1. **Introduction to Operating Systems**
2. **What is Operating System?**

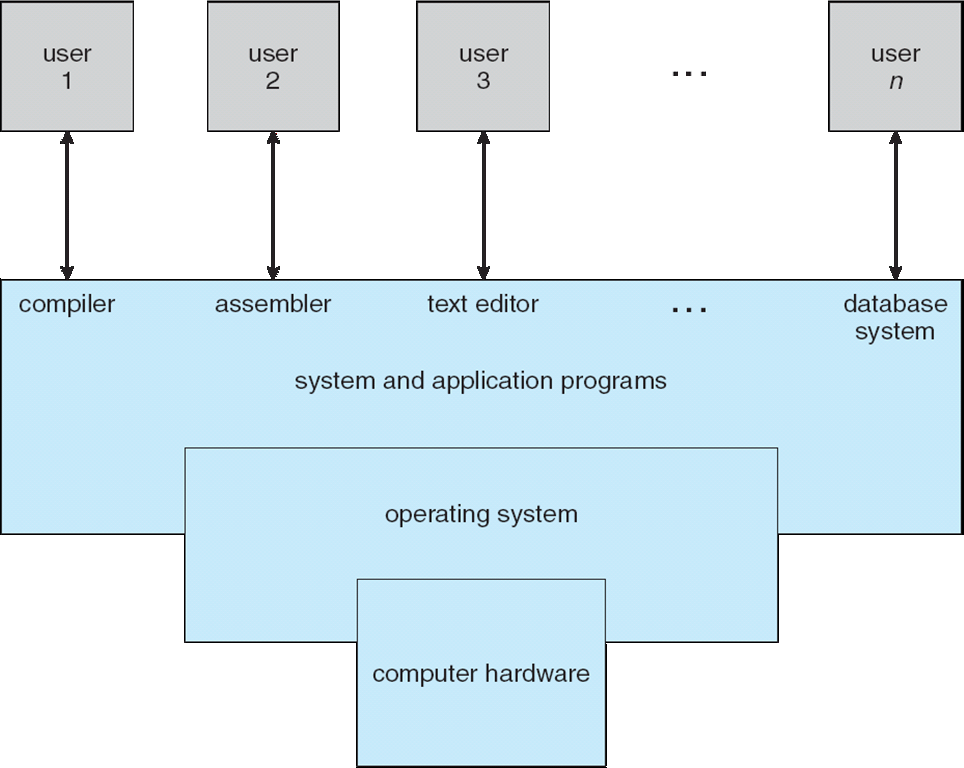
Operating system is a type of system software between a user of a computer and the computer hardware having its own goals. The goals of operating system can be generalized as:

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

Computer system has mainly the following four components:

* + Hardware- provides basic computing resources such as CPU, memory, I/O devices etc.
  + Operating SystemControls and coordinates use of hardware among various applications and users
  + Application Program – define the ways in which the system resources are used to solve the computing problems of the users. Word processors, compilers, web browsers, database systems, video games are examples of applications program.
  + Users of a computer system include people, machines, and other computers

The figure bellow illustrates the layered structure of the components:



1. **Types of OS**

A number of types of operating systems have been developed in the different generations of computers. One distinction is on the basis of how they accomplish their work and accordingly we can have the following types:

**Serial Processing**:

The Serial Processing Operating Systems are those which perform all the instructions into a Sequence Manner or the Instructions those are given by the user will be executed by using the FIFO Manner.

Mainly those Operating systems are used to work with Punched Cards. In this, all the Jobs are firstly prepared and stored on the card and after that the card will be entered in the System and then all the Instructions will be executed one by One.

**Batch Processing:**

The Batch Processing is same as the serial processing technique except that in the Batch Processing system similar types of jobs are prepared and stored on the card and submitted to the system to get executed.

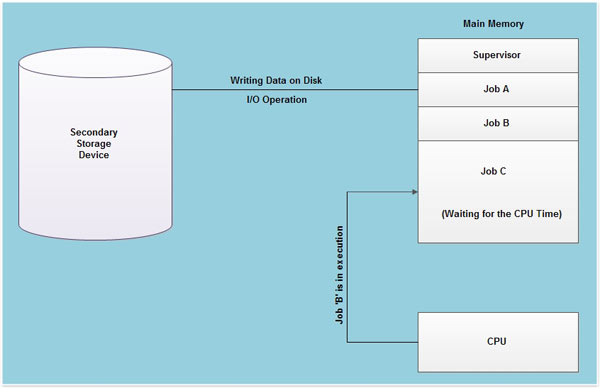
It is the grouping together of several processing jobs to be executed one after another by a [computer](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-computer), without any user interaction. This is achieved by placing a list of the commands to start the required jobs into a BATCH FILE that can be executed as if it were a single program: hence batch processing is most often used in operating systems that have a COMMAND LINE user interface. Indeed, batch processing was the normal mode of working in the early days of mainframe [computers](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-computer), but modern [personal computer](http://ecomputernotes.com/fundamental/introduction-to-computer/personal-computer) applications typically require frequent user interaction, making them unsuitable for batch execution.

Running a batch file is one example of batch processing, but there are plenty of others. When you select several documents from the same application and print them all in one step (if the application allows you to do that), you are "batch printing," which is a form of batch processing. Or let's say that you want to send a whole group of files to someone else via your modem-if your communications software permits batch processing, you can choose all the files you want to send, and have the software send them off in a batch. Batch processing is a good feature to have in most applications.

**Multi-Programming**

To overcome the problem of underutilization of [CPU](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-cpu) and main [memory](http://ecomputernotes.com/fundamental/input-output-and-memory/what-are-the-different-types-of-ram-explain-in-detail), the multiprogramming operating system was introduced. The multiprogramming is interleaved execution of multiple jobs by the same [computer](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-computer).

In multiprogramming system, when one program is waiting for I/O transfer; there is another program ready to utilize the CPU. So it is possible for several jobs to share the time of the CPU. But it is important to note that multiprogramming is not defined to be the execution of jobs at the same instance of time. Rather it does mean that there are a number of jobs available to the CPU (placed in main memory) and a portion of one is executed then a segment of another and so on.

A simple process of multiprogramming is shown in the figure bellow.

As shown in fig, at the particular situation, job 'A' is not utilizing the CPU time because it is busy in I/O operations. Hence the CPU becomes busy to execute the job 'B'. Another job C is waiting for the CPU for getting its execution time. So in this state the CPU will never be idle and utilizes maximum of its time.

A program in execution is called a "Process", "Job" or a "Task". The concurrent execution of programs improves the utilization of system resources and enhances the system throughput as compared to batch and serial processing. In this system, when a process request some I/O to allocate, meanwhile the CPU time is assigned to another ready process. So, here when a process is switched to an I/O operation, the CPU is not set idle.

Multiprogramming is a common approach to resource management. The essential components of a single-user [operating system](http://ecomputernotes.com/fundamental/disk-operating-system/what-is-operating-system) include a command processor, an input/ output control system, a file system, and a transient area. A multiprogramming operating system builds on this base, subdividing the transient area to hold several independent programs and adding resource management routines to the operating system's basic functions.

**Multitasking Operating System:**

Multitasking is a logical extension of multiprogramming system that supports multiple programs to run concurrently. In multitasking more than one task are executed at the same time. In this technique the multiple tasks, also known as processes, share common processing resources such as a [CPU](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-cpu). In the case of a [computer](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-computer) with single CPU, only one job can be processed at a time. Multitasking solves the problem by scheduling and deciding which task should be the running task and when a waiting task should get turn. This attempt is done with the help of [interrupt](http://ecomputernotes.com/fundamental/input-output-and-memory/what-is-interrupt-types-of-interrupts) (a signal) which is attended by CPU by keeping the current activity aside, saves its present status in buffer and returns to another important job whatever task it was doing earlier. The act of re-assigning a CPU from one task to another one is known as *context switch*.

The multitasking systems were developed to provide interactive use of a computer system. This system uses the CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Thus, multitasking makes the best possible use of available hardware at any given instance of time and improves the overall efficiency of computer system.

A multitasking operating system is characterized by its capability to support the concurrent execution of more than one task. This is achieved by simultaneous management of several processes in the main [memory](http://ecomputernotes.com/fundamental/input-output-and-memory/what-are-the-different-types-of-ram-explain-in-detail) at the same time and by availing I/O resources amongst the active tasks. The multi-tasking OS monitors the state of all the tasks and of the system resources.

Multitasking provides the fundamental mechanism for an application to control and react to multiple, discrete real-world events and is therefore essential for many real-time applications. A multitasking environment allows applications to be constructed as a set of independent tasks, each with a separate thread of execution and its own set of system resources. The inter-task communication facilities allow these tasks to synchronize and coordinate their activity. Multitasking creates the appearance of many threads of execution running concurrently when, in fact, the kernel interleaves their execution on the basis of a scheduling algorithm. This also leads to efficient utilization of the CPU time and is essential for many embedded applications where processors are limited in computing speed due to cost, power and other constraints.

In a multitasking operating system, it is assumed that the various tasks are to cooperate in communicating with each other and share common data in an orderly and disciplined manner, without creating the contention and deadlocks.

Actually, even in true multitasking, only one application is ever running at anyone instant. But because the computer automatically switches from one to the next so quickly, all the programs seem to run simultaneously. (With context switching, you decide when to shift from one program to another, by hitting a key or clicking the mouse.) Each individual program runs slower, of course, since each gets only a portion of the computer's time, and since some time is lost in the process of going from program to program.

Ideally, multi-tasking capability is built into your computer's operating system. DOS has absolutely no multi-tasking features, but Windows provides it somehow.

**Comparison of Multiprogramming, Multitasking, and Multithreading:**

***Multiprogramming*:** Multiprogramming is the technique of running several programs at a time using timesharing. It allows a [computer](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-computer) to do several things at the same time. Multiprogramming creates logical parallelism. The concept of multiprogramming is that the [operating system](http://ecomputernotes.com/fundamental/disk-operating-system/what-is-operating-system) keeps several jobs in [memory](http://ecomputernotes.com/fundamental/input-output-and-memory/what-are-the-different-types-of-ram-explain-in-detail) simultaneously.

The operating system selects a job from the job pool and starts executing a job, when that job needs to wait for any i/o operations the [CPU](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-cpu) is switched to another job. So the main idea here is that the CPU is never idle.

***Multitasking***: Multitasking is the logical extension of multiprogramming. The concept of multitasking is quite similar to multiprogramming but the difference is that the switching between jobs occurs so frequently that the users can interact with each program while it is running. This concept is also known as time-sharing systems. A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of time-shared system.

For example, let us say you are printing a document of 100 pages. While your computer is performing that, you still can-do other jobs like typing a new document. So, more than one task is performed.

One of the main differences between multiprogramming and multitasking is, "In multiprogramming, a user cannot interact (everything is decided by the OS, like picking the next program and sharing on time basis, etc...); where as in multitasking, a user can interact with the system (you can type a letter, while the other task of printing is going on)".

***Multithreading***: An application typically is implemented as a separate process with several threads of control. In some situations, a single application may be required to perform several similar tasks for example a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several of clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time.

The amount of time that a client might have to wait for its request to be serviced could be enormous. So, it is efficient to have one process that contains multiple threads to serve the same purpose. This approach would multithread the web-server process. The server would create a separate thread that would listen for client requests. When a request was made rather than creating another process it would create another thread to service the request. To get the advantages like responsiveness, resource sharing economy and utilization of multiprocessor architectures multithreading concept can be used.

This type of programming helps when more than one client uses it. For example, let us take DB. While I'm entering records in the DB, there would be someone else, doing the same type of job. If DB is not having a multithread option, then not more than one person will be able to do the same job.

**Multiuser Operating Systems:**

A multi-user system is able to perform so (pseudo-) and competing separate applications belonging to several users.

Competing means that applications can be active at the same time and has access to different resources such as [CPU](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-cpu), memory, and hard drives. Independent means that each application can perform their work without worrying what are the applications of other users.

A multiuser system is necessarily multitasking, but the converse is false: the MS-DOS operating system is single-user and single-task; MacOS 6.1 and Windows 3.1 is single user but multitasking; UNIX and Windows NT are multiuser.

When they allow multi-use, the operating system must provide a number of mechanisms including:

* An authentication mechanism for verifying the identity of the user;
* A protection mechanism against erroneous user programs that could block other applications running on the system, or malicious, which could disrupt or spy on the activities of other users;
* An accounting mechanism to limit the amount of resources allocated to each user.

In a multi-user system, each user has a private area on the machine: generally, it has a certain quota of disk space to store its files, receives private e-mails, etc. The operating system must ensure that the private party space of a user can not be visible to the owner. The operating system, in particular, ensures that no user can use an application of the system for the purpose of violating the private area of ​​another user.

Each user is identified by a unique number, called the ID of the user, or UID (User Identifier). In general, only a limited number of persons are allowed to use a computer system. When one of these users starts a work session, the operating system asks for a user name and password. If the user does not respond with valid information, access is denied.

To selectively share materials with others, each user can be member of one or multiple user groups. A group is also identified by a unique number called the group ID (GID for Group Identifier). For example, each file is associated with one and only one group.

A multi-user operating system provides a special user called root or supervisor. The system administrator must log as super user to manage user accounts and perform maintenance tasks such as backups and program updates. The super user can do almost anything in that the operating system never applies the protection mechanisms they do on other users, called ordinary users. The root can, in particular, access all files in the system and interfere with the activity of any running process running. It cannot, however, access to the input-output ports that have not been provided by the kernel.

**Real Time System (Hard/Soft Real Time System)**

Real time [Operating Systems](http://ecomputernotes.com/fundamental/disk-operating-system/what-is-operating-system) are very fast and quick respondent systems. These systems are used in an environment where a large number of events (generally external) must be accepted and processed in a short time. Real time processing requires quick transaction and characterized by supplying immediate response.

For example, a measurement from a petroleum refinery indicating that temperature is getting too high and might demand for immediate attention to avoid an explosion.

In real time [operating system](http://ecomputernotes.com/fundamental/disk-operating-system/what-is-operating-system), Response Time is already fixed; means the time to display the results after possessing is fixed by the [processor](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-cpu), or [CPU](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-cpu).

In real time [operating system](http://ecomputernotes.com/fundamental/disk-operating-system/what-is-operating-system) there is a little swapping of programs between primary and secondary [memory](http://ecomputernotes.com/fundamental/input-output-and-memory/what-are-the-different-types-of-ram-explain-in-detail). Most of the time, processes remain in primary memory in order to provide quick response, therefore, memory management in real time system is less demanding compared to other systems.

The primary functions of the real time operating system are to:

* Manage the processor and other system resources to meet the requirements of an application.
* Synchronize with and respond to the system events.
* Move the data efficiently among processes and to perform coordination among these processes.

The Real Time systems are used in the environments where a large number of events (generally external to the [computer](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-computer) system) is required to be accepted and is to be processed in the form of quick response. Such systems have to be the multitasking. So the primary function of the real time operating system is to manage certain system resources, such as the [CPU](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-cpu), memory, and time. Each resource must be shared among the competing processes to accomplish the overall function of the system. Apart from these primary functions of the real time operating system there are certain secondary functions that are not mandatory but are included to enhance the performance:

The term real time refers to the technique of updating files with the transaction data immediately just after the event that it relates with.

Few more examples of real time processing are:

* Airlines reservation system.
* Air traffic control system.
* Systems that provide up to the minute [information](http://ecomputernotes.com/fundamental/information-technology/what-do-you-mean-by-data-and-information) on stock prices.
* Defense application systems like as RADAR.

Real time operating systems mostly use the preemptive priority scheduling. These support more than one scheduling policy and often allow the user to set parameters associated with such policies, such as the time-slice in Round Robin scheduling where each task in the task queue is scheduled up to a maximum time, set by the time-slice parameter, in a round robin manner. Hundreds of the priority levels are commonly available for scheduling. Some specific tasks can also be indicated to be non-preemptive.

There are two types of real time systems: *hard real time and soft real time systems*.

A Hard Real-Time System guarantees that critical tasks complete on time. This goal requires that all delays in the system be bounded from the retrieval of the stored data to the time that it takes the [operating system](http://ecomputernotes.com/fundamental/disk-operating-system/what-is-operating-system) to finish any request made of it.

A soft real time system is where a critical real-time task gets priority over other tasks and retains that priority until it completes. As in hard real time systems kernel delays need to be bounded.

**Distributed Operating System**

Distributed [Operating System](http://ecomputernotes.com/fundamental/disk-operating-system/what-is-operating-system) is a model where distributed applications are running on multiple [computers](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-computer) linked by communications. A distributed [operating system](http://ecomputernotes.com/fundamental/disk-operating-system/what-is-operating-system) is an extension of the network operating system that supports higher levels of communication and integration of the machines on the network.

This system looks to its users like an ordinary centralized operating system but runs on multiple, independent [central processing units](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-cpu) (CPUs) through a network.

The Distributed OS involves a collection of autonomous [computer](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-computer) systems, capable of communicating and cooperating with each other through a LAN / WAN. A Distributed OS provides a virtual machine abstraction to its users and wide sharing of resources like computational capacity, I/O and files etc.

In a distributed OS, it is the software, not the hardware, that determines whether the system is distributed or not.

The users of a true distributed system should not know, on which machine their programs are running and where their files are stored.

**Examples of distributed operating systems:**

* Solaris operating system for SUN multiprocessor workstations.
* Mac/OS is a multithreading and multitasking UNIX compatible operating system;

Distributed systems provide the following advantages:

* Sharing of resources
* Reliability
* Communication
* Computation speedup

Distributed systems are potentially more reliable than a central system because if a system has only one instance of some critical component, such as a CPU, disk, or network interface, and that component fails, the system will go down. When there are multiple instances, the system may be able to continue in spite of occasional failures. In addition to hardware failures, one can also consider software failures. Distributed systems allow both hardware and software errors to be dealt with.

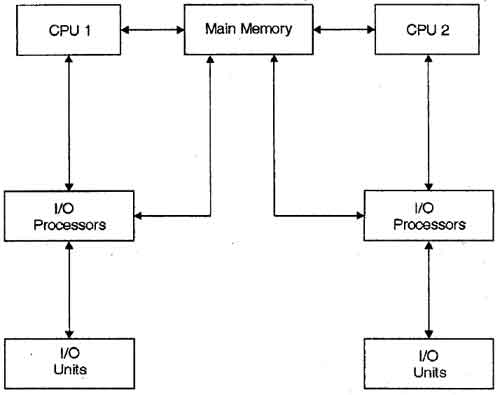
A distributed system is a set of computers that communicate and collaborate each other using software and hardware interconnecting components.

A distributed system is managed by a distributed operating system. A distributed operating system manages the system shared resources used by multiple processes, the process scheduling activity (how processes are allocating on available processors), the communication and synchronization between running processes and so on. The software for parallel computers could be also tightly coupled or loosely coupled. The loosely coupled software allows computers and users of a distributed system to be independent each other but having a limited possibility to cooperate. An example of such a system is a group of computers connected through a local network. Every computer has its own memory, hard disk. There are some shared resources such as files and printers. If the interconnection network broke down individual computers could be used but without some features like printing to a non-local [printer](http://ecomputernotes.com/fundamental/input-output-and-memory/what-is-a-printer-and-what-are-the-different-types-of-printers).

**Multiprocessor** [**Operating System**](http://ecomputernotes.com/fundamental/disk-operating-system/what-is-operating-system)**:**

Multiprocessor OS refers to the use of two or more [central processing units](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-cpu) ([CPU](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-cpu)) within a single [computer](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-computer) system. These multiple CPUs are in a close communication sharing the computer bus, [memory](http://ecomputernotes.com/fundamental/input-output-and-memory/what-are-the-different-types-of-ram-explain-in-detail) and other peripheral devices. These systems are referred as tightly coupled systems.

These types of systems are used when very high speed is required to process a large volume of data. These systems are generally used in environment like satellite control, weather forecasting etc. The basic organization of multiprocessing system is shown in the following figure.



Multiprocessing system is based on the symmetric multiprocessing model, in which each processor runs an identical copy of [operating system](http://ecomputernotes.com/fundamental/disk-operating-system/what-is-operating-system) and these copies communicate with each other. In this system processor is assigned a specific task. A master processor controls the system. This scheme defines a master-slave relationship. These systems can save money in compare to single processor systems because the processors can share peripherals, power supplies and other devices. The main advantage of multiprocessor system is to get more work done in a shorter period of time. Moreover, multiprocessor systems prove more reliable in the situations of failure of one processor. In this situation, the system with multiprocessor will not halt the system; it will only slow it down.

In order to employ multiprocessing operating system effectively, the computer system must have the following:

***Motherboard Support***: A motherboard capable of handling multiple processors. This means additional sockets or slots for the extra chips and a chipset capable of handling the multiprocessing arrangement.

[***Processor***](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-cpu) ***Support***: processors those are capable of being used in a multiprocessing system.

The whole task of multiprocessing is managed by the operating system, which allocates different tasks to be performed by the various processors in the system.

Applications designed for the use in multiprocessing are said to be threaded, which means that they are broken into smaller routines that can be run independently. This allows the operating system to let these threads run on more than one processor simultaneously, which is multiprocessing that results in improved performance.

Multiprocessor system supports the processes to run in parallel. Parallel processing is the ability of the CPU to simultaneously process incoming jobs. This becomes most important in computer system, as the CPU divides and conquers the jobs. Generally, the parallel processing is used in the fields like artificial intelligence and expert system, image processing, weather forecasting etc.

In a multiprocessor system, the dynamically sharing of resources among the various processors may cause therefore, a potential bottleneck. There are three main sources of contention that can be found in a multiprocessor operating system:

***Locking system:*** In order to provide safe access to the resources shared among multiple processors, they need to be protected by locking scheme. The purpose of a locking is to serialize accesses to the protected resource by multiple processors. Undisciplined use of locking can severely degrade the performance of system. This form of contention can be reduced by using locking scheme, avoiding long critical sections, replacing locks with lock-free algorithms, or, whenever possible, avoiding sharing altogether.

***Shared data***: The continuous accesses to the shared data items by multiple processors (with one or more of them with data write) are serialized by the cache coherence protocol. Even in a moderate-scale system, serialization delays can have significant impact on the system performance. In addition, bursts of cache coherence traffic saturate the memory bus or the interconnection network, which also slows down the entire system. This form of contention can be eliminated by either avoiding sharing or, when this is not possible, by using replication techniques to reduce the rate of write accesses to the shared data.

***False sharing:*** This form of contention arises when unrelated data items used by different processors are located next to each other in the memory and, therefore, share a single cache line: The effect of false sharing is the same as that of regular sharing bouncing of the cache line among several processors. Fortunately, once it is identified, false sharing can be easily eliminated by setting the memory layout of non-shared data.

Apart from eliminating bottlenecks in the system, a multiprocessor operating system developer should provide support for efficiently running user applications on the multiprocessor. Some of the aspects of such support include mechanisms for task placement and migration across processors, physical memory placement insuring most of the memory pages used by an application is located in the local memory, and scalable multiprocessor synchronization primitives.

**Parallel operating systems/processing**

Parallel processing systems are designed to speed up the execution of programs by dividing the program into multiple fragments and processing these fragments simultaneously. Such systems are multiprocessor systems also known as tightly coupled systems. Parallel systems deal with the simultaneous use of multiple [computer](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-computer) resources that can include a single computer with multiple processors, a number of [computers](http://ecomputernotes.com/fundamental/introduction-to-computer/what-is-computer) connected by a network to form a parallel processing cluster or a combination of both.

Parallel computing is an evolution of serial computing where the jobs are broken into discrete parts that can be executed concurrently. Each part is further broken down to a series of instructions. Instructions from each part execute simultaneously on different CPUs.

Parallel systems are more difficult to program than computers with a single processor because the architecture of parallel computers varies accordingly and the processes of multiple CPUs must be coordinated and synchronized. Several models for connecting processors and [memory](http://ecomputernotes.com/fundamental/input-output-and-memory/what-are-the-different-types-of-ram-explain-in-detail) modules exist, and each topology requires a different programming model. The three models that are most commonly used in building parallel computers include synchronous processors each with its own memory, asynchronous processors each with its own memory and asynchronous processors with a common, shared memory.

1. **Computer Hardware**

Computer hardware is the physical or visible component that you can see and touch. It may be categorized as main and optional. Main components are the components that are compulsory to the computer system. Without these main components the computer can’t perform any type of task. Examples of these main components can be CPU, RAM, ROM, Power supply, Hard disk, System board / motherboard etc.

The second category of hardware components, i.e. optional components, contains components that add additional features/functionalities to the computer system. Most of such components are peripheral components that serve as input or output of data. An input device is a device that is used to feed data, or instruction, to the computer system; and an output device is a device that is used to retrieve/take data from the computer system. Data can be in the form of text, picture, audio, or video. Examples can be keyboard, monitor/screen/display, expansion cards, mouse, printer, scanner, digital camera, plotter, joystick, flash disk, compact disk (CD), floppy disk, DVD, speaker, microphone, light pen, Web cam etc.

**Input devices:**

* A computer keyboard contains keys you press to enter data into the computer.
* A mouse is a small handheld device. With the mouse, you control movement of a small symbol on the screen, called the pointer, and you make selections from the screen.
* A microphone allows a user to speak into the computer, i.e., to input audio data.
* A scanner converts printed material (such as text and pictures) into a form the computer can use (hard copy to soft copy). So, it is input device.
* With a digital camera, you take pictures and then transfer the photographed images to the computer or printer instead of storing the images on traditional film.
* A Web cam is a digital video camera that allows users to create a movie or take still pictures electronically.

**Output devices:**

* A printer produces text and graphics on a physical medium such as paper.
* A monitor displays text, graphics, and videos on a screen.
* Speakers allow you to hear music, voice, and other audio (sounds).

**Both Input and Output:**

* Hard disk, flash disk, compact disk (CD) etc are devices that serve as both input and output of data.

**Processor**

The processor, also called the central processing unit (CPU), interprets and carries out the basic instructions that operate a computer. The processor significantly impacts overall computing power and manages most of a computer’s operations. Speed of a processor is measured in Hertz (HZ), kilo hertz (KHZ), megahertz (MHZ), giga hertz (GHZ) etc.

**Memory**

Memory consists of electronic components that store instructions waiting to be executed by the processor, data needed by those instructions, and the results of processed data (information). Storage capacity of memories is measured in byte (B), kilo byte (KB), megabytes (MB), giga byte (GB) etc.

1. **Operating System Tasks**

Operating system is designed to provide various services to the user. The services that an operating system may provide include the following:

* **User interface (UI)**

The user interface services can be Command-Line (CLI), Graphics User Interface (GUI), or Batch. Command Line Interface (CLI) or command interpreter allows direct command entry. The CLI primarily fetches a command from user and executes it.

* **Program execution**

The operating system is used to load a program into memory and to run that program, end execution, either normally or abnormally.

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. In the case of a system with multiple users, the OS may provide protection mechanisms to control access to the files.

* **Communications**

OS is used to exchange information, on the same computer or between computers over a network via shared memory or through message passing

* **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. In each case, the OS must provide a response that clears the error condition with the least impact on running applications. The response may range from ending the program that caused the error, to retrying the operation, to simply reporting the error to the application.

* **Resource allocation**

To allocate resources (such as CPU cycles, main memory, file storage, I/O devices) to each of multiple users or jobs running concurrently

* **Accounting:**

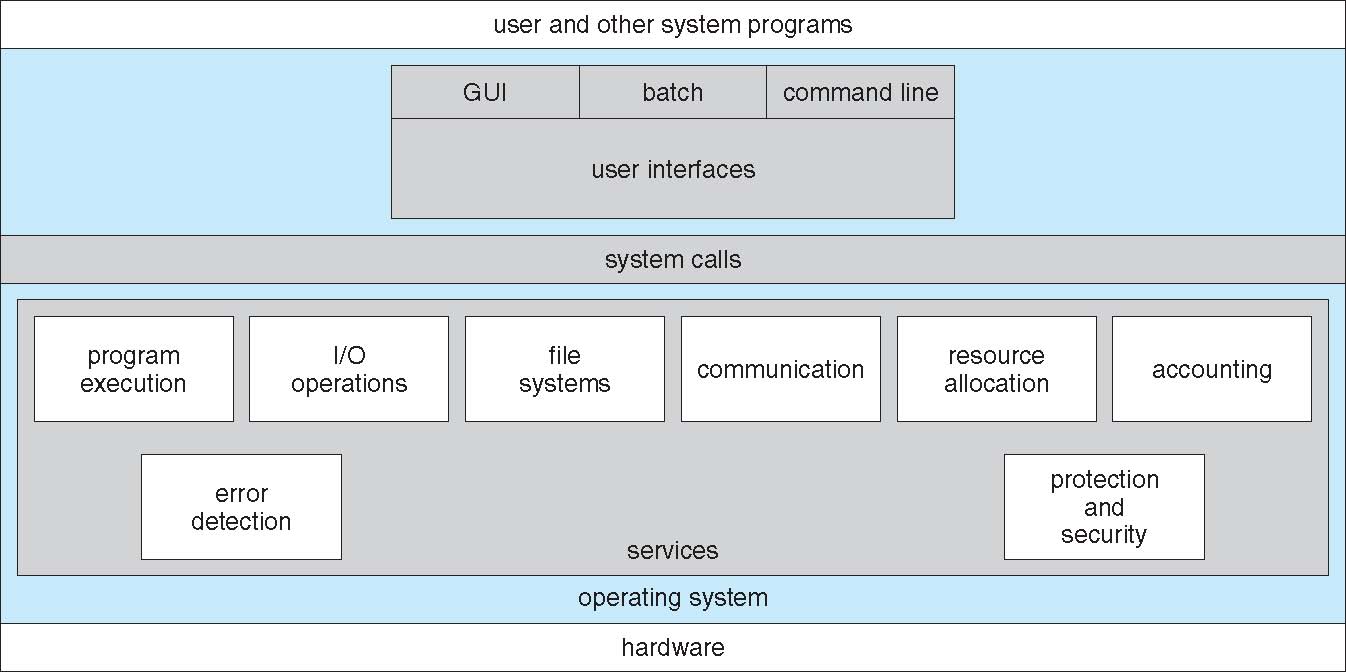
A good OS will collect usage statistics for various resources and monitor performance parameters such as response time. On any system, this information is useful in anticipating the need for future enhancements and in tuning the system to improve performance. On a multiuser system, the information can be used for billing purposes.

* **Protection and security**

Protection involves ensuring that all access to system resources is controlled

Security of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts.

The diagrammatic view of operating system user services is as follows:



System calls are programming interfaces to the operating system services, usually written in high level language. The system calls are accessed by application program interfaces, APIs.

Examples of APIs:

* + - Win32 API for Windows
    - Java API for the Java virtual machine (JVM)

The reason to access system calls using APIs instead of direct access are portability and the fact that actual system calls might be detailed and difficult to work with.

1. **Processes and Threads**
2. **Processes**

A process is just an executing program, including the current values of the program counter, registers, and variables, and which might include other resources like opened files. It is an activity of some kind. It has a program, input, output, and a state. Process execution must progress in sequential fashion.

All modern computers can do several things at the same time. While running a user program, a computer can also be reading from a disk and outputting text to a screen or printer. In a multiprogramming system, the CPU also switches from program to program, running each for tens or hundreds of milliseconds. While, strictly speaking, at any instant of time, the CPU is running only one program, in the course of 1 second, it may work on several programs, thus giving the users the illusion of parallelism.

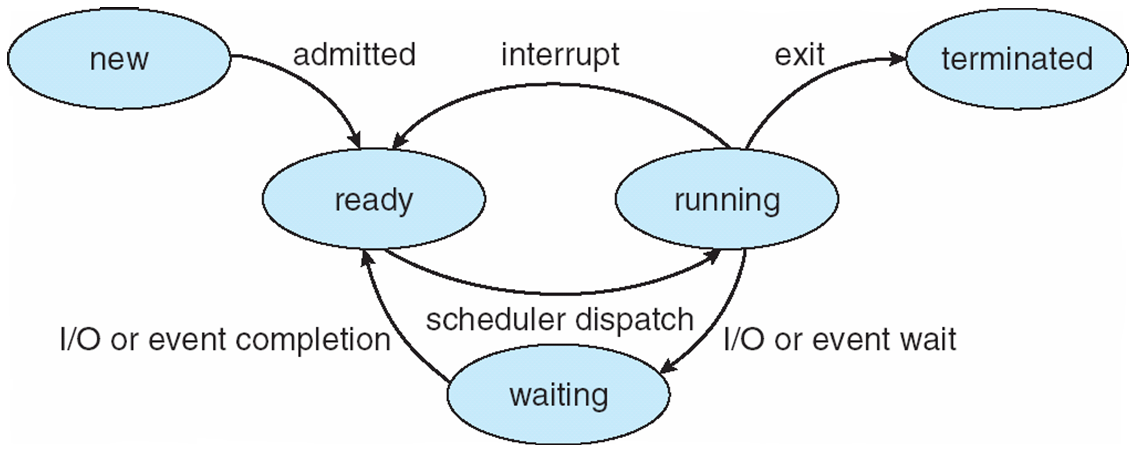
Two essential elements of a process are program code (which may be shared with other processes that are executing the same program) and a set of data associated with that code.

**Process States:**

As a process executes, it changes state and the state of a process can be one of the following:

* **New**: A process that has just been created but has not yet been admitted to the pool of executable processes by the OS. Typically, a new process has not yet been loaded into main memory, although its process control block has been created.
* **Running**: A process that is prepared to execute when given the opportunity.
* **waiting**: waiting for some event to occur
* **ready**: waiting to be assigned to a processor
* **terminated**: has finished execution

The diagram bellow illustrates the different states a process can be and the events that cause it to change state.



**The Process Control Block (PCB)**

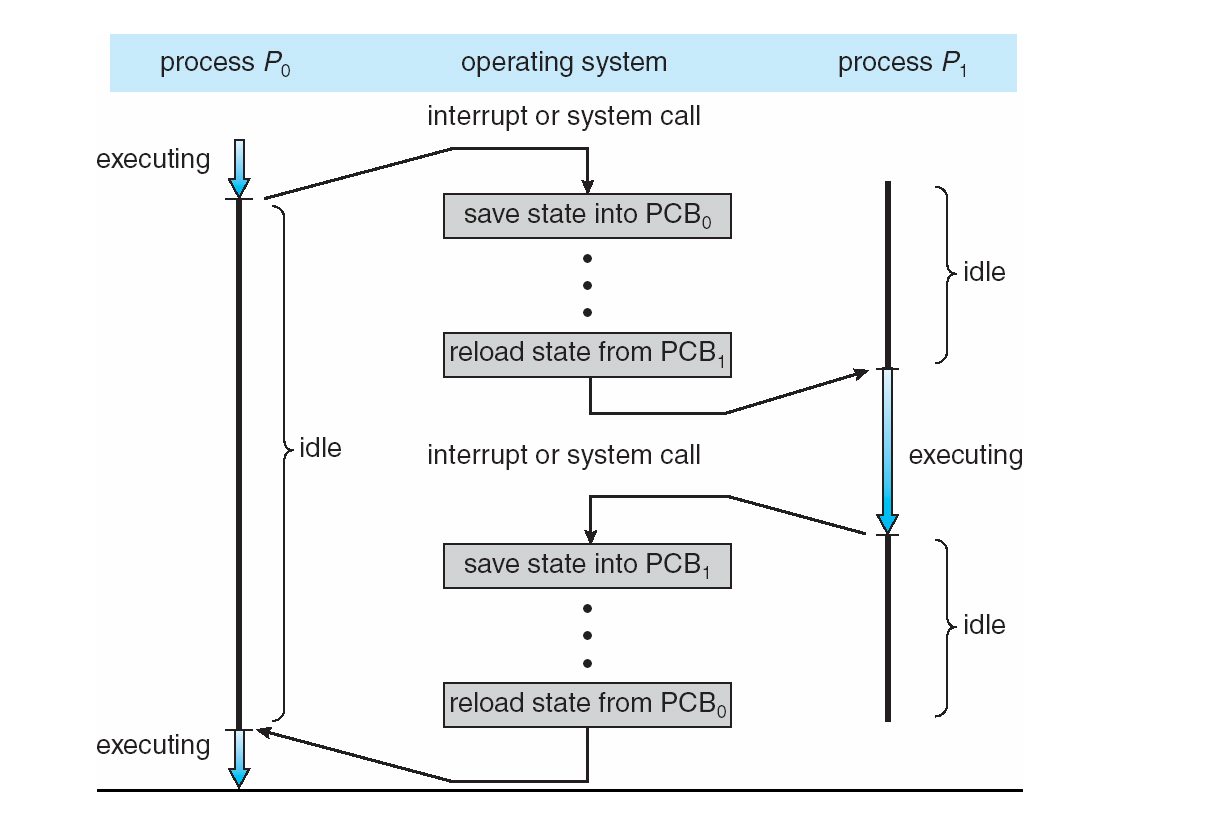
The operating system stores/saves all process related information in a memory block referred as process control block, PCB. PCB is created and managed by the OS.

The PCB contains sufficient information so that it is possible to interrupt a running process and later resume execution as if the interruption had not occurred. The process control block is the key tool that enables the OS to support multiple processes and to provide for multiprocessing. When a process is interrupted, the current values of the program counter and the processor registers (context data) are saved in the appropriate fields of the corresponding process control block, and the state of the process is changed to some other value, such as blocked (waiting) or ready.

The OS is now free to put some other process in the running state. The program counter and context data for this process are loaded into the processor registers and this process now begins to execute. Thus, we can say that a process consists of program code and associated data plus a process control block. For a single-processor computer, at any given time, at most one process is executing and that process is in the running state.

Generally, the information that the PCB of every process includes the following:

* *Identifier*: A unique identifier associated with this process, to distinguish it from all other processes
* *Process state*: If the process is currently executing, it is in the running state.
* *Program counter:* The address of the next instruction in the program to be executed.
* *Memory pointers*: Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
* *CPU scheduling information*: Priority level relative to other processes.
* *Accounting information*: May include the amount of processor time and clock time used, time limits, account numbers, and so on.
* *Context data*: These are data that are present in registers in the processor while the process is executing.
* *I/O status information*: Includes outstanding I/O requests, I/O devices (e.g., tape drives) assigned to this process, a list of files in use by the process, and so on.

When the CPU executes two or more processes, it saves the state of the currently being executed process in its own PCB, and then shifts to the next one. This can be conceived by the following diagram:

As it can be depicted from the above diagram, the CPU executes a single process at a given time.

All the run-able software on the computer, sometimes including the operating system, is organized into a number of sequential processes, or just processes for short. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process. This rapid switching back and forth is called *multiprogramming*.

**Process Creation:**

There are four principal events that cause processes to be created:

|  |  |
| --- | --- |
| 1. | System initialization. |
| 2. | Execution of a process creation system call by a running process. |
| 3. | A user request to create a new process (users can start a program by typing a command.) |
| 4. | Initiation of a batch job: (applies only to the batch systems found on large mainframes. Here users can submit batch jobs to the system (possibly remotely). When the operating system decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.) |

When an operating system is booted, often several processes are created. Some of these are foreground processes, that is, processes that interact with (human) users and perform work for them. Others are background processes, which are not associated with particular users, but instead have some specific function. For example, a background process may be designed to accept incoming requests for web pages hosted on that machine, waking up when a request arrives to service the request.

In addition to the processes created at boot time, new processes can be created afterward as well. Often a running process will issue system calls to create one or more new processes to help it do its job. Creating new processes is particularly useful when the work to be done can easily be formulated in terms of several related, but otherwise independent interacting processes.

Technically, in all these cases, a new process is created by having an existing process execute a process creation system call. That process may be a running user process, a system process invoked from the keyboard or mouse, or a batch manager process. What that process does is execute a system call to create the new process. This system call tells the operating system to create a new process and indicates, directly or indirectly, which program to run in it.

**Process Termination:**

After a process has been created, it starts running and does whatever its job is. Sooner or later the new process will terminate, usually due to one of the following conditions:

|  |
| --- |
| * Normal exit (voluntary). |
| * Error exit (voluntary): |
| * Fatal error (involuntary). |
| * Killed by another process (involuntary): When a parent process terminates, the operating system may automatically terminate all of the offspring/child processes of that parent. A parent process is a process that creates another process, which is called child process. Even if the parent process doesn’t terminate, it typically has the authority to terminate any of its offspring. |

Most processes terminate because they have done their work. When a compiler has compiled the program given to it, the compiler executes a system call to tell the operating system that it is finished. This is a normal/voluntary exit.

When a process terminates normally, all the resources are de-allocated by the operating system.

1. **Threads:**

Thread is a fundamental unit of CPU utilization. Sometimes, a thread is called a lightweight process.

A process is a collection of virtual memory space, code, data, and system resources. A thread is code that is to be serially executed within a process. A processor executes threads, not processes, so each application has at least one process, and a process always has at least one thread of execution, known as the primary thread.

A process can have multiple threads in addition to the primary thread. Prior to the introduction of multiple threads of execution, applications were all designed to run on a single thread of execution.

When a thread begins to execute, it continues until it is killed or until it is interrupted by a thread with higher priority (by a user action or the kernel’s thread scheduler). Each thread can run separate sections of code, or multiple threads can execute the same section of code. Threads executing the same block of code maintain separate stacks. Each thread in a process shares that process’s global variables and resources.

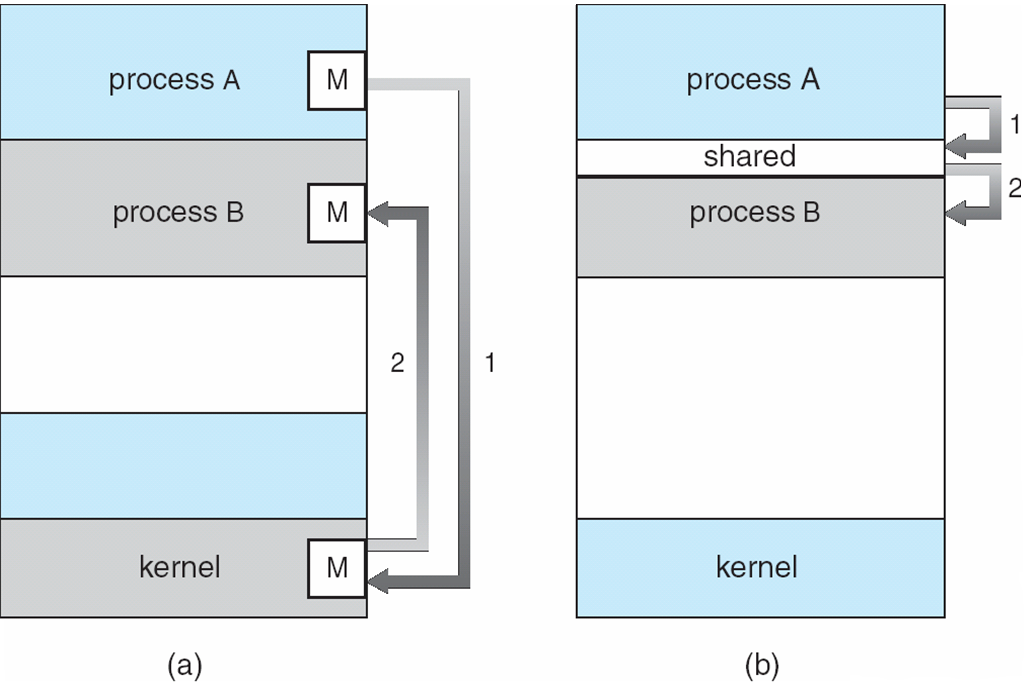
As an example of multithreaded pr ocess, a word processor may have one thread to display image, a second thread to listen to keystrokes from the keyboard, and a third one to check spelling.

1. **Inter-Process Communication (IPC)**

Processes within a system may be independent or cooperating. Independent processes cannot affect or be affected by the execution of another process. But, cooperating process can affect or be affected by other processes, including shared data. Those processes need an inter-process communication (IPC) model, which can be either of the following two:

* Shared memory
* Message passing

Bellow, are diagrams that illustrates the two IPC models: Message passing (a) and shared memory (b).



Following are some advantages of process cooperation:

* *Information sharing*: In some situation, we may want to share the content of a file among two or more processes. So, this concurrent access to a file is possible through inter-process communication.
* *Computation speed-up*: For a task to be executed faster, it is better to break it into smaller tasks and execute each in parallel. And this is possible if the system supports multiprocessing.
* *Modularity*: It is possible to write an application which is modular, each module being a process.
* *Convenience*: It is possible to run different processes in parallel of a program, such as editing, formatting, and printing a text document.

In the shared memory model, there needs to be a shared region of memory among the communicating processes, and the communication is faster, such as at a speed of memory read/write speed. And also, larger amount of information can be transferred. In such model, processes can communicate by writing and reading information on the shared memory.

In the shared memory model, because the kernel intervention is not required during writing and reading operations, the processes are responsible for ensuring that they are not writing to the same location simultaneously.

In message passing type of IPC, the communicating processes perform two operations: sending message and receiving message. When two processes need to communicate, they first establish a communication link between them and then exchange message through a send/receive mode. In such a model, the communicating processes needs system calls to the kernel and that is why they are slower as compared to the shared memory model, which doesn’t need system call to the kernel except when only system call is needed to establish the shared memory.

In the message passing model, the communicating processes may reside in different computers connected by a network. This model is useful especially in distributed system. An example of such communication model can be chat processes using the Internet.

The processes can have a direct communication or an indirect communication.

In a direct communication, the communication links are established automatically and a link is associated with exactly one pair of communicating processes, and there can be only one link. Links can be either unidirectional or bidirectional.

In the Direct Communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send and receive primitives are defined as follows:

* **Send (P, message)** - Send a message to process P
* **Receive (Q, message)** - Receive a message from process Q

In an indirect communication, messages are directed and received from mailboxes (also referred to as ports). Each mailbox has a unique id and processes can communicate only if they share a mailbox, i.e. a link is established only if processes share a common mailbox. A link may be associated with many processes and also each pair of processes may share several communication links. Again, a link may be unidirectional or bi-directional.

In this scheme, the send and receive primitives are defined as follows:

* **Send (A, message)** - Send a message to mailbox **A**
* **Receive (A, message)** - Receive a message from mailbox **A**

1. **Process Scheduling**

Scheduling a process is a way of assigning resources to the process. It is the CPU scheduler that selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them. Scheduling provides time of CPU to each process.

CPU scheduling decisions may take place when a process:

* Switches from running to waiting state
* Switches from running to ready state
* Switches from waiting to ready
* Terminates

In scheduling processes, the following criteria are considered based on the scheduling algorithm:

* CPU utilization – keep the CPU as busy as possible
* Throughput – # of processes that complete their execution per time unit
* Turnaround time – amount of time to execute a particular process, which is the sum of the times the process spends waiting to enter into the memory, waiting in the ready queue, executing in the running state by the CPU, and doing input/output operations.
* Waiting time – amount of time a process has been waiting in the **ready** queue. Note that, the scheduling algorithm doesn’t affect the time the process spends in the waiting list or the time the process spends in doing input/output operations.
* Response time – amount of time it takes from when a request was submitted until the first response is produced. This excludes the time the process is doing outputs, which is actually considered in turnaround time. This differs response time from turnaround time because of the fact that turnaround time depends on the speed of the output device, but response time doesn’t.

And based on the above criteria, there are a number of different types of scheduling algorithms which focused to optimize one of the following criteria:

* Max CPU utilization
* Max throughput
* Min turnaround time
* Min waiting time
* Min response time

Examples of such algorithms:

* First-Come, First-Served (FCFS) Scheduling
* Shortest-Job-First (SJF) Scheduling
* Round Robin (RR)

There are two major types of scheduling

* **Preemptive**: In this all the processes are executed by using some amount of time of CPU. The time of CPU is divided into the process by using some rules. If the time is divided into equal interval, then it is called Quantum time. In the Preemptive Scheduling, jobs are executed one by one, according to the scheduling algorithms; but when a higher priority job comes, then the current job will be paused and the newly coming job will be executed.
* **NON-Preemptive**: In this no Time Scheduling is used and the CPU will automatically execute the whole process one after the other. In such scheduling scheme, it is only when a process changes state or terminates that the next process gets executed. This means neither a special scheduling algorithm nor a timer is needed.

We have various Techniques in Preemptive Scheduling which are briefed bellow:

* **First Come First Serve**: As the name Suggest, the Processes those are coming first, will be Executed first. Also, the non-preemptive scheduling follows this.
* **Shortest Job first**: In this algorithm, all the process are arranged according to their time requirement to execute; the process that has the shortest execution time will be executed first and the process that has the highest execution time will be executed last. Sometimes, this algorithm is referred as shortest-remaining-time-first algorithm.
* **Priority Scheduling**: Each process has a priority, i.e., some preference issue. Then the processes get executed according to their priority, it begins with the process with the highest priority number. Equal priority processes are governed by FCFS.
* **Round Robin**: In this scheduling the time of CPU is divided into equal parts and is assign to the various processes. In this algorithm, time of CPU is known as Quantum Time. In the Round Robin, when the time of the first process has finished, it then starts the second process. This way all the processes will get executed. If a process doesn’t finish by the specified time, the CPU follows context switching, i.e. the CPU records the current state of the process and starts the next one.

1. **Deadlocks**
2. **Introduction to Deadlocks**

Computer systems are full of resources that can only be used by one process at a time. Common examples include printers, tape drives, and slots in the system's internal tables. Having two processes simultaneously writing to the printer leads to gibberish. Having two processes using the same file system table slot will invariably lead to a corrupted file system. Consequently, all operating systems have the ability to (temporarily) grant a process exclusive access to certain resources, both hardware and software.

For many applications, a process needs exclusive access to not one resource, but several. Suppose, for example, two processes each want to record a scanned document on a CD. Process A requests permission to use the scanner and is granted it. Process B is programmed differently and requests the CD recorder first and is also granted it. Now A asks for the CD recorder, but the request is denied until B releases it. Unfortunately, instead of releasing the CD recorder B asks for the scanner. At this point both processes are blocked and will remain so forever. This situation is called a deadlock.

Deadlocks can occur in a variety of situations besides requesting dedicated I/O devices. In a database system, for example, a program may have to lock several records it is using, to avoid race conditions. If process A locks record R1 and process B locks record R2, and then each process tries to lock the other one's record, we also have a deadlock. Thus, deadlocks can occur on hardware resources or on software resources.

Resources come in two types: preempt-able and non-preempt-able. A preempt-able resource is one that can be taken away from the process owning it with no bad effects.

A non-preempt-able resource, in contrast, is one that cannot be taken away from its current owner without causing the computation to fail. CD recorders are not preempt-able at an arbitrary moment, for example.

In general, deadlocks involve non-preempt-able resources. Potential deadlocks that involve preempt-able resources can usually be resolved by reallocating resources from one process to another.

The sequence of events required to use a resource is given below in an abstract form.

|  |  |
| --- | --- |
| **1.** | Request the resource. |
| **2.** | Use the resource. |
| **3.** | Release the resource. |

If the resource is not available when it is requested, the requesting process is forced to wait. In some operating systems, the process is automatically blocked when a resource request fails, and awakened when it becomes available. In other systems, the request fails with an error code, and it is up to the calling process to wait a little while and try again.

**Conditions for Deadlock:**

Four conditions must hold simultaneously for there to be a deadlock:

|  |  |
| --- | --- |
| **1.** | *Mutual exclusion condition*. Each resource is either currently assigned to exactly one process or is available; i.e., only one process can use a resource at a time. |
| **2.** | *Hold and wait condition*. A process holding at least one resource is waiting to acquire additional resources held by other processes. |
| **3.** | *No preemption condition*. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them; i,e, a resource can be released only voluntarily by the process holding it, after that process has completed its task. |
| **4.** | *Circular wait condition*. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain. This means that there exists a set {*P*0, *P*1, …, *P*n} of waiting processes such that   * *P*0 is waiting for a resource that is held by *P*1, * *P*1 is waiting for a resource that is held by *P*2, …, * *Pn*–1 is waiting for a resource that is held by *P*n, and * *Pn* is waiting for a resource that is held by *P*0 |

All the above four conditions must be present for a deadlock to occur. If one of them is absent, no deadlock is possible.

1. **Deadlock Detection and Recovery**

When this technique is used, the system does not do anything except monitor the requests and releases of resources. Every time a resource is requested or released, the resource graph is updated, and a check is made to see if any cycles exist. If a cycle exists, one of the processes in the cycle is killed. If this does not break the deadlock, another process is killed, and so on until the cycle is broken.

Detection and recovery are the strategy often used on large mainframe computers, especially batch systems in which killing a process and then restarting it is usually acceptable.

1. **Deadlock Avoidance**

The avoidance algorithm prevents deadlocks by restricting how resource requests are made.

Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. Then, the deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state.*

System is in safe state if there exists a sequence <P1, P2, …, Pn> of ALL processes in the system such that for each P**i**, the resources that P**i** can still request can be satisfied by currently available resources plus resources held by all P**j** with j < i.

If P**i** resource needs are not immediately available, then P**i** can wait until all P**j** have finished.

When P**j** is finished, P**i** can obtain needed resources, execute, return allocated resources, and terminate.

When *P****i*** terminates, *P****i* +1** can obtain its needed resources, and so on.

Observe the following facts:

* If a system is in safe state ⇒ no deadlocks
* If a system is in unsafe state ⇒ possibility of deadlock
* Avoidance ⇒ ensure that a system will never enter an unsafe state

The drawback of the avoidance algorithm is that it results in low device utilization and reduced system throughput.

Exercise: Based on the following information of a system, describe whether it is in a safe state or not at the given snapshot of the system.

Given: Five processes: p0, p1, p2, p3, p4

Total Resources:

|  |  |
| --- | --- |
| **Resource:** | **Instances:** |
| A | 10 |
| B | 5 |
| C | 7 |

Snapshot of the system at a particular instant of time:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Allocated:** | | | **Maximum Amount:** | | |
|  | **A** | **B** | **C** | **A** | **B** | **C** |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 |
| P1 | 2 | 0 | 0 | 3 | 2 | 2 |
| P2 | 3 | 0 | 2 | 9 | 0 | 2 |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 |

The freely available resources:

|  |  |  |
| --- | --- | --- |
| A | B | C |
| 3 | 3 | 2 |

* Calculate the resource needs of all processes of each type.
* Is the above system in safe state? Why?
* Can request for (1, 0, 2) by P1 be granted?
* Can request for (3, 3, 0) by P4 be granted?
* Can request for (0, 2, 0) by P0 be granted?

1. **Deadlock Prevention**

To make a system capable of preventing a deadlock, it is enough to disallow/prevent at least one of the above four conditions by controlling the way the processes get resources. So, deadlock prevention algorithm provides a set of methods or techniques for ensuring that at least one of the four conditions will never occur.

* **First condition: Disallowing Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources.
* **Second condition: Disallowing Hold and Wait –** must guarantee that whenever a process requests a resource, it does not hold any other resources. This is possible by one of the following:
* Require processes to request and be allocated all its resources before it begins execution
* Allow process to request resources only when the process has none
* **Third condition: Applying preemption:** designing the system in such a way that if a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held by this process are released.
* **Fourth condition: Disallowing Circular Wait** – impose a **total ordering** of all resource types, and require that each process requests resources in an increasing order of enumeration.

1. **Resource Allocation Graph Algorithm**

Resource allocation graph algorithm is a diagrammatic presentation that shows the allocation and requests of resource among process in a system. It consists of a set of nodes or vertices and edges. The vertices represent either resource or process and the edges show the request/assignment of resources. The set of processes is represented as P= {p1, p2, p3, … pn} and the set of resources is represented as R= {R1, R2, R3, … Rn}. The vertices for processes are represented by small circles and the vertices for resources are small squares.

The edges in a resource allocation graph are represented by a single headed arrow directing from a resource to a process, or from a process to a resource. An edge pointing from a resource to a process is an assignment edge; and an edge pointing from a process to a resource is a request edge.

The following table summarizes the components in a resource allocation graph and their meaning:

|  |  |
| --- | --- |
| **Symbol:** | **Description:** |
|  | Process |
|  | Resource type with four instances |
|  | Process Pi requests an instance of resource Rj |
|  | Process Pi holds an instance of resource Rj |

R**j**

R**j**

In a resource allocation graph, if a cycle exists and if there are single instances of each resource in the cycle, then there is a deadlock among the processes that are in the cycle. But, if there exists a cycle and there are more than one instance of each resource, there may or may not be a deadlock, and hence we can’t be sure for the existence of deadlock.

Following are some examples of resource allocation graphs:

R**4**

R**1**

R**2**

R**3**

The above resource allocation graph shows no deadlock occurrence, but the following graph does among the processes P1 and P2, excluding P3:

R**4**

R**1**

R**2**

R**3**

Sometimes, there may exist a cycle without causing a deadlock. In the following resource allocation graph, a cycle exists but a deadlock doesn’t:

R**4**

R**1**

R**2**

R**3**

In addition to the two types of edges (i.e. resource request and resource assignment edges) in a resource allocation graph, there can also be a third type known as **claim edge** to show that the resource needs of processes in the future. Resource need of a process is the difference we get when we subtract the currently allocated resources to the process from the maximum resource requirement of the process.

Claim edges are directed graph pointing from a process to a resource but with dotted line. For example, in the following resource allocation graph, the resource need of P1 is an instance of R1and the need of P2 is an instance of R3. A claim edge describes that a process may require the indicated resource in the future.

R**4**

R**1**

R**2**

R**3**

When a process is granted the resources it claimed, then the claim edge is converted to assignment edge in the graph; and when the process releases the allocated resources, the assignment edge is converted to a claim edge.

Generally, the resource allocation graph can be used for deadlock avoidance only when each resource type has a single instance.

1. **Banker’s Algorithm**

The banker’s algorithm is one type of deadlock avoidance algorithm. This algorithm is used when the resources consist of multiple instances. Each process in the system must declare its maximum resource requirement before beginning execution.

Below are the data structures required in the banker’s algorithm:

Let *n* = number of processes, and *m* = number of resources types:

* *Available:* Vector of length *m*. If *available* [*j*] = *k*, there are *k* instances of resource type *Rj*available.
* *Max: n X m* matrix. If *Max* [*i,j*] = *k*, then process *Pi* may request at most *k* instances of resource type *Rj*.
* *Allocation: n X m* matrix. If *Allocation* [*i,j*] = *k* then *Pi* is currently allocated *k* instances of *Rj.*
* *Need: n X m* matrix. If *Need*[*i,j*] = *k*, then *Pi* may need *k* more instances of *Rj*to complete its task.

*Need* [*i,j]* = *Max*[*i,j*] – *Allocation* [*i,j*].

The banker’s algorithm is composed of two algorithms:

* Safety algorithm: to check the current status of the processes are in safe state or not, and
* The resource request algorithm: to check whether the system will remain in safe state if allocating the resources that an arbitrary process *Pi* requested.

**The Safety Algorithm:**

To check whether a system is in safe state or not, the safety algorithm proceeds as follows:

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

**work** = **available** ………. *work* is vector of resources & *available* is free resources

**finish[i]** =**false** *…………. for i* = 1, 2, …, *n.* This indicates the fact that none of the processes are finished before execution.

1. Find an ***i*** such that both of the following are true:

(a) *finish* [*i*] = *false*

(b) *needi* ≤ *work*

If no such *i* exists, go to step 4.

1. **work = work + allocationi**  
   **finish[i] = true**  
   Go to step 2.
2. If **finish [i]** = = **true** for all *i*, then the system is in a safe state

**The Resource Request Algorithm**

This algorithm is used to check what if an arbitrary process P*i* requests a resource.

Let ***requesti*** be the request vector for process ***Pi***. If ***requesti*** [***j***] = ***k*** then process ***Pi*** wants/requests ***k*** instances of resource type ***Rj***. When ***Pi*** requests the resources, the algorithm proceeds as follows:

1. If *Requesti* ≤ *Needi* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If Requesti ≤ Available, go to step 3. Otherwise, Pi must wait, since resources are not available.
3. Pretend to allocate requested resources to Pi by modifying the state as follows:

Available = Available - Requesti;

Allocationi = Allocationi + Requesti;

Needi = Needi – Requesti;

* *If safe* ⇒ *the resources are allocated to Pi.*
* *If unsafe* ⇒ *Pi must wait, and the old resource-allocation state is restored*

1. **Memory and I/O Device Management and File Systems**
2. **Basics of Memory Management**

A memory is used to store programs and data. A program must be brought (from disk) into memory and placed within a process to run. Many programs/processes can run in memory at the same time and for this, protection of memory (which is the responsibility of the processor, not the operating system for some reason) is required to ensure correct operation.

In a uni-programming system, main memory is divided into two parts:

* one part for the operating system (resident monitor, kernel) and
* another part for the program currently being executed (user program)

In a multiprogramming system, the “user” part of memory must be further subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the operating system and is known as *memory management*.

Effective memory management is vital in a multiprogramming system. If only a few processes are in memory, then for much of the time all of the processes will be waiting for I/O and the processor will be idle. Thus memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time.

Memory management needs to satisfy the following requirements, and many more:

* Relocation
* Protection
* Sharing

**Relocation:**

In a multiprogramming system, the available main memory is generally shared among a number of processes. Typically, it is not possible for the programmer to know in advance which other programs will be resident in main memory at the time of execution of his or her program. In addition, we would like to be able to swap active processes in and out of main memory to maximize processor utilization by providing a large pool of ready processes to execute. Once a program is swapped out to disk, it would be difficult to place it in the same main memory region when it next swapped back in. Instead, we may need to **relocate** the process to a different area of memory.

Thus, we cannot know ahead of time where a program will be placed, and we must allow for the possibility that the program may be moved about in main memory due to swapping. These facts raise some technical concerns related to addressing.

The operating system will need to know the location of process control information and of the execution stack, as well as the entry point to begin execution of the program for this process. In addition, however, the processor must deal with memory references within the program. Branch instructions contain an address to reference the instruction to be executed next. Data reference instructions contain the address of the byte or word of data referenced. Somehow, the processor hardware and operating system software must be able to translate the memory references found in the code of the program into actual physical memory addresses, reflecting the current location of the program in main memory.

**Protection:**

Each process should be protected against unwanted interference by other processes, whether accidental or intentional. Thus, programs in other processes should not be able to reference memory locations in a process for reading or writing purposes without permission.

Because the location of a program in main memory is unpredictable due to the relocation of processes after swapping, it is impossible to check absolute addresses at compile time to assure protection. Furthermore, most programming languages allow the dynamic calculation of addresses at run time (e.g., by computing an array subscript or a pointer into a data structure). Hence all memory references generated by a process must be checked at run time to ensure that they refer only to the memory space allocated to that process.

Note that the memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software). This is because the OS cannot anticipate all of the memory references that a program will make. Even if such anticipation were possible, it would be prohibitively time consuming to screen each program in advance for possible memory-reference violations. Thus, it is only possible to assess the permissibility of a memory reference (data access or branch) at the time of execution of the instruction making the reference. To accomplish this, the processor hardware must have that capability.

**Sharing:**

Any protection mechanism must have the flexibility to allow several processes to access the same portion of main memory. For example, if a number of processes are executing the same program, it is advantageous to allow each process to access the same copy of the program rather than have its own separate copy. Processes that are cooperating on some task may need to share access to the same data structure. The memory management system must therefore allow controlled access to shared areas of memory without compromising essential protection.

1. **Paging, Page tables, and Virtual Memory**

Suppose that main memory is partitioned into equal fixed-size chunks that are relatively small, and that each process is also divided into small fixed-size chunks of the same size. Then the chunks of a process, known as **pages**, could be assigned to available chunks of memory, known as **frames**, or page frames.

To illustrate the paging concept, take a look at the following figures.

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

Main Memory

Frame number

1. Fifteen available frames

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

**A0**

**A1**

**A2**

**A3**

Main Memory

1. Load Process A

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

**A0**

**A1**

**A2**

**A3**

**B0**

**B1**

**B2**

Main Memory

1. Load Process B

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

**A0**

**A1**

**A2**

**A3**

**B0**

**B1**

**B2**

**C0**

**C1**

**C2**

**C3**

Main Memory

1. Load Process C

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

**A0**

**A1**

**A2**

**A3**

**C0**

**C1**

**C2**

**C3**

Main Memory

1. Swap out process B

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

**A0**

**A1**

**A2**

**A3**

**D0**

**D1**

**D2**

**C0**

**C1**

**C2**

**C3**

**D3**

**D4**

Main Memory

1. Load Process D

At a given point in time, some of the frames in memory are in use and some are free. A list of free frames is maintained by the OS. Process A, stored on disk, consists of four pages. When it is time to load this process, the OS finds four free frames and loads the four pages of process A into the four frames. Process B, consisting of three pages, and process C, consisting of four pages, are subsequently loaded. Then process B is suspended and is swapped out of main memory. Later, the OS loads a new process, i.e. process D, which consists of five pages.

Now suppose, as in this example, that there are not sufficient unused contiguous frames to hold the process. Does this prevent the operating system from loading D? No.

The operating system maintains a page table for each process. The page table shows the frame location for each page of the process. Within the program, each logical address consists of a page number and an offset within the page.

With paging, the logical-to-physical address translation is done by processor hardware. Now the processor must know how to access the page table of the current process. The processor then converts the logical address (page number and offset) to physical address (frame number and offset) using the page table.

In our example above, at the instant of time process D is loaded in memory, the page tables that correspond each process looks like the following:

0

0

Process A page table

1

1

2

2

3

3

0

-

Process B page table

1

-

2

-

0

7

Process C page table

1

8

2

9

3

10

0

4

Process D page table

1

5

2

6

3

11

4

12

Free frame list

13

14

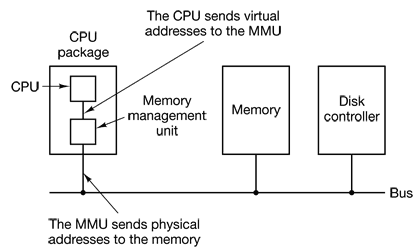
A page table contains one entry for each page of the process, so that the table is easily indexed by the page number (starting at page 0). Each page table entry contains the number of the frame in main memory, if any, that holds the corresponding page. In addition, the OS maintains a single free-frame list of all the frames in main memory that are currently unoccupied and available for pages.

As a summary, with simple paging, main memory is divided into many small equal-size frames. Each process is divided into frame-size pages. Smaller processes require fewer pages; larger processes require more. When a process is brought in, all of its pages are loaded into available frames, and a page table is set up.

**Virtual Memory:**

Virtual memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of secondary memory available and not by the actual number of main storage locations.

The program-generated addresses are called virtual addresses and form the virtual address space. On computers without virtual memory, the virtual address is put directly onto the memory bus and causes the physical memory word with the same address to be read or written. When virtual memory is used, the virtual addresses do not go directly to the memory bus. Instead, they go to an MMU (Memory Management Unit) that maps the virtual addresses onto the physical memory addresses. The following figure illustrates this:



All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process may be swapped in and out of main memory such that it occupies different regions of main memory at different times during the course of execution.

A process may be broken up into a number of pieces (pages or segments) and these pieces need not be contiguously located in main memory during execution. The combination of dynamic run-time address translation and the use of a page, or segment, table permits this.

It is not necessary that all of the pages or all of the segments of a process be in main memory during execution. If the piece (segment or page) that holds the next instruction to be fetched and the piece that holds the next data location to be accessed are in main memory, then the execution may proceed for a time.

There are two major techniques/capabilities for better system utilization:

* **The capability to maintain more number of processes in main memory**: that is because we are only going to load some of the pieces of any particular process, there is room for more processes. This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in a Ready state at any particular time.
* **The possibility to write larger processes that require more memory space than the total available main memory in a system**: Before this possibility, a programmer must be acutely aware of how much memory is available. If the program being written is too large, the programmer must devise ways to structure the program into pieces (pages or segments) that can be loaded separately in some sort of overlay strategy. With virtual memory based on paging or segmentation, that job is left to the OS and the hardware. As far as the programmer is concerned, he or she is dealing with a huge memory, the size associated with disk storage. The OS automatically loads pieces of a process into main memory as required.

Because a process executes only in main memory, that memory is referred to as real memory. But a programmer or user perceives a potentially much larger memory— that which is allocated on disk. This latter is referred to as virtual memory. Virtual memory allows for very effective multiprogramming and relieves the user of the unnecessarily tight constraints of main memory.

1. **Page Replacement Algorithm/Policy**

Page replacement deals with the selection of a page in main memory to be replaced when a new page must be brought in. When all of the frames in main memory are occupied and it is necessary to bring in a new page to satisfy a page fault, the replacement policy determines which page currently in memory is to be replaced. All of the policies have as their objective that the page that is removed should be the page least likely to be referenced in the near future. Because of the principle of locality, there is often a high correlation between recent referencing history and near-future referencing patterns. Thus, most policies try to predict future behavior on the basis of past behavior.

Following are certain basic algorithms that are used for the selection of a page to replace.

* Optimal
* Least Recently Used (LRU)
* First-in-first-out (FIFO)
* Clock

The optimal policy selects for replacement that page for which the time to the next reference is the longest. Clearly, this policy is impossible to implement, because it would require the OS to have perfect knowledge of future events. However, it does serve as a standard against which to judge real-world algorithms.

The least recently used (LRU) policy replaces the page in memory that has not been referenced for the longest time.

In fact, the LRU policy does nearly as well as the optimal policy. The problem with this approach is the difficulty in implementation. One approach would be to tag each page with the time of its last reference; this would have to be done at each memory reference, both instruction and data. Even if the hardware would support such a scheme, the overhead would be tremendous. Alternatively, one could maintain a stack of page references, again an expensive prospect.

The first-in-first-out (FIFO) policy treats the page frames allocated to a process as a circular buffer, and pages are removed in round-robin style. All that is required is a pointer that circles through the page frames of the process. This is therefore one of the simplest page replacement policies to implement. The logic behind this choice, other than its simplicity, is that one is replacing the page that has been in memory the longest: A page fetched into memory a long time ago may have now fallen out of use. This reasoning will often be wrong, because there will often be regions of program or data that are heavily used throughout the life of a program. Those pages will be repeatedly paged in and out by the FIFO algorithm.

Although the LRU policy does nearly as well as an optimal policy, it is difficult to implement and imposes significant overhead. On the other hand, the FIFO policy is very simple to implement but performs relatively poorly. Over the years, OS designers have tried a number of other algorithms to approximate the performance of LRU while imposing little overhead. Many of these algorithms are variants of a scheme referred to as the clock policy.

The simplest form of clock policy requires the association of an additional bit with each frame, referred to as the use bit. When a page is first loaded into a frame in memory, the use bit for that frame is set to 1. Whenever the page is subsequently referenced its use bit is set to 1. For the page replacement algorithm, the set of frames that are candidates for replacement is considered to be a circular buffer, with which a pointer is associated. When a page is replaced, the pointer is set to indicate the next frame in the buffer after the one just updated. When it comes time to replace a page, the OS scans the buffer to find a frame with a use bit set to 0. Each time it encounters a frame with a use bit of 1, it resets that bit to 0 and continues on. If any of the frames in the buffer have a use bit of 0 at the beginning of this process, the first such frame encountered is chosen for replacement. If all of the frames have a use bit of 1, then the pointer will make one complete cycle through the buffer, setting all the use bits to 0, and stop at its original position, replacing the page in that frame.

We can see that this policy is similar to FIFO, except that, in the clock policy, any frame with a use bit of 1 is passed over by the algorithm. The policy is referred to as a clock policy because we can visualize the page frames as laid out in a circle.

A number of operating systems have employed some variation of this simple clock policy.

In summary, the page replacement algorithm cycles through all of the pages in the buffer looking for one that has not been modified since being brought in and has not been accessed recently. Such a page is a good bet for replacement and has the advantage that, because it is unmodified, it does not need to be written back out to secondary memory. If no candidate page is found in the first sweep, the algorithm cycles through the buffer again, looking for a modified page that has not been accessed recently.

1. **Segmentation**

A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of segments. It is not required that all segments of all programs be of the same length, although there is a maximum segment length. As with paging, a logical address using segmentation consists of two parts, in this case a segment number and an offset.

Whereas paging is invisible to the programmer, segmentation is usually visible and is provided as a convenience for organizing programs and data. Typically, the programmer or compiler will assign programs and data to different segments. For purposes of modular programming, the program or data may be further broken down into multiple segments. The principal inconvenience of this service is that the programmer must be aware of the maximum segment size limitation.

1. **Principles of i/o In Computer Systems**

There are three basic techniques for performing I/O:

* **Programmed I/O**: The processor issues an I/O command, on behalf of a process, to an I/O module; that process then busy waits for the operation to be completed before proceeding.
* **Interrupt-driven I/O**: The processor issues an I/O command on behalf of a process. There are then two possibilities. If the I/O instruction from the process is non-blocking, then the processor continues to execute instructions from the process that issued the I/O command. If the I/O instruction is blocking, then the next instruction that the processor executes is from the OS, which will put the current process in a blocked state and schedule another process.
* **Direct memory access (DMA)**: A DMA module controls the exchange of data between main memory and an I/O module. The processor sends a request for the transfer of a block of data to the DMA module and is interrupted only after the entire block has been transferred.

The figure bellow indicates, in general terms, the DMA logic. The DMA unit is capable of mimicking the processor and, indeed, of taking over control of the system bus just like a processor. It needs to do this to transfer data to and from memory over the system bus.

Data count

Data register

Address register

Control logic

Data lines

Address lines

Request to DMA

Acknowledge from DMA

Interrupt

Read

Write

The DMA technique works as follows. When the processor wishes to read or write a block of data, it issues a command to the DMA module by sending to the DMA module the following information:

* Whether a read or write is requested, using the read or write control line between the processor and the DMA module
* The address of the I/O device involved, communicated on the data lines
* The starting location in memory to read from or write to, communicated on the data lines and stored by the DMA module in its address register
* The number of words to be read or written, again communicated via the data lines and stored in the data count register

The processor then continues with other work. It has delegated this I/O operation to the DMA module. The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus, the processor is involved only at the beginning and end of the transfer.

The DMA mechanism can be configured in a variety of ways. Some possibilities are shown in the figure bellow.

Processor

DMA

I/O

I/O

Memory

..........

1. Single-bus, detached DMA
2. Single-bus, integrated DMA-I/O

Processor

I/O

Memory

I/O

DMA

I/O

DMA

I/O

1. Single-bus, detached DMA

Processor

DMA

I/O

I/O

Memory

I/O

I/O bus

System bus

In the first example, all modules share the same system bus. The DMA module, acting as a surrogate processor, uses programmed I/O to exchange data between memory and an I/O module through the DMA module. This configuration, while it may be inexpensive, is clearly inefficient: As with processor-controlled programmed I/O, each transfer of a word consumes two bus cycles (transfer request followed by transfer).

The number of required bus cycles can be cut substantially by integrating the DMA and I/O functions. As figure (b) indicates, this means that there is a path between the DMA module and one or more I/O modules that does not include the system bus. The DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules. This concept can be taken one step further by connecting I/O modules to the DMA module using an I/O bus (figure (c)). This reduces the number of I/O interfaces in the DMA module to one and provides for an easily expandable configuration. In all of these cases ( figure (b) and (c) ), the system bus that the DMA module shares with the processor and main memory is used by the DMA module only to exchange data with memory and to exchange control signals with the processor. The exchange of data between the DMA and I/O modules takes place off the system bus.

1. **Why External Storage?**

All computer applications need to store and retrieve information. While a process is running, it can store a limited amount of information within its own address space. However the storage capacity is restricted to the size of the virtual address space. For some applications this size is adequate, but for others, such as airline reservations, banking, or corporate record keeping, it is far too small.

A second problem with keeping information within a process’ address space is that when the process terminates, the information is lost. For many applications, (e.g., for databases), the information must be retained for weeks, months, or even forever.

A third problem is that it is frequently necessary for multiple processes to access (parts of) the information at the same time. But, if the information is stored in an address space of a single process, then other processes cannot get access to that information. The way to solve this problem is to make the information itself independent of any one process.

Thus we have three essential requirements for long-term information storage:

* It must be possible to store a very large amount of information.
* The information must survive the termination of the process using it.
* Multiple processes must be able to access the information concurrently.

The usual solution to all these problems is to store information on disks and other external media in units called *files*. Processes can then read them and write new ones if needed. Information stored in files must be persistent, that is, not be affected by process creation and termination. A file should only disappear when its owner explicitly removes it.

1. **Allocation Methods**

On secondary storage, a file consists of a collection of blocks. The operating system or file management system is responsible for allocating blocks to files. This raises two management issues. First, space on secondary storage must be allocated to files, and second, it is necessary to keep track of the space available for allocation.

There are three major file **allocation** methods on a disk: contiguous, chained (also called linked list), and indexed methods.

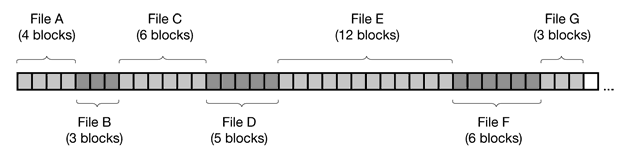
**Contiguous Allocation:**

With contiguous allocation, a single contiguous set of blocks is allocated to a file at the time of file creation. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file. Contiguous allocation is the best from the point of view of the individual sequential file. Multiple blocks can be read in at a time to improve I/O performance for sequential processing. It is also easy to retrieve a single block. For example, if a file starts at block ***b***, and the ***i th*** block of the file is wanted, its location on secondary storage is simply ***b + i –1***.

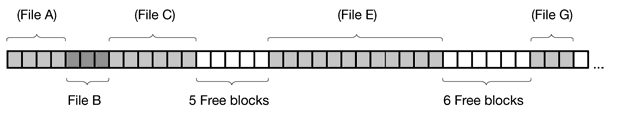
Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. With 2-KB blocks, it would be allocated 25 consecutive blocks.

Contiguous allocation presents some problems. External fragmentation will occur, making it difficult to find contiguous blocks of space of sufficient length. From time to time, it will be necessary to perform a compaction algorithm to free up additional space on the disk.

Following is a figure that shows how files are stored in contiguous scheme.



And the following figure shows how the storage looks like after file D and F are deleted.

****

**Advantages of Contiguous Scheme:**

Contiguous disk space allocation has the following two significant advantages.

* First, it is simple to implement because keeping track of where a file’s blocks are is reduced to remembering two numbers: the disk address of the first block and the number of blocks in the file. Given the number of the first block, the number of any other block can be found by a simple addition.
* Second, the read performance is excellent because the entire file can be read from the disk in a single operation. Only one seek is needed (to the first block). After that, no more seeks or rotational delays are needed so data come in at the full bandwidth of the disk. Thus contiguous allocation is simple to implement and has high performance.

Although contiguous allocation has these advantages, it has also the problem of fragmentation as mentioned above. As an example, in the second figure above, two files, D and F have been removed. When a file is removed, its blocks are freed, leaving a run of free blocks on the disk. The disk is not compacted on the spot to squeeze out the hole since that would involve copying all the blocks following the hole, potentially millions of blocks, As a result, the disk ultimately consists of files and holes, as illustrated in the figure.

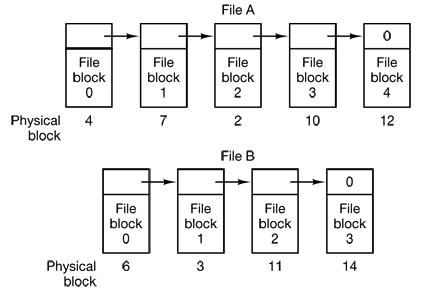
However, there is one situation in which contiguous allocation is feasible and, in fact, widely used: on CD-ROMs. Here all the file sizes are known in advance and will never change during subsequent use of the CD-ROM file system.

**Chained or Linked List Allocation:**

At the opposite extreme from contiguous allocation is chained allocation. Typically, allocation is on an individual block basis. Each block contains a pointer to the next block in the chain. Thus a file is a linked list of disk blocks. Again, the file allocation table needs just a single entry for each file, showing the starting block and the length of the file.

The first word of each block is used as a pointer to the next one. The rest of the block is for data.

The diagram bellow shows how the chained allocation concept works:



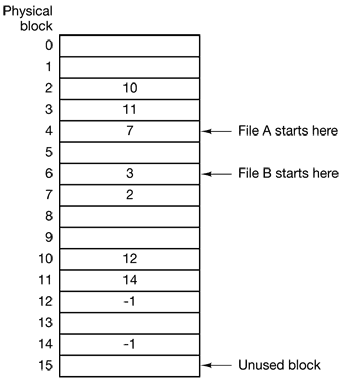
Although reading a file sequentially is straightforward, random access is extremely slow. To get to block *n*, the operating system has to start at the beginning and read all the *n – 1* blocks, prior to it, one at a time.

Also, the amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes. While not fatal, having a peculiar size is less efficient because many programs read and write in blocks whose size is a power of two. With the first few bytes of each block occupied to a pointer to the next block, reads of the full block size require acquiring and concatenating information from two disk blocks, which generates extra overhead due to the copying.

**Linked List Allocation Using a Table in Memory:**

This is similar to the chained technique in that it implements a linked list. The difference is that, it takes the pointer word from each disk block and put it in a table in memory. This eliminates the problem of contiguous allocation and as well chained allocation.

The figure bellow illustrates this allocation method by simply storing the two files above (File A and File B) described in the chained allocation technique.



Both chains are terminated with a special marker (e.g., –1) that is not a valid block number. Such a table in main memory is called a **FAT (File Allocation Table).**

Using this organization, the entire block is available for data. Furthermore, random access is much easier. Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory, so it can be followed without making any disk references. Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is.

The primary disadvantage of this method is that the entire table must be in memory all the time to make it work. With a 20-GB disk and a 1-KB block size, the table needs 20 million entries, one for each of the 20 million disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 60 MB or 80 MB of main memory all the time, depending on whether the system is optimized for space or time. Conceivably the table could be put in page-able memory, but it would still occupy a great deal of virtual memory and disk space as well as generating extra paging traffic.

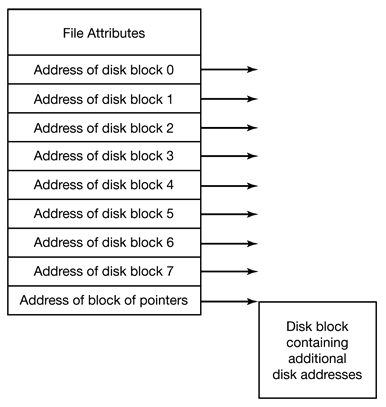
**Indexed Allocation (I-Node):**

In this case, the file allocation table contains a separate one-level index for each file; the index has one entry for each portion allocated to the file. Typically, the file indexes are not physically stored as part of the file allocation table. Rather, the file index for a file is kept in a separate block and the entry for the file in the file allocation table points to that block.

The big advantage of this scheme over linked files using an in-memory table is that the i-node need only to be in memory when the corresponding file is open. If each i-node occupies *n* bytes and a maximum of *k* files may be open at once, the total memory occupied by the array holding the i-nodes for the open files is only *kn* bytes. Only this much space need be reserved in advance.

This array is usually far smaller than the space occupied by the file table described in the previous section. The reason is simple. The table for holding the linked list of all disk blocks is proportional in size to the disk itself. If the disk has *n* blocks, the table needs *n* entries. As disks grow larger, this table grows linearly with them. In contrast, the i-node scheme requires an array in memory whose size is proportional to the maximum number of files that may be open at once. It does not matter if the disk is 1 GB or 10 GB or 100 GB.

One problem with i-nodes is that if each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit? One solution is to reserve the last disk address not for a data block, but instead for the address of a block containing more disk block addresses, as shown in the figure bellow. Even more advanced would be two or more such blocks containing disk addresses or even disk blocks pointing to other disk blocks full of addresses.



1. **File Systems**

From the user’s point of view, one of the most important parts of an operating system is the file system. The file system provides the resource abstractions typically associated with secondary storage. The file system permits users to create data collections, called files, with desirable properties, such as:

* **Long-term existence**: Files are stored on disk or other secondary storage and do not disappear when a user logs off.
* **Sharable between processes**: Files have names and can have associated access permissions that permit controlled sharing.
* **Structure**: Depending on the file system, a file can have an internal structure that is convenient for particular applications. In addition, files can be organized into hierarchical or more complex structure to reflect the relationships among files.

Any file system provides not only a means to store data organized as files, but also a collection of functions that can be performed on files. Typical operations include the following:

* **Create**: A new file is defined and positioned within the structure of files.
* **Delete**: A file is removed from the file structure and destroyed. There is always a system call for this purpose.
* **Open**: An existing file is declared to be “opened” by a process, allowing the process to perform functions on the file. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.
* **Close**: The file is closed with respect to a process, so that the process no longer may perform functions on the file, until the process opens the file again.
* **Read**: A process reads all or a portion of the data in a file. The caller must specify how much data are needed and must also provide a buffer to put them in.
* **Write**: A process updates a file, either by adding new data that expands the size of the file or by changing the values of existing data items in the file.

Typically, a file system maintains a set of attributes associated with the file.

These include owner, creation time, time last modified, access privileges, and so on.

1. **File System Layout**

File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition. Sector 0 of the disk is called the MBR (Master Boot Record) and is used to boot the computer. The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition. One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR. The first thing the MBR program does is locate the active partition, read in its first block, called the boot block, and execute it. The program in the boot block loads the operating system contained in that partition.