate of the parent process (it has the same o program and data as the parent).
o The child process has a new program loaded into it.

In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer.
A new process is created by the **fork( )** system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork( ), with one difference. The return code for the fork( ) is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Typically, the **exec( )** system call is used after a fork( ) system call by one of the two processes to replace the process's memory space with a new program. The exec( ) system call loads a binary file into memory (destroying the memory image of the program containing the exec( ) system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways.



The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a **wait( )** system call to move itself off the ready queue until the termination of the child.

To illustrate these differences, let's first consider the UNIX operating system.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

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

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
   }

   return 0;
}
```

The C program illustrates the UNIX system calls previously described. We now have two different processes running a copy of the same program. The value of pid for the child process is zero; that for the parent is an integer value greater than zero. The child process overlays its address space with the UNIX command / b i n / I s (used to get a directory listing) using the execl( ) system call (execlp( ) is a version of the exec() system call). The parent waits for the child process to complet e with the wait () system call. When the child process completes (by either implicitly or explicitly invoking e x i t ( ) ) the parent process resumes from the call to wait (), where it completes using the exit( ) system call.

**Process Termination**

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit( ) system call. At that point, the process may return a status value (typically an integer) to its parent process (via the wait( ) system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

o The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)

o The task assigned to the child is no longer required.

o The parent is exiting, and the operating system does not allow a child to

o continue if its parent terminates.

If a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system

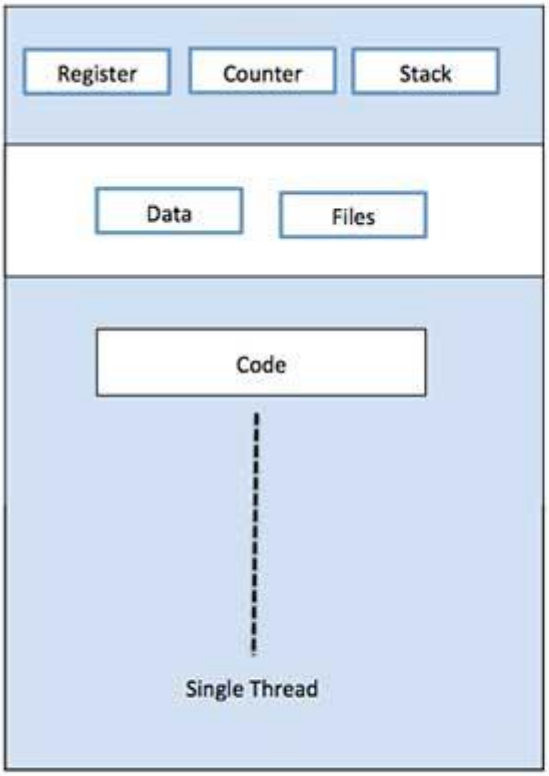
**What is Thread?**

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.

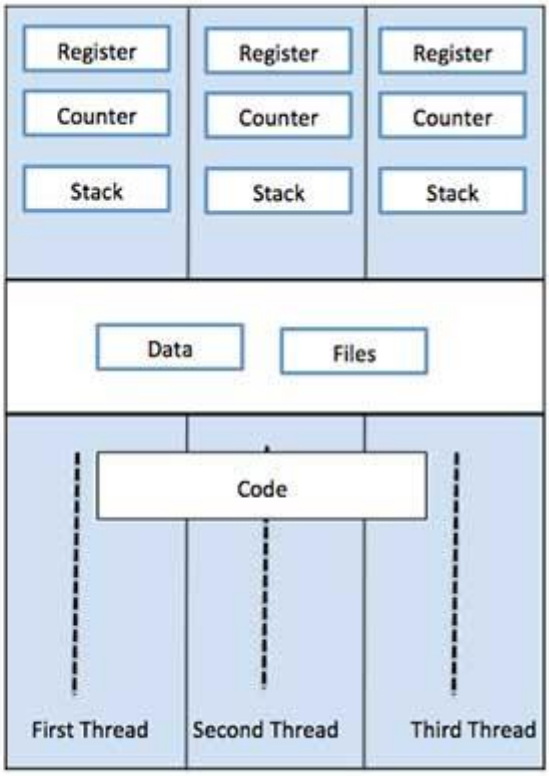
A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

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.

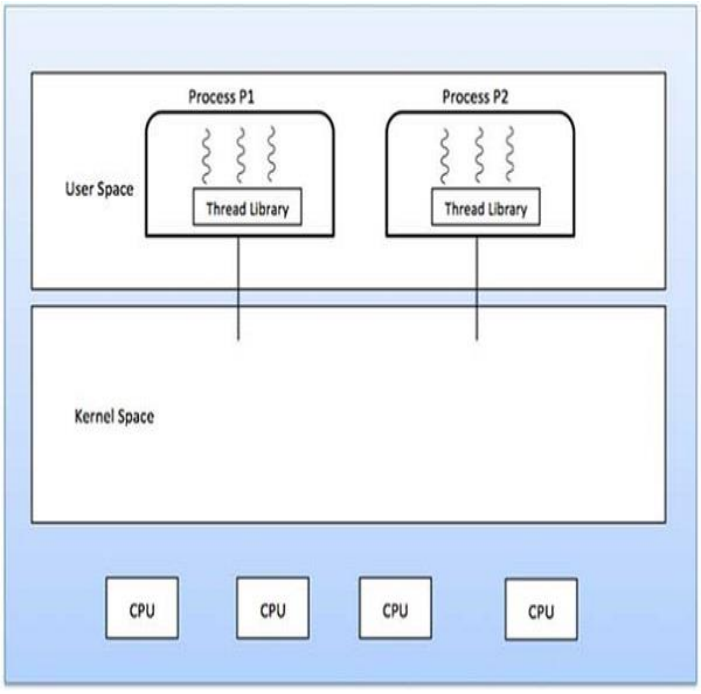
Types of Thread

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.

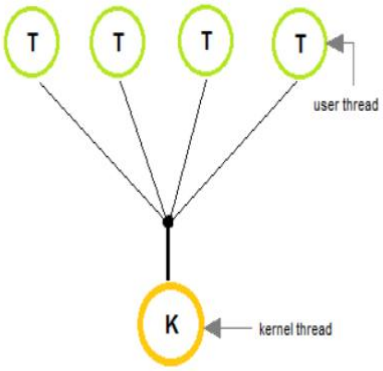
Disadvantages

- 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.



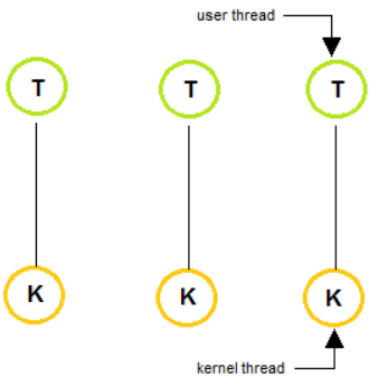

**Many to one Model**

- The many-to-many model multiplexes any number of In the **many to one** model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is efficient in nature.

.

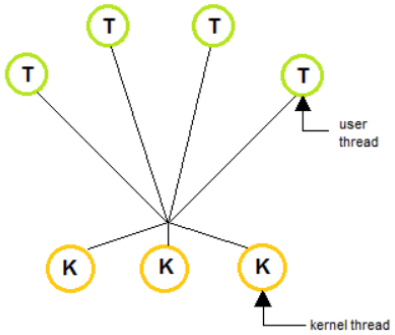
**One to One Model**

- The one to one model creates a separate kernel thread to handle each and every user thread.
- M ost implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.



**Many to Many Model**

- The many to many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users can create any number of the threads.
- Blocking the kernel system calls does not block the entire process.
- Processes can be split across multiple processors




**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. |

**Benefits of Multithreading**

1. Responsiveness
2. Resource sharing, hence allowing better utilization of resources.
3. Economy. Creating and managing threads becomes easier.
4. Scalability. One thread runs on one CPU. In Multithreaded processes, threads can be distributed over a series of processors to scale.
5. Context Switching is smooth. Context switching refers to the procedure followed by CPU to change from one task to another.

**Multithreading Issues**

Below we have mentioned a few issues related to multithreading. Well, it's an old saying, *All good things, come at a price.*

**fork() System Call**

fork() is a system call executed in the kernel through which a process creates a copy of itself. Now the problem in Multithreaded process is, if one thread forks, will the entire process be copied or not? And same applies to exec()

### Thread Cancellation

Thread cancellation means terminating a thread before it has finished working. There can be two approaches for this, one is **Asynchronous cancellation**, which terminates the target thread immediately. The other is **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.

### Signal Handling

Signals are used in UNIX systems to notify a process that a particular event has occurred. Now in when a Multithreaded process receives a signal, to which thread it must be delivered? It can be delivered to all, or a single thread.

### Security Issues

Yes, there can be security issues because of extensive sharing of resources between multiple threads.

# Inter Process Communication (IPC)

A process can be of two type:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilised for increasing computational speed, convenience and modularity. **Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them.** Processes can communicate with each other using these two ways:

1. Shared Memory
2. Message passing

**i.Shared memory model.**

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.

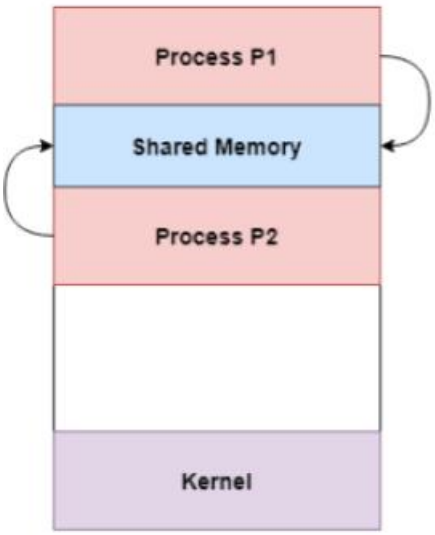
Advantage of Shared Memory Model

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.




Shared Memory Model

**Ex: Producer-Consumer problem :**

There are two processes: Producer and Consumer. Producer produces some item and Consumer consumes that item. The two processes shares a common space or memory location known as buffer where the item produced by Producer is stored and from where the Consumer consumes the item if needed. There are two version of this problem: first one is known as unbounded buffer problem in which Producer can keep on producing items and there is no limit on size of buffer, the second one is known as bounded buffer problem in which producer can produce up to a certain amount of item and after that it starts waiting for consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, then producer will start producing items. If the total produced item is equal to the size of buffer, producer will wait to get it consumed by the Consumer. Sim-ilarly, the consumer first check for the availability of the item and if no item

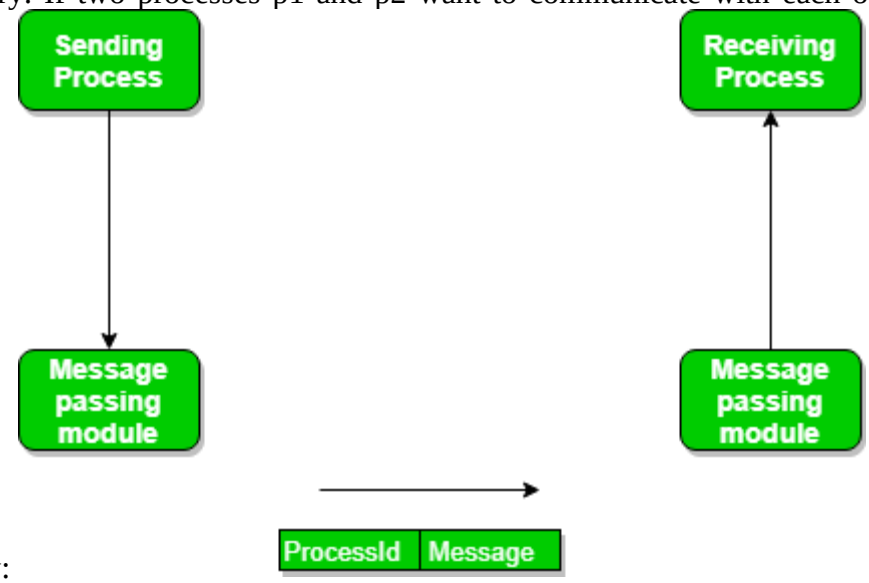
is available, Consumer will wait for producer to produce it. If there are items available, consumer will consume it. The pseudo code are given below:

| Producer process | Consumer Process |
|---|---|
| item nextProduced;<br><br>while(1){<br>/ / check if there is no space for production.<br>// if so keep waiting.<br>while((free_index+1) mod buff_max == full_index);<br>shared_buff[free_index] = nextProduced;<br>free_index = (free_index + 1) mod buff_max;<br>} | item nextConsumed;<br><br>while(1){<br><br>// check if there is an available<br>// item for consumption.<br>// if not keep on waiting for<br>// get them produced.<br>while((free_index == full_index);<br>nextConsumed = shared_buff[full_index];<br>full_index = (full_index + 1) mod buff_max;<br>} |

In the above code, The producer will start producing again when the (free_index+1) mod buff max will be free because if it it not free, this implies that there are still items that can be consumed by the Consumer so there is no need to produce more. Similarly, if free index and full index points to the same index, this implies that there are no item to consume.

**ii) Messaging Passing Method**

Now, We will start our discussion for the communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:




- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives. We need at least two primitives:
  – **send**(message, destinaion) or **send**(message)
  – **receive**(message, host) or **receive**(message)

a. **Message Passing through Communication Link.**

**Direct and Indirect Communication link**

Now, We will start our discussion about the methods of implementing communication link. While implementing the link, there are some questions which need to be kept in mind like :

1. How are links established?
2. Can a link be associated with more than two processes?
3. How many links can there be between every pair of communicating processes?
4. What is the capacity of a link? Is the size of a message that the link can accommodate fixed or variable?
5. Is a link unidirectional or bi-directional?

A link has some capacity that determines the number of messages that can reside in it temporarily for which Every link has a queue associated with it which can be either of zero capacity or of bounded capacity or of unbounded capacity. In zero capacity, sender wait until receiver inform sender that it has received the message. In non-zero capacity cases, a process does not know whether a message has been received or not after the send operation. For this, the sender must communicate to receiver explicitly. Implementation of the link depends on the situation, it can be either a Direct communication link or an In-directed communication link. **Direct Communication links** are implemented when the processes use specific process identifier for the communication but it is hard to identify the sender ahead of time. **For example: the print server.**

In Direct message passing, The process which want to communicate must explicitly name the recipient or sender of communication.

e.g. send(p1, message) means send the message to p1. similarly, receive(p2, message) means receive the message from p2. In this method of communication, the communication link get established automatically, which can be either unidirectional or bidirectional, but one link can be used between one pair of the sender and receiver and one pair of sender and receiver should not possess more than one pair of link. Symmetry and asymmetry between the sending and receiving can also be implemented i.e. either both process will name each other for sending and receiving the messages or only sender will name receiver for sending the message and there is no need for receiver for naming the sender for receiving the message.The problem with this method of communication is that if the name of one process changes, this method will not work.

**In-directed Communication** is done via a shred mailbox (port), which consists of queue of messages. Sender keeps the message in mailbox and receiver picks them up. **In Indirect message passing**, processes uses mailboxes (also referred to as ports) for sending and receiving messages. Each mailbox has a unique id and processes can communicate only if they share a mailbox. Link established only if processes share a common mailbox and a single link can be associated with many processes. Each pair of processes can share several communication links and these link may be unidirectional or bi-directional. Suppose two process want to communicate though Indirect message passing, the required operations are: create a mail box, use this mail box for sending and receiving messages, destroy the mail box. The standard primitives used are : **send(A,**

**message)** which means send the message to mailbox A. The primitive for the receiving the message also works in the same way e.g. **received (A, message)**.

b. **Message Passing through Exchanging the Messages.**

**Synchronous and Asynchronous Message Passing:**

A process that is blocked is one that is waiting for some event, such as a resource becoming available or the completion of an I/O operation. IPC is possible between the processes on same computer as well as on the processes running on different computer i.e. in networked/distributed system. In both cases, the process may or may not be blocked while sending a message or attempting to receive a message so Message passing may be blocking or non-blocking. Blocking is considered **synchronous** and **blocking send** means the sender will be blocked until the message is received by receiver. Similarly, **blocking receive** has the receiver block until a message is available. Non-blocking is considered **asynchronous** and Non-blocking send has the sender sends the message and continue. Similarly, Non-blocking receive has the receiver receive a valid message or null. After a careful analysis, we can come to a conclusion that, for a sender it is more natural to be non-blocking after message passing as there may be a need to send the message to different processes But the sender expect acknowledgement from receiver in case the send fails. Similarly, it is more natural for a receiver to be blocking after issuing the receive as the information from the received message may be used for further execution but at the same time, if the message send keep on failing, receiver will have to wait for indefinitely. That is why we also consider the other possibility of message passing. There are basically three most preferred combinations:

- Blocking send and blocking receive
- Non-blocking send and Non-blocking receive
- Non-blocking send and Blocking receive (Mostly used)

The other message passing variants include :

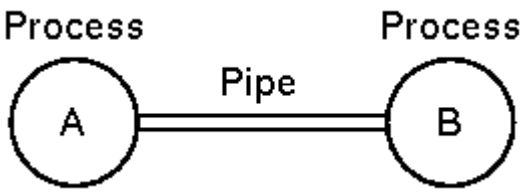
- Send by copy / Send by reference

Buffering - queue of messages attached to the link; implemented in one of three ways.

- Zero capacity - 0 messages Sender must wait for receiver (rendezvous).
- Bounded capacity - finite length of n messages Sender must wait if link full.
- Unbounded capacity - infinite length Sender never waits.

**Pipes :**

A pipe is a simple method for communicating between two processes.




- As far as the processes are concerned the pipe appears to be just like a file.
- When A performs a write, it is buffered in the pipe.

- When B reads then it reads from the pipe, blocking if there is no input.
- in UNIX (and DOS) one process can be piped into another pipe using the '|' character. e.g.

  **cat classlist | sort | more**

  the **cat** command prints the contents of the file 'classlist', this is piped into the **sort** command which sorts the list. Finally, the sorted list is sent to the **more** command that prints it one screenfull at a time.

- Pipes may be implemented using shared memory (UNIX) or even with temporary files (DOS).

# pipe() System call

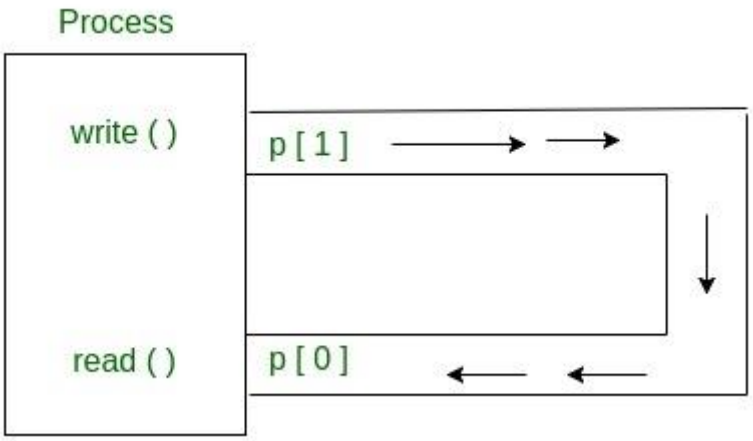
Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process. In UNIX Operating System, Pipes are useful for communication between related processes(inter-process communication).

- Pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe.
- The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this "virtual file" or pipe and another related process can read from it.
- If a process tries to read before something is written to the pipe, the process is suspended until something is written.
- The pipe system call finds the first two available positions in the process's open file table and allocates them for the read and write ends of the pipe.

Syntax

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

Parameters :
fd[0] will be the fd(file
descriptor) for the
read end of pipe.
fd[1] will be the fd for the
write end of pipe.
Returns : 0 on Success.
-1 on error.
```




// C program to illustrate pipe system call in C

```
#include <stdio.h>
#include <unistd.h>
#define MSGSIZE 16
```

```
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";

int main()
{
        char inbuf[MSGSIZE];
        int p[2], i;

        if (pipe(p) < 0)
                exit(1);

        /* continued */
        /* write pipe */

        write(p[1], msg1, MSGSIZE);
        write(p[1], msg2, MSGSIZE);
        write(p[1], msg3, MSGSIZE);

        for (i = 0; i < 3; i++) {
                /* read pipe */
                read(p[0], inbuf, MSGSIZE);
                printf("% s\n", inbuf);
        }
        return 0;
}
```