There are four elements in computer system: the hardware, the users, the application programs, and the operating system. The hardware—the central processing unit (CPU), the input/output (I/O) devices and the memory—gives the basic computing resources. The application programs—such as compilers, spreadsheets, word processors and web browsers—explain the ways in which these resources are utilized to solve the computing troubles of the users. The operating system controls and synchronize the use of the hardware among the various application programs for the various users. [1]

The operating system propounds the means for the proper use of these resources in the operation of the computer system. An operating system is similar to a administration. Like a administration, it performs no useful function by itself. It simply provides an administration within which other programs can do suitable work. Operating systems can be traversed from two viewpoints: the user and the system. [1]

Scheduling can be as simple as running the next process, or it can use relatively complex rules to pick a running process. Processing is going on even as Input Output is occurring in preparation for future CPU work. Off Line managing; not only are Input Output and CPU happening simultaneously, but some off-board processing is occurring with the Input Output. The CPU is wasted if a job waits for Input Output. This steers to Multiprogramming (dynamic switching). While one job waits for a resource, the CPU can find another job to run. It means that few jobs are ready to run and only need the CPU in order to continue. CPU scheduling is needed for it. All of this conducts to resource scheduling, memory management, and deadlock protection.

Other attributes include Time Sharing - multiprogramming environment that's also interactive. Multiprocessing - Compactly coupled systems that communicate via shared memory. Used for scientific approach. Used for speed enhancement by putting together a number of off-the-shelf processors. Distributed Systems - Loosely coupled systems that communicate via message passing. Advantages include resource sharing; speed up, reliability, communication. Real Time Systems - Quick response time is main characteristic. This is used in control of applications where rapid response to a stimulus is vital. It is
• A program that is accomplish by the processor that frequently relinquishes control and must depend on the processor to recover control.

• An interface between applications and hardware

• A set of plans that enable a group of people to use a computer system. A program that controls the implementation of application programs

• A program that resolve between application programs and the hardware

The general thought behind these definitions is that the operating systems control and support the use of computer systems i.e.:

a. Support

b. Usage

c. Computer system

d. Control

**Computer System:** A system of interconnected computers that share a central use system and different peripheral devices such as a printers, scanners, or routers. Each computer connected to the system can work independently, but has the ability to communicate with other external devices and computers. OS is a element of the computer software, it is a program. It is a very special program that is the first to be achieved when the computer is switched on, and is supposed to control and support the execution of other programs and the net usage of the computer system.

**Control:** The operating system controls the expenditure of the computer resources - hardware devices and software utilities. We can think of an operating system as a Resource Manager. Here are some of the resources commanded by the OS: Processors, Secondary Memory, Main memory, Peripheral devices and Information.

**Support:** The operating system provides a number of facility to serve the users of the computer system: For the programmers: Utilities - debuggers, editors, file management, etc. For the end users - give the interface to the application programs. For programs -
loads instructions and data into memory, devises for usage, prepares I/O handles
interrupts and error conditions.

**Usage:** A user is an agent, either a software agent or human agent (end-user), who uses a
computer or network service. A user often has a user account and is identified by a
username (also user name). Other terms for username include login name, screen name
(also screen name), nickname (also nick), or handle, which is derived from the identical
citizen's Band radio term.

Users are also widely characterized as the class of people that use a system without entire
technical competence required to understand the system completely. In projects in
which the actor of the system is an additional system or a software agent, it is quite
sensible that there is no end-user for the system. In this case, the end-users for the system
would be uninventive end-users.

**Computer System Users**

End users - utilize application programs, e.g. Internet explorer.

Human users and Programmers - use program development tools such as debuggers,
editors.

Programs - operate memory, use CPU time, and use I/O devices.

The stratified view of the computer system illustrates how the operating system interacts
with the users of the computer system:
Objectives in OS design

- Efficiency- it allows computer to use resources logically.
- Ability to evolve- constructed in a way to permit effectual development, testing and introduction of new functions without interfere with service.
- Convenience – it makes computer user friendly.

1.2 Types of Operating Systems

1.2.1 Serial Processing Operating System

In the era of 1940’s – 1950’s programmer interacted straightly with hardware. There was no operating system at that time.

Limitations of Serial Processing

- Setup Time- Setup included loading the compiler, source program, saving compiled program, and loading and linking. If an error occurred - start over.
- Scheduling - users sign up for machine time. Misused computing time.
1.2.2 Simple Batch Systems

- Programs were acceded on cards or tape to an operative who batches jobs together sequentially. The program that controls the implementation of the jobs was called monitor - a manageable version of an operating system. The interface to the monitor was experted through Job Control Language (JCL). For example, a JCL request could be to run the compiler for a specific programming language, then to link and load the program, then to run the user program.

- Simple batch system upgrade the utilization of computers.

Hardware features

- Memory protection: do not allow the memory area holding the monitor to be altered

- Timer: prevents a job from monopolizing the system

Problems

- Bad utilization of CPU time - the processor stays idle while I/O devices are in use.

1.2.3 Multiprogrammed Batch Operating Systems

More than one program responses in the main memory. While a program A uses an I/O device the processor does not stay idle, instead it runs another program B.

![Figure 1- 3 Uniprogramming](image)
Multiprogramming with more than one job

New features

- Memory management - to have some jobs ready to run, they must be kept in main memory

- Job scheduling - the processor must resolve which program to run.

1.2.4 Multiprogramming Operating System

In multiprogramming systems, the running task retains running until it performs an operation that needs waiting for an outer event (e.g. reading from a tape) or until the computer's scheduler compulsorily swaps the running task out of the CPU. Multiprogramming systems are planned to maximize CPU usage.

In the early days of computing, CPU time was costly, and peripherals were very slow. When the computer ran a program that needed ingress to a peripheral, the Central processing unit (CPU) would have to stop executing program commands while the marginal processed the data. This was deemed very inefficient. The first computer using a multiprogramming system was the British Leo III owned by J. Lyons and Co.. Several different programs in batch were imposed in the computer memory, and the first one began to run. When the first program reached an instruction waiting for a peripheral, the
context of this program was stored away, and the second program in memory was given a chance to run. The process continued until all programs finished running.

The use of multiprogramming was enhanced by the arrival of virtual memory and virtual machine technology, which enabled individual programs to make use of memory and operating system resources as if other concurrently running programs were, for all practical purposes, non-existent and invisible to them.

Multiprogramming doesn't give any guarantee that a program will run in a timely manner. Indeed, the very first program may very well run for hours without needing access to a peripheral. As there were no users waiting at an interactive terminal, this was no problem: users handed in a deck of punched cards to an operator, and came back a few hours later for printed results. Multiprogramming greatly reduced wait times when multiple batches were being processed.

### 1.2.5 Multithreading Operating System

As multitasking very much increased the throughput of computers, programmers started to apply applications as sets of cooperating processes (e.g., one process gathering input data, one process processing input data, one process writing out results on disk). This, however, involved some tools to allow processes to efficiently interchange data. Threads were born from the idea that the most systematic way for cooperating processes to exchange data would be to share their entire memory space. Thus, threads are mainly procedure that run in the same memory context. Threads are reported as lightweight process because switching between threads does not involve changing the memory context.

While threads are scheduled preemptively, some operating systems provide a alternative to threads, named fibers that are scheduled cooperatively. On operating systems that do not give fibers, an application may apply its own fibers using repeated calls to worker functions. Fibers are even extra lightweight than threads, and somewhat easier to program with, although they tend to lose some or all of the advantages of threads on processors. Some systems directly bear multithreading in hardware.
1.2.6 Multitasking/Time-Sharing Systems

Multiprogramming with two programs systems: several programs use the computer system

Time-sharing systems: several (human) users utilize the computer system interactively.

Characteristics

- Using multiprogramming to grasp multiple interactive jobs
- Processor’s time is divided among multiple users
- Multiple users simultaneously gain the system through terminals

Table 1-1 Multiprogramming vs. Time-Sharing Systems

<table>
<thead>
<tr>
<th></th>
<th>Batch Multiprogramming</th>
<th>Time Sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Principal objective</td>
<td>Maximize processor use</td>
<td>Minimize response time</td>
</tr>
<tr>
<td>Source of directives to</td>
<td>Job control language commands provided</td>
<td>Commands entered at the</td>
</tr>
<tr>
<td>operating system</td>
<td>with the job</td>
<td>terminal</td>
</tr>
</tbody>
</table>

Time sharing is multiprogramming. The key differences between time-sharing systems and batch multiprogramming systems are given in the table above.

Multitasking is a method to allow multiple processes to share processors (CPUs) and other system resources. Each CPU executes a single task at a time. However, multitasking allows each processor to switch between tasks that are being executed without having to wait for each task to finish. Depending on the operating system implementation, switches could be performed when tasks perform input/output operations, when a task indicates that it can be switched, or on hardware interrupts.

A common form of multitasking is time-sharing. Time-sharing is a method to allow fast response for interactive user applications. In time-sharing systems, context switches are
performed rapidly. This makes it seem like multiple processes are being executed simultaneously on the same processor. The execution of multiple processes seemingly simultaneously is called concurrency.

For security and reliability reasons most modern operating systems prevent direct communication between independent processes, providing strictly mediated and controlled inter-process communication functionality.

In general, a computer system process consists of (or is said to 'own') the following resources:

- An image of the executable machine code associated with a program.
- Memory (typically some region of virtual memory); which comprises the executable code, process-specific data (input and output), a call stack (to keep track of active subroutines and/or other events), and a heap to hold transitional computation data generated during run time.
- Operating system descriptors of resources that are allocated to the process, for example file descriptors (Unix terminology) or handles (Windows), and data sources and sinks.
- Security feature, such as the process owner and the process' set of permissions (allowable operations).
- Processor state (context), such as the content of registers, physical memory addressing, etc. The state is typically stored in computer registers when the process is executing and in memory otherwise.[40]

The operating system envelops most of this information about active processes in data structures called process control blocks.

Any collections of resource, but typically at least the processor state, may be associated with each of the process' threads in operating systems that help threads or 'daughter' processes.
The operating system keeps its processes disparate and allocate the resources they need, so that they are less likely to restrict with each other and cause system failures (e.g., deadlock or thrashing). The operating system may also give mechanisms for inter-process communication to enable processes to interact in safe and predictable ways.

1.2.6.1 Process management in multi-tasking operating systems

A multitasking operating system may just switch between processes to provide the appearance of many processes accomplishing concurrently or simultaneously, though in fact only one process can be accomplishing at any one time on a single-core CPU (unless using multithreading or other similar technology). [42]

It is usual to associate a single process with a main program, and 'daughter' ('child') processes with any spin-off, parallel processes, which behave like asynchronous subroutines. A process is said to own resources, of which an image of its program (in memory) is one such resource. (Note, however, that in multiprocessing systems, many processes may run off of, or share, the same reentrant program at the same location in memory - but each process is said to own its own image of the program.)

Processes are often known as "tasks" in embedded operating systems. The sense of "process" (or task) is "something that takes up time", as opposed to 'memory', which is "something that takes up space".[43]

The above explanation applies to both processes managed by an operating system, and processes as defined by process calculi.

If a process requires something for which it must wait, it will be blocked. When the process is in the Blocked State, it is eligible for exchanging to disk, but this is direct in a virtual memory system, where blocks of memory values may be really on disk and not in main memory at any time. Note that even unused portions of active processes/tasks (executing programs) are eligible for exchanging to disk. All parts of an executing program and its data do not have to be in physical memory for the associated process to be active.
1.2.6.2 Programming in a multitasking environment

Processes that are totally independent are not much trouble to program. Most of the complications in multitasking systems comes from the need to distribute computer resources between tasks and to synchronize the operation of co-operating tasks. Various synchronous computing techniques are used to avoid potential problems caused by multiple tasks attempting to access the same resource. In computing, multitasking is a method where multiple tasks, also known as processes, are completed during the same period of time. The tasks share common processing resources, such as a CPU and main memory. In the case of a computer with a single CPU, only one task is said to be running at any point in time, meaning that the CPU is actively performing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reallocating a CPU from one task to another one is called a context switch. When context switches occur frequently enough the illusion of parallelism is achieved. Even on computers with more than one CPU (called multiprocessor machines), multitasking allows many more tasks to be run than there are CPUs. The term "multitasking" has become an international term, as the same word in many other languages such as German, Italian, Dutch, Danish and Norwegian.

Bigger systems were sometimes built with a central processor(s) and some number of I/O processors, a kind of asymmetric multiprocessing. Over the years, multitasking systems have been refined. Modern operating systems generally include detailed mechanisms for prioritizing processes, while symmetric multiprocessing has introduced new complexities and capabilities.

1.2.6.3 Cooperative multitasking/time-sharing

The expression 'time sharing' was usually used to designate computers shared by interactive users at terminals, such as IBM's TSO, and VM/CMS. The term time-sharing is no longer commonly used, having been replaced by simply multitasking, and by the advent of personal computers and workstations rather than shared interactive systems. When computer usage evolved from batch mode to interactive mode, multiprogramming
was no longer a suitable approach. Each user wanted to see his program running as if it were the only program in the computer. The use of time sharing made this possible, with the qualification that the computer would not seem as fast to any one user as it really would be if it were running only that user's program. In time-sharing systems, the running task is required to relinquish the CPU, either voluntarily or by an external event such as a hardware interrupt. Time sharing systems are designed to allow several programs to execute apparently simultaneously. Early multitasking systems used applications that voluntarily ceded time to one another. This approach, which was eventually supported by many computer operating systems, is known today as cooperative multitasking. Although it is now rarely used in larger systems, cooperative multitasking was once the scheduling scheme employed by Microsoft Windows (prior to Windows 95 and Windows NT) and Mac OS (prior to Mac OS X) in order to enable multiple applications to be run simultaneously. Windows 9x also used cooperative multitasking, but only for 16-bit legacy applications, much the same way as pre-Leopard PowerPC versions of Mac OS X used it for Classic applications. The network operating system NetWare used cooperative multitasking up to NetWare 6.5. Cooperative multitasking is still used today on RISC OS systems.

Because a cooperatively multitasked system relies on each process regularly giving up time to other processes on the system, one poorly designed program can consume all of the CPU time for itself or cause the whole system to hang. In a server environment, this is a hazard that makes the entire network brittle and fragile. All software must be evaluated and cleared for use in a test environment before being installed on the main server or a misbehaving program on the server slows down or freezes the entire network.

Despite the difficulty of designing and implementing cooperatively multitasked systems, time-constrained, real-time embedded systems (such as spacecraft) are often implemented using this paradigm. This allows highly reliable, deterministic control of complex real time sequences, for instance, the firing of thrusters for deep space course corrections.
1.2.6.4 Preemptive multitasking/time-sharing

Preemptive multitasking allows the computer system to guarantee more reliably each process a regular "slice" of operating time. It also allows the system to deal rapidly with important external events like incoming data, which might require the immediate attention of one or another process.

Operating systems were developed to take advantage of these hardware capabilities and run multiple processes preemptively. Digital Equipment Corporation was a leader in this. For example, preemptive multitasking was implemented in the earliest version of Unix in 1969, and is standard in Unix and Unix-like operating systems, including Linux, Solaris and BSD with its derivatives.[40]

At any specific time, processes can be grouped into two categories: those that are waiting for input or output (called "I/O bound"), and those that are fully utilizing the CPU ("CPU bound"). In primitive systems, the software would often "poll", or "busy wait" while waiting for requested input (such as disk, keyboard or network input). During this time, the system was not performing useful work. With the advent of interrupts and preemptive multitasking, I/O bound processes could be "blocked", or put on hold, pending the occurrence of the necessary data, allowing other processes to utilize the CPU. As the occurrence of the requested data would generate an interrupt, blocked processes could be guaranteed a timely return to execution.

The earliest preemptive multitasking OS available to home users was Sinclair QDOS on the Sinclair QL, released in 1984, but very few people bought the machine. Commodore's powerful Amiga, released the following year, was the first commercially successful home computer to use the technology, and its multimedia abilities make it a clear ancestor of contemporary multitasking personal computers. Microsoft made preemptive multitasking a core feature of their flagship operating system in the early 1990s when developing Windows NT 3.1 and then Windows 95. It was later adopted on the Apple Macintosh by Mac OS 9.x [2] as an additional API, i.e. the application could be programmed to use the preemptive or cooperative model, and all legacy applications were multitasked cooperatively within a single process. Mac OS X, being a Unix-like system, uses
preemptive multitasking for all native applications, although Classic applications are
multitasked cooperatively in a Mac OS 9 environment that itself is running as an OS X
process (and is subject to preemption like any other OS X process).

A similar model is used in Windows 9x and the Windows NT family, where native 32-bit
applications are multitasked preemptively, and legacy 16-bit Windows 3.x programs are
multitasked cooperatively within a single process, although in the NT family it is possible
to force a 16-bit application to run as a separate preemptively multitasked process.[3] 64-bit
editions of Windows, both for the x86-64 and Itanium architectures, no longer provide
support for legacy 16-bit applications, and thus provide preemptive multitasking for all
supported applications.

1.2.7 Desktop Systems

Personal computers PCs appeared in the 1970s. During their first decade, the CPUs in
PCs lacked the features needed to protect an operating system from user programs. PC
operating systems therefore were neither multiuser nor multitasking. However, the goals
of these operating systems have changed with time; instead of maximizing CPU and
peripheral utilization, the systems opt for maximizing user convenience and
responsiveness. These systems include PCs running Microsoft Windows and the Apple
Macintosh. The MS-DOS operating system from Microsoft has been superseded by
multiple flavors of Microsoft Windows, and IBM has upgraded MS-DOS to the OS/2
multitasking system.

The Apple Macintosh operating system has been ported to more advanced hardware, and
now comprises new features, such as virtual memory and multitasking.

With the release of MacOS X, the core of the operating system is now based on Mach
and FreeBSD UNIX for scalability, performance, and features, but it maintains the same
rich GUI. Linux, a UNIX-like operating system available for PCs, has also become
popular recently.

Operating systems for these computers have benefited in several ways from the
development of operating systems for mainframes. Microcomputers were immediately
able to adopt some of the technology developed for larger operating systems. On the
other hand, the hardware costs for microcomputers are sufficiently low that individuals have sole use of the computer, and CPU utilization is no longer a prime concern. Thus, some of the design decisions made in operating systems for mainframes may not be appropriate for smaller systems.

Other design decisions still apply. For example, file protection was, at first, not necessary on a personal machine. However, these computers are now often tied into other computers over local-area networks or other Internet connections.

When other computers and other users can access the files on a PC, file protection again becomes a necessary feature of the operating system. The lack of such protection has made it easy for malicious programs to destroy data on systems such as MS-DOS and the Macintosh operating system. These programs may be self-replicating, and may spread rapidly via worm or virus mechanisms and disrupt entire companies or even worldwide networks. Advanced timesharing features such as protected memory and file permissions are not enough, on their own, to safeguard a system from attack. Recent security breaches have shown that time and again.

1.2.8 Multiprocessor Systems

Most systems to date are single-processor systems; that is, they have only one main CPU. However, multiprocessor systems (also known as parallel systems or tightly coupled systems) are growing in importance. Such systems have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices.

Multiprocessor systems have three main advantages.

1. Increased throughput: By increasing the number of processors, we hope to get more work done in less time. The speed-up ratio with N processors is not N; rather, it is less than N. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors. Similarly, a group of N programmers working closely together does not result in N times the amount of work being accomplished.
2. Economy of scale: Multiprocessor systems can save more money than multiple single-processor systems, because they can share peripherals, mass storage, and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them, than to have many computers with local disks and many copies of the data.

3. Increased Reliability: If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors must pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether. This ability to continue providing service proportional to the level of surviving hardware is called graceful degradation. Systems designed for graceful degradation are also called fault tolerant.

Continued operation in the presence of failures requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected. The Tandem system uses both hardware and software duplication to ensure continued operation despite faults. The system consists of two identical processors, each with its own local memory. The processors are connected by a bus. One processor is the primary and the other is the backup. Two copies are kept of each process: one on the primary processor and the other on the backup. At fixed Check points in the execution of the system, the state information of each job including a copy of the memory image is copied from the primary machine to the backup. If a failure is detected, the backup copy is activated and is restarted from the most recent checkpoint. This solution is expensive, since it involves considerable hardware duplication.

The most common multiple-processor systems now use symmetric multiprocessing (SMP), in which each processor runs an identical copy of the operating system, and these copies communicate with one another as needed.

Some systems use asymmetric multiprocessing, in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to
the master for instruction or have predefined tasks. This scheme defines a master-slave relationship. The master processor schedules and allocates work to the slave processors.

SMP means that all processors are peers; no master-slave relationship exists between processors. Each processor concurrently runs a copy of the operating system. An example of the SMP system is Encore's version of UNIX for the Multimax computer. This computer can be configured such that it employs dozens of processors, all running copies of UNIX. The benefit of this model is that many processes can run simultaneously—N processes can run if there are N CPUs—without causing a significant deterioration of performance. However, we must carefully control I/O to ensure that the data reach the appropriate processor. Also, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. These inefficiencies can be avoided if the processors share certain data structures. A multiprocessor system of this form will allow processes and resources—such as memory—to be shared dynamically among the various processors, and can lower the variance among the processors. Virtually all modern operating systems—including Windows NT, Solaris, Digital UNIX, OS/2, and Linux—now provide support for SMP.

The difference between symmetric and asymmetric multiprocessing may be the result of either hardware or software. Special hardware can differentiate the multiple processors, or the software can be written to allow only one master and multiple slaves. For instance, Sun's operating system SunOS Version 4 provides asymmetric multiprocessing, whereas Version 5 (Solaris 2) is symmetric on the same hardware.

As microprocessors become less expensive and more powerful, additional operating-system functions are off-loaded to slave processors (or back-ends).

For example, it is fairly easy to add a microprocessor with its own memory to manage a disk system. The microprocessor could receive a sequence of requests from the main CPU and implement its own disk queue and scheduling algorithm. This arrangement relieves the main CPU of the overhead of disk scheduling. PCs contain a microprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU. In fact, this
use of microprocessors has become so common that it is no longer considered multiprocessing.

1.2.9 Distributed Systems

A network, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. By being able to communicate, distributed systems are able to share computational tasks, and provide a rich set of features to users.

Networks vary by the protocols used, the distances between nodes, and the transport media. TCP/IP is the most common network protocol, although ATM and other protocols are in widespread use. Likewise, operating-system support of protocols varies. Most operating systems support TCP/IP, including the Windows and UNIX operating systems. Some systems support proprietary protocols to suit their needs. To an operating system, a network protocol simply needs an interface device—a network adapter, for example—with a device driver to manage it, and software to package data in the communications protocol to send it and to unpack it to receive it. These concepts are discussed throughout the book.

Networks are typecast based on the distances between their nodes. A local-area network (LAN), exists within a room, a floor, or a building. A wide-area network (WAN) usually exists between buildings, cities, or countries. A global company may have a WAN to connect its offices, worldwide.

These networks could run one protocol or several protocols. The continuing advent of new technologies brings about new forms of networks. For example, a metropolitan-area network (MAN), could link buildings within a city.

BlueTooth devices communicate over a short distance of several feet, in essence creating a small-area network.

The media to carry networks are equally varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios. When computing devices are connected to cellular phones, they create a network. Even very
short-range infrared communication can be use for networking. At a rudimentary level, whenever computers communicate they use or create a network. These networks also vary by their performance and reliability.

1.2.9.1 **Client-Server Systems**

As PCs have become faster, more powerful, and cheaper, designers have shifted away from the centralized system architecture. Terminals connected to centralized systems are now being supplanted by PCs. Correspondingly; user-interface functionality that used to be handled directly by the centralized systems is increasingly being handled by the PCs. As a result, centralized systems today act as server systems to satisfy requests generated by client systems. Server systems can be broadly categorized as compute servers and file servers.

Compute-server systems provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client.

File-server systems provide a file-system interface where clients can create, update, read, and delete files.

1.2.9.2 **Peer-to-Peer Systems**

The growth of computer networks-especially the Internet and World Wide Web (WWW) - has had a profound influence on the recent development of operating systems. When PCs were introduced in the 1970s, they were designed for "personal" use and were generally considered standalone computers. With the beginning of widespread public use of the Internet in the 1980s for electronic mail, ftp, and gopher, many PCs became connected to computer networks. With the introduction of the Web in the mid-1990s, network connectivity became an essential component of a computer system.

Virtually all modern PCs and workstations are capable of running a web browser for accessing hypertext documents on the Web. Operating systems (such as Windows, OS/2, MacOS, and UNIX) now also include the system software (such as TCP/IP and PPP) that enables a computer to access the Internet via a local-area network or telephone
connection. Several include the web browser itself, as well as electronic mail, remote login, and file-transfer clients and servers.

In contrast to the tightly coupled systems, the computer networks used in these applications consist of a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication lines, such as high-speed buses or telephone lines. These systems are usually referred to as loosely coupled systems (or distributed systems).

Some operating systems have taken the concept of networks and distributed systems further than the notion of providing network connectivity. A network operating system is an operating system that provides features such as file sharing across the network, and that comprises a communication scheme that allows different processes on different computers to exchange messages. A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers. A distributed operating system is a less autonomous environment: The different operating systems communicate closely enough to provide the illusion that only a single operating system controls the network.

1.2.10 Clustered Systems

Like parallel systems, clustered systems gather together multiple CPUs to accomplish computational work. Clustered systems differ from parallel systems, however, in that they are composed of two or more individual systems coupled together. The definition of the term clustered is not concrete; many commercial packages wrestle with what a clustered system is, and why one form is better than another. The generally accepted definition is that clustered computers share storage and is closely associated via LAN networking.

Clustering is usually performed to provide high availability. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the LAN). If the monitored machine fails, the monitoring machine can take ownership of its
storage, and restart the application(s) that were running on the failed machine. The failed machine can remain down, but the users and clients of the application would only see a brief interruption of service.

In asymmetric clustering, one machine is in hot standby mode while the other is running the applications. The hot standby host (machine) does nothing but monitor the active server. If that server fails, the hot standby host becomes the active server. In symmetric mode, two or more hosts are running applications, and they are monitoring each other. This mode is obviously more efficient, as it uses all of the available hardware. It does require that more than one application be available to run.

Other forms of clusters include parallel clusters and clustering over a WAN. Parallel clusters allow multiple hosts to access the same data on the shared storage. Because most operating systems lack support for this simultaneous data access by multiple hosts, parallel clusters are usually accomplished by special versions of software and special releases of applications. For example, Oracle Parallel Server is a version of Oracle's database that has been designed to run on parallel clusters. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database.

In spite of improvements in distributed computing, most systems do not offer general-purpose distributed file systems. Therefore, most clusters do not allow shared access to data on the disk. For this, distributed file systems must provide access control and locking to the files to ensure no conflicting operations occur. This type of service is commonly known as a distributed lock manager (DLM). Work is ongoing for general-purpose distributed file systems, with vendors like Sun Microsystems announcing roadmaps for delivery of a DLM within the operating system.

Cluster technology is rapidly changing. Cluster directions include global clusters, in which the machines could be anywhere in the world (or anywhere a WAN reaches). Such projects are still the subject of research and development.

Clustered system use and features should expand greatly as storage-area networks (SANS), become prevalent. SANs allow easy attachment of multiple hosts to multiple
storage units. Current clusters are usually limited to two or four hosts due to the complexity of connecting the hosts to shared storage.

1.2.11 Real-Time Systems

In real-time systems, some waiting tasks are guaranteed to be given the CPU when an external event occurs. Real time systems are designed to control mechanical devices such as industrial robots, which require timely processing.

Another reason for multitasking was in the design of real-time computing systems, where there are a number of possibly unrelated external activities needed to be controlled by a single processor system. In such systems a hierarchical interrupt system is coupled with process prioritization to ensure that key activities were given a greater share of available process time.

Another form of a special-purpose operating system is the real-time system. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real-time systems.

Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

A real-time system has well-defined, fixed time constraints. Processing must be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt after it had smashed into the car it was building. A real-time system functions correctly only if it returns the correct result within its time constraints. Contrast this requirement to a time-sharing system, where it is desirable (but not mandatory) to respond quickly, or to a batch system, which may have no time constraints at all.
Real-time systems come in two flavors: hard and soft. A hard real-time system guarantees that critical tasks be completed on time. This goal requires that all delays in the system be bounded, from the retrieval of stored data to the time that it takes the operating system to finish any request made of it. Such time constraints dictate the facilities that are available in hard real-time systems.

A Secondary storage of any sort is usually limited or missing, with data instead being stored in short-term memory or in read-only memory (ROM). ROM is located on nonvolatile storage devices that maintain their contents even in the case of electric outage; most other types of memory are volatile. Most advanced operating-system features are absent too, since they tend to separate the user from the hardware, and that separation results in uncertainty about the amount of time an operation will take. Therefore, hard real-time systems conflict with the operation of time-sharing systems, and the two cannot be mixed. Since none of the existing general-purpose operating systems support hard real-time functionality, we do not concern ourselves with this type of system in this text.

A less restrictive type of real-time system is a soft real-time system, where a critical real-time task gets priority over other tasks, and maintains that priority until it completes. As in hard real-time systems, the operating-system kernel delays need to be bounded: A real-time task cannot be kept waiting indefinitely for the kernel to run it. Soft real time is an achievable goal that can be mixed with other types of systems. Soft real-time systems, however, have more limited utility than hard real-time systems. Given their lack of deadline support, they are risky to use for industrial control and robotics. They are useful, however in several areas, including multimedia, virtual reality, and advanced scientific projects such as undersea exploration and planetary rovers. These systems need advanced operating-system features that cannot be supported by hard real-time systems. Because of the expanded uses for soft real-time functionality, it is finding its way into most current operating systems, including major versions of UNIX.
1.2.12 Handheld Systems

Handheld systems include personal digital assistants (PDAs), such as Palm Pilots or cellular telephones with connectivity to a network such as the Internet.

Developers of handheld systems and applications face many challenges, most of which are due to the limited size of such devices. For example, a PDA is typically about 5 inches in height and 3 inches in width, and it weighs less than one-half pound. Due to this limited size, most handheld devices have a small amount of memory, include slow processors, and feature small display screens.

Limitations

- Many handheld devices have between 512 KB and 8 MB of memory. (Contrast this with a typical PC or workstation, which may have several hundred megabytes of memory) As a result, the operating system and applications must manage memory efficiently. This comprises returning all allocated memory back to the memory manager once the memory is no longer being used. Virtual memory, which allows developers to write programs that behave as if the system has more memory than may be physically available.

- Currently, many handheld devices do not use virtual memory techniques, thus forcing program developers to work within the confines of limited physical memory.

- A second issue of concern to developers of handheld devices is the speed of the processor used in the device. Processors for most handheld devices often run at a fraction of the speed of a processor in a PC. Faster processors require more power. To include a faster processor in a handheld device would require a larger battery that would have to be replaced (or recharged) more frequently. To minimize the size of most handheld devices, smaller, slower processors which consume less power are typically used. Therefore, the operating system and applications must be designed not to tax the processor.
The last issue confronting program designers for handheld devices is the small display screens typically available. Whereas a monitor for a home computer may measure up to 21 inches, the display for a handheld device is often no more than 3 inches square. Familiar tasks, such as reading e-mail or browsing web pages, must be condensed onto smaller displays. One approach for displaying the content in web pages is web clipping, where only a small subset of a web page is delivered and displayed on the handheld device. Some handheld devices may use wireless technology, such as BlueTooth allowing remote access to e-mail and web browsing. Cellular telephones with connectivity to the Internet fall into this category. However, many PDAs currently do not provide wireless access. To download data to these devices, typically one first downloads the data to a PC or workstation, and then downloads the data to the PDA. Some PDAs allow data to be directly copied from one device to another using an infrared link. Generally, the limitations in the functionality of PDAs are balanced by their convenience and portability.

Their use continues to expand as network connections become more available and other options, such as cameras and MP3 players, expand their utility.

1.3 Major roles of operating system

Five major roles of operating system are as below:

- Processes Management
- Memory Management
- Information protection and security
- Scheduling and resource management
- System structure

1.3.1 Processes Management

- Process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system
(OS), a process may be made up of multiple threads of execution that execute instructions concurrently. A computer program is a passive collection of instructions; a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the same program often means more than one process is being executed or a unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources as shown in figure 1.5.

Figure 1-5 Typical Process Implementation
1.3.1.1 Difficulties with designing system software

- Improper Synchronization: Communication between processes takes place by calls to send and receive primitives. There are different design options for implementing each primitive. Message passing may be either blocking or non-blocking also known as synchronous and asynchronous. Blocking send: The sending process is blocked until the message is received by the receiving process or by the mailbox. Non blocking send: The sending process sends the message and resumes operation.

Blocking receive: The receiver blocks until a message is available.

No- blocking Receive: The receiver retrieves either a valid message or a null. Different combinations of send and receive are possible. When both the send and receive are blocking, we have a rendezvous between the sender and the receiver. Results from an improper design of a signaling mechanism can result in duplicate signals, or lost signals.

- Nondeterministic program operation: a program may change the contents of memory used by another program, and in this way affect unpredictably the operation of the other program.

- Deadlock: In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock. A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual exclusion: At least one resource must be held in a non-sharable mode; i.e. only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. Hold and wait: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. No preemption: Resources cannot be preempted; i.e. a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular wait: A set \( \{P_0, P_1, \ldots, P_n\} \) of waiting processes must exist 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 \( P_2 \), \ldots, \( P_{n-1} \) is waiting for a resource that is held by \( P_n \), and \( P_n \) is waiting for a resource that is held by \( P_0 \).

**1.3.1.2 Solution to the problems with designing system software**

Use the concept of a process that consists of

- An executable program
- Associated data needed by the program
- Execution context of the program: All information the operating system needs to manage the process.

**1.3.2 Memory Management**

Memory management is the function of a computer operating system responsible for managing the computer's primary memory.

The memory management function keeps track of the status of each memory location, either allocated or free. It determines how memory is allocated among competing processes, deciding who gets memory, when they receive it, and how much they are allowed. When memory is allocated it determines which memory locations will be assigned. It tracks when memory is freed or unallocated and updates the status.
1.3.2.1 Memory protection

When multiple programs are present in memory, an ill-behaved program may (inadvertently or deliberately) overwrite memory belonging to another program, or even to the operating system itself.

The operating system therefore restricts the memory accessible to the running program. A program trying to access memory outside its allowed range is immediately stopped before it can change memory belonging to another process. Another key innovation was the idea of privilege levels. Low privilege tasks are not allowed some kinds of memory access and are not allowed to perform certain instructions. When a task tries to perform a privileged operation a trap occurs and a supervisory program running at a higher level is allowed to decide how to respond.

1.3.2.2 Memory swapping

Use of a swap file or swap partition is a way for the operating system to provide more memory than is physically available by keeping portions of the primary memory in secondary storage. While multitasking and memory swapping are two completely unrelated techniques, they are very often used together, as swapping memory allows more tasks to be loaded at the same time. Typically, a multitasking system allows another process to run when the running process hits a point where it has to wait for some portion of memory to be reloaded from secondary storage.

1.3.2.3 Memory management techniques

1.3.2.3.1 Single contiguous allocation

Single allocation is the simplest memory management technique. Computer's full memory, usually with the exception of a small portion reserved for the operating system, is available to the single application. MS-DOS is an example of a system which allocates memory in this way. An embedded system running a single application might also use this technique.
A system using single contiguous allocation may still multitask by swapping the contents of memory to switch among users. Early versions of the Music operating system used this technique.

### 1.3.2.3.2 Partitioned allocation

Partitioned allocation divides primary memory into multiple memory partitions, usually contiguous areas of memory. Each partition might contain all the information for a specific job or task. Memory management consists of allocating a partition to a job when it starts and unallocating it when the job ends.

Partitioned allocation usually requires some hardware support to prevent the jobs from interfering with one another or with the operating system. The IBM System/360 used a lock-and-key technique. Other systems used base and bounds registers which contained the limits of the partition and flagged invalid accesses. The UNIVAC 1108 Storage Limits Register had separate base/bound sets for instructions and data. The system took advantage of memory interleaving to place what were called the i bank and d bank in separate memory modules.

Partitions may be either static, that is defined at Initial Program Load (IPL) or boot time or by the computer operator, or dynamic, that is automatically created for a specific job. IBM System/360 Operating System Multiprogramming with a Fixed Number of Tasks (MFT) is an example of static partitioning, and Multiprogramming with a Variable Number of Tasks (MVT) is an example of dynamic. MVT and successors use the term region to distinguish dynamic partitions from static ones in other systems.

Partitions may be re-locatable using hardware typed memory, like the Burroughs Corporation B5500, or base and bounds registers like the PDP-10 or GE-635. Re-locatable partitions are able to be compacted to provide larger chunks of contiguous physical memory. Compaction moves "in-use" areas of memory to eliminate "holes" or unused areas of memory caused by process termination in order to create larger contiguous free areas.
Some systems allow partitions to be swapped out to secondary storage to free additional memory. Early versions of IBM's Time Sharing Option (TSO) swapped users in and out of a single time-sharing partition.

1.3.2.3.3 Paged memory management

Paged allocation divides the computer's primary memory into fixed-size units called page frames, and the program's address space into pages of the same size. The hardware memory management unit maps pages to frames. The physical memory can be allocated on a page basis while the address space appears contiguos.

Usually, with paged memory management, each job runs in its own address space, however, IBM OS/VS/2 SVS ran all jobs in a single 16MiB virtual address space.

Paged memory can be demand-paged when the system can move pages as required between primary and secondary memory.

1.3.2.3.4 Segmented memory management

Segmented memory is the only memory management technique that does not provide the user's program with a 'linear and contiguous address space.' Segments are areas of memory that usually correspond to a logical grouping of information such as a code procedure or a data array. Segments require hardware support in the form of a segment table which usually contains the physical address of the segment in memory, its size, and other data such as access protection bits and status (swapped in, swapped out, etc.)

Segmentation allows better access protection than other schemes because memory references are relative to a specific segment and the hardware will not permit the application to reference memory not defined for that segment.

It is possible to implement segmentation with or without paging. Without paging support the segment is the physical unit swapped in and out of memory if required. With paging support the pages are usually the unit of swapping and segmentation only adds an additional level of security.
Addresses in a segmented system usually consist of the segment id and an offset relative to the segment base address, defined to be offset zero.

The Intel IA-32 (x86) architecture allows a process to have up to 16,383 segments of up to 4GiB each. IA-32 segments are subdivisions of the computer's linear address space, the virtual address space provided by the paging hardware.

The Multics operating system is probably the best known system implementing segmented memory. Multics segments are subdivisions of the computer's physical memory of up to 256 pages, each page being 1K 36-bit words in size, resulting in a maximum segment size of 1MiB (with 9-bit bytes, as used in Multics). A process could have up to 4046 segments.

1.3.2.3.5 Virtual memory and paging

- Allows programmers to address memory from a logical point of view
- Allows process to be comprised of a number of fixed-size blocks, called pages
- Virtual address is a page number and an offset within the page
- Each page may be located anywhere in main memory
- Dynamic mapping between the virtual address used in the program and the real address in main memory

1.3.3 File System Management

The file system is the most visible aspect of an operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system. Some file systems have a third part, partitions, which are used to separate physically or logically large collections of directories.[1]
1.3.4 Information protection and security

Computer protection and security mechanisms provided by an operating system must address the following requirements:

- Confidentiality: (or privacy) the requirement that information maintained, by a computer system be accessible only by authorized parties (users and the processes that run as/represent those users). Interception occurs when an un-authorised party gains access to a resource; examples include illicit file copying and the invocation of programs.

- Integrity: the requirement that a computer system’s resources can be modified only by authorized parties. Modification occurs when an unauthorized party not only gains access to but changes a resource such as data or the execution of a running process.

- Availability: the requirement that a computer system be accessible at required times by authorized parties. Interruption occurs when an unauthorized party reduces the availability of or to a resource.

- Authenticity: the requirement that a computer system can verify the identity of a user. Fabrication occurs when an unauthorized party inserts counterfeit data amongst valid data.[47]

1.3.4.1 Assets and their Vulnerabilities

Hardware is mainly vulnerable to interruption, either by theft or by vandalism. Physical security measures are used to prevent these attacks.

Software is also vulnerable to interruption, as it is very easy to delete. Backups are used to limit the damage caused by deletion. Modification or fabrication through alteration (e.g. by viruses) is a major problem, as it can be hard to spot quickly. Software is also vulnerable to interception through unauthorized copying: this problem is still largely unsolved.
Data is vulnerable in many ways. Interruption can occur through the simple destruction of data files. Interception can occur through unauthorized reading of data files, or more perniciously through unauthorized analysis and aggregation of data. Modification and fabrication are also obvious problems with potentially huge consequences.

Communications are vulnerable to all types of threats.

Passive attacks take the form of eavesdropping, and fall into two categories: reading the contents of a message, or more subtly, analyzing patterns of traffic to infer the nature of even secure messages. Passive attacks are hard to detect, so the emphasis is usually on prevention.

Active attacks involve modification of a data stream, or creation of a false data stream. One entity may masquerade as another (presumably one with more or different privileges), maybe by capturing and replaying an authentication sequence.

Replay is a similar attack, usually on data. Message contents may also be modified, often to induce incorrect behavior in other users.

Denials of service attacks aim to inhibit the normal use of communication facilities. Active attacks are hard to prevent (entirely), so the emphasis is usually on detection and damage control.[47]

**Protection**

Multiprogramming involves the sharing of many resources, including processor, memory, I/O devices, programs, and data. Protection of such resources runs along the following spectrum:

- No protection may be adequate e.g. if sensitive procedures are run at separate times.

- Isolation implies that entities operate separately from each other in the physical sense.

- Share all or nothing implies that an object is either totally private or totally public.
• Share via access limitation implies that different entities enjoy different levels of access to an object, at the gift of the owner. The OS acts as a guard between entities and objects to enforce correct access.

• Share via dynamic capabilities extends the former to allow rights to be varied dynamically.

• Limit use of an object implies that not only is access to the object controlled, the use to which it may be put also varies across entities.

• The above spectrum is listed roughly in order of increasing fineness of control for owners, and also increasing difficulty of implementation.

**Intruders**

Intruders and viruses are the two most publicized security threats. We identify three classes of intruders:

1. A masquerador is an unauthorized individual (an outsider) who penetrates a system to exploit legitimate users’ accounts.

2. A misfeasor is a legitimate user (an insider) who accesses resources to which they are not privileged, or who abuses such privilege.

3. A clandestine user is an individual (an insider or an outsider) who seizes control of a system to evade auditing controls, or to suppress audit collection.

Intruders are usually trying to gain access to a system, or to increased privileges to which they are not entitled, often by obtaining the password for a legitimate account. Many methods of obtaining passwords have been tried:

• Trying default passwords;

• Exhaustively testing short passwords;

• Trying words from a dictionary, or from a list of common passwords;

• Collecting personal information about users;
• Using a Trojan horse;

• Eavesdropping on communication lines.

The usual methods for protecting passwords are through one-way encryption, or by limiting access to password files. However, passwords are inherently vulnerable.

1.3.4.2 Malicious program

The most sophisticated threats to computer systems are through malicious software, sometimes called malware. Malware attempts to cause damage to, or consume the resources of, a target system. Malware can be divided into programs that can operate independently, and those that need a host program; and also into programs that can replicate themselves, and those that cannot.

• A trap door is a secret entry point into a program, often left by the program’s developers, or sometimes delivered via a software update.

• A logic bomb is code embedded in a program that ”explodes” when certain conditions are met, e.g. a certain date or the presence of certain files or users. Logic bombs also often originate with the developers of the software.

• A Trojan horse is a functional (or apparently useful) program that contains hidden code to perform some unwanted or harmful function.

• A virus is a program that can “infect” other programs by alteration, as well as causing local damage. Such modification comprises a copy of the virus, which can then spread further to other programs.

• A worm is an independent program that grows via network connections, typically using email, remote execution, or remote login to deliver or accomplish a copy of itself to or on another system, as well as causing local damage.

• A zombie is an self-governing program that secretly takes over a system and uses that system to launch attacks on other systems, thus hiding the original instigator.
Such attacks often require further replication of the zombie itself. Zombies are generally used in denial-of-service attacks.

The last three of these include replication. In all cases, prevention is much easier than detection and recovery.

1.3.4.3 Trusted Systems

So far we have discussed protecting a given resource from attack by a given user. Another necessity is to protect a resource on the basis of levels of security, e.g. the military-style system, where users are allowed clearance to view certain categories of data.

This is known as multi-level security. The basic principle is that a subject at a higher level may not transfer information to a subject at a lower level against the wishes of the authorized user. This principle has two facts:

1. No read-up involves that a subject can only read objects of less or equal security level.

2. No write-down involves that a subject can only write objects of greater or equal security level.

These requirements are implemented by a reference monitor, which are having three roles:

1. Complete mediation involves that rules are imposed on every access.

2. Isolation involves that the monitor and database are protected from unauthorized modification.

3. Verifiability involves that the monitor is probably correct.

Such a system is called as a trusted system. These requirements are very hard to meet, both in terms of assuring correctness and in terms of delivering adequate performance.
Protection and Security Design Principles

Saltzer and Schroeder discovered a core set of principles to operating system security design:

- Least privilege: Every object (users and their processes) should work within a minimum set of privileges; access rights should be obtained by explicit request, and the default level of access should be “none”.

- Economy of mechanisms: security mechanisms should be as small and simple as possible, assisting in their verification. This involves that they should be integral to an operating system’s design, and not an afterthought.

- Acceptability: security method must at the same time be robust yet non-intrusive. An intrusive method is likely to be counter-productive and avoided by users, if possible.

- Complete: Technique must be pervasive and access control checked during all operations — including the tasks of backup and maintenance.

- Open design: An operating system’s security should not remain secret, nor be produce by stealth. Open method are subject to scrutiny, review, and continued refinement.[47][48]

1.3.4.4 The Unix/Linux Security Model

UNIX in comparison to more modern operating systems like Windows - NT, provides a relatively simple model of security.

System calls are the only means by which processes may interact with the operating system and the resources it is preserving and managing. Each user and each process performed on behalf of that user, is identified by (minimally) two non-negative 16-bit integers: The user-identifier is fixed when logging into a Unix system. A correct union of user-name and password when logging in, or the validation of a network-based connection, set the user-identifier (uid) in the process control block of the user’s login shell, or command interpreter. Unless altered, this user-identifier is received by all
processes invoked from the initial login shell. Under definite conditions, the user-
identifier may be changed and determined with the system calls setuid() and getuid(). The
effective user-identifier is, by default, the same as the user-identifier, but may be
temporarily changed to a different value to provide temporary privileges.

The successful invocation of set-user-id programs, for example Password and login will,
typically, set the effective user-identifier for the lifetime of that process. Under certain
conditions, the effective user-identifier may be changed and determined with the system
calls seteuid() and geteuid().

1.3.4.5 Properties of the UNIX Super user

Unix uses the special user id value of 0 to present its only special user, the super user(or
root). Processes acting on behalf of the super user can generally access additional
resources (often incorrectly stated as “everything”) because most system calls authoritive
for checking user permissions bypass their checks if invoked with userid = 0. The result
is that there seem to be no files, etc, that cannot be accessed from the super user utilizing
the standard application programs which report and manipulate such resources. Instead,
attackers must attempt to hide additional or modified files utilizing other techniques
which typically exploit social engineering issues — playing on human nature to overlook
or insufficiently examine for problems.

Although the superuser has greater ingress to otherwise protected resources, the Unix
kernel will not allow the superuser to undermine the integrity of the operating system
itself. For example, although the superuser can generate a new file in any directory
through a call to the open() or creat() system calls, and descriptions of this new file are
written to the directory by the kernel itself, the superuser cannot open and clearly write to
the directory. Unix is frequently censured for both having a concept such as a superuser,
or for encouraging security practices which now depended on it.

It is thus the single strongest target of attack on a Unix system.
### 1.3.4.6 The UNIX Security Model - Groups

Each process is also recognised by a primary group identifier and a list of up to 32 (see<asm/limits.h>) secondary group identifiers. Under Linux, each classification identifier is a non-negative 16-bit integer. The unlimited membership of each classification consists of user identifiers. As with user identification, the login procedures construct the primary and secondary groups of the user’s login shell and these group identifiers are inherited by all processes invoked from the initial login shell. Each process also has a single essential group identifier, which may be set by privileged programs or by supplicating set-group-id programs. The system calls of interest here are setgid(), getgid(), seteuid() , and geteuid().

Information about the array of 32 secondary groups is set/read with setgroups() and getgroups().

### 1.3.4.7 Protection For Unix Files and Directories

The use of user identifiers and group identifiers under Linux is most observable with regard to file system access. All files and directories have certain ingress permissions which constrain ingress to only those users having the correct user and group permissions.

Let’s consider a typical instance, running/bin/ls -l:rw--r--r-- 1 chris staff 8362 Oct 17 12:40 /tmp/textfile. The first character of the ingress permissions indicates what type of file is being displayed.

- plain files (such as Java and C source code)
- directories
- character special files (such as terminals)
- block special files (such as disk drives)
- named pipes (FIFOs)
Each of the following three triples reports, from left to right, the read, write, and accomplishes permission for (respectively) the owner, the file’s group, and the “rest-of-the-world”. Each file and directory is granted by a single user, and considered to be “in” a single group (the owner does not have to be in that group). The UID and GID may be acquired via the stat system call.

1.3.4.8 The Meaning of Permissions

Read permission means that the file may be displayed on the screen, copied to another file or printed on the printer — any operation which needs reading the contents of the file. Having read permission on a directory means that its contents may be listed — ls may read the file’s names (and attributes).

Write permission means that the file or directory may be altered, changed or overwritten. Most significantly, write permission means that a file may be deleted. Write permission on a directory allows permission to delete a file from within that directory, if the permission also exists for the file.

Execute permission means that the file may be executed. Execute permission for a directory means that the user may alter into that directory.

Shellscripts must have both read and execute permission — zsh must both be able to read the shellscript and know to execute it.

Annoyingly, on dissimilar variants of Unix/Linux the permission mode bits, in combination, have some obscure meanings:

- having execute access, but not read access, to a directory still allows someone to “guess” filenames therein,

- having the sticky bit set on a directory permits only the owner of a file therein to remove or alter the file,

- having the setgid bit set on a directory means that files created in the directory collect the groupid of the directory, and not of their creator (owner).
1.3.4.9 Changing File and Directory Permissions

The permissions on files and directories may be changed with chmod, standing for “change mode”. Chmod is both the name of a command and the name of a system call.

```c
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
```

- Only the three permission triplets may be altered — a directory cannot be changed into a plain file or vice-versa.
- Permissions provided to the Chmod command can be either absolute or relative.
- All triplet is the sum of the octal digits 4, 2 and 1, read from left to right.

For example rwx is represented by 7, rw- by 6 and r– by 4, and so on.

**Table 1-2 Permission on any file or directory**

<table>
<thead>
<tr>
<th>Octal value</th>
<th>Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>Read by owner</td>
</tr>
<tr>
<td>200</td>
<td>Write (delete) by owner</td>
</tr>
<tr>
<td>100</td>
<td>Execute (search in directory) by owner</td>
</tr>
<tr>
<td>040</td>
<td>Read by group</td>
</tr>
<tr>
<td>020</td>
<td>Write (delete) by group</td>
</tr>
<tr>
<td>010</td>
<td>Execute (search) by group</td>
</tr>
<tr>
<td>004</td>
<td>Read by others</td>
</tr>
<tr>
<td>002</td>
<td>Write (delete) by others</td>
</tr>
<tr>
<td>001</td>
<td>Execute (search) by others</td>
</tr>
</tbody>
</table>
The entire permission on any file or directory is the “sum” of the appropriate values.

- Home directories are typically 700 which provide you (the owner) read, write and execute (search) permission but denies all entrance by others:

  chmod 700 /home/year2/charles-p

- If you wish others to read some of your files, set their mode to 644:

  chmod 644 funny.jokes

- Alternatively, we can set a file’s permission bits using a symbolic notation:

  chmod u+rwx /home/year2/charles-p

  chmod u+rw,g+r,o+r funny.jokes

  chmod u+w,a+r funny.jokes

- Programatically, we can set a file’s protection mode when it is originally created, and change it thereafter:

  fd = open ("myfile", O_RDWR | O_CREAT, 0600);

  ...

  (void) chmod ("myfile", 0644);

1.3.4.10 The Windows-NT Security Model

While the Unix security model gives system-wide and consistent relief of user and group identification, it constrains their exploitation to the system administrator (root). In contrast, the newer Windows-NT security model enables each permitted user (and process) to both examine and exploit access to a variety of objects. Again, the ingress controls provided by Windows-NT are best seen by examining the file system — but we must be using a partition bearing the Windows-NT File System (NTFS) and not simply a FAT-based (ala.Windows’98) partition.
Access controls in Windows-NT can be very particular – for example, considering a file on an NTFS volume, one can:

- let no one but the owner access it,
- let any single user access it,
- let various individual users access it,
- name an NT group and let any classified members access the file,
- name an NT group and let any group members access the file, while also rejecting access to individual members of that group,
- name various groups that can access it, or
- let anyone, possibly excluding certain individuals, ingress it. A variety of objects under Windows-NT can have also locks applied to them.

Under Windows-NT, these locks are called security descriptors. They may be set when an object is created, or they may be examined or set (with permission) once the object exists.

A security descriptor has four attributes:

- An owner identifier, indicating the current holder of the object (it can be given away),
- A primary classified identifier,
- A system access control list (SACL) containing auditing information, and
- A discretionary access control list (DACL), that determines which users can and cannot ingress the object.

Security descriptors may be ascribed to files (under NTFS), directories (under NTFS), registry keys, processes, threads, mutexes and semaphores (synchronization objects), events, named pipes, anonymous pipes, mailslots, console screen buffers, file mappings, network services, and private objects(!).
1.3.4.11 Access control lists (ACLs)

A substitute to directories is for objects to manage their own access. An access control list (ACL) contains a list of which subjects (and their processes) may access the object, and in which ways.

There is one ACL for each object. Each list is traditionally conserved, and access checked, by the kernel, although it could be practical for objects to maintain and constrain their own access (but remember that most objects are passive).

**ACLs**

- will typically list valid ingress modes by individual users. The owner of an object may or may not have permission to alter the ACL of the object itself.

- may list valid access modes by whole named groups of users,

- similarly reject access by users even if they are members of permitted groups,

- and allow access using wildcards naming users and groups. Wildcards usually appear, or are evaluated, last so that access may be first denied.

Searching an access control list is undertaken until a match of the requesting user and ingress mode is located. Default conditions (considered unwise) may be supported using “open” wildcards, or specific default ingress elements at the tail of each ACL.

In contrast to the use of directories, it is now more inconvenient for a subject to list all objects to which they have access, but simpler for an object to determine which subjects may access it.

1.3.4.12 Access Tokens and User Rights

When a user logs into a Windows-NT system they are given an access token. Each process control block entry (i.e. each process) contains its default access token. The access token recognises the user, the primary groups, such as Power User, Backup Operator, etc, and any custom groups, such as year2. The access token also contains the
user rights or privileges. These may be as reported on a per-user or per-group basis, and include the abilities to:

- Create Audit Logs,
- Modify the Token Itself,
- Perform Backups,
- Change a Process Priority,
- Shutdown the System,
- Change Quotas,
- Lock Physical Pages into Memory,
- Debug Processes,
- View Security Logs.
- Take Ownership of an Object, and
- View Security Logs.

1.3.4.13 Discretionary Access Control List

The Discretionary Access Control List (DACL) is the heart of Windows-NT security, determining who can and cannot ingress an object. It is a list (actually stored as an array) of access control entries (ACEs) each of which indicates abilities of users and groups on that object. For example, if the user charles-p may read a file, an access allowed ACE will permit this, while a separate access denied ACE may deny access to diana-l.

1.3.4.14 System Access Control List

The System Access Control List (SACL) also contains ACEs, but these ACEs determine who (which users and groups) will be audited.

An ACE in a SACL is called an audit access ACE. For example, an audit access ACE could indicate that every time charles-preads an object (such as a file object), that
information should be registered to a file. Similarly, each time diana-l attempts to read that object, that attempt will be registered

1.3.5 Scheduling and resource management

1.3.5.1 Factors to be considered

- Fairness- gives balanced and fair access to all jobs of the same class.
- Differential Responsiveness- discriminates between dissimilar classes of jobs.
- Efficiency- maximize throughput, minimize response time and accommodate as multiple users as possible.

The vital elements of the operating system involved in scheduling and resource management in a multiprogramming environment.

- I/O queues: processes waiting for I/O devices (a queue is related with each I/O device)
- Short-term queue: a list of processes whose implementation has been started. They reside in main memory. Round-robin technique: give each process some time in turn.
- Long-term queue: a list of fresh jobs waiting to be started.

1.3.5.2 Working

Suppose that a process is running. The following events fetch the control from the process to the OS.

- Service call from the process - the process explicitly implores some of the OS services, e.g. I/O operations.
- Interrupt from the process, e.g. exception handling.
- Timer interrupt - the time slice for the process is extinct.
- I/O interrupt - an I/O device reports its status to the OS.
In each case the equating handler is invoked, and then the short-tem scheduler picks up the next process to be run as shown in figure 1.6.

**Figure 1-6 Key Elements of an Operating System for multi-programming**

### 1.4 Characteristics of Modern Operating Systems

#### 1.4.1 Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which holds the addresses of all the service routines.

- Interrupt architecture must rescue the address of the interrupted instruction.

- Incoming interrupts are disabled while another interrupt is being processed to prevent a lost interrupt.

- A trap is a software-generated interrupt averted either by an error or a user request.

- An operating system is interrupt driven and Interrupt time line for a single process doing output shown in Figure 1-5.
Figure 1-7  Interrupt time line for a single process doing output.

1.4.2  Hardware Support

The operating system utilizes a table containing an entry for each input output device called device status table. Every table entry indicates the device’s type, address and state. If the device is busy with a request, the type of request and other variables stored in table entry for that device. Since it is feasible for other process to issue the request for the same device, the OS maintains a wait queue.

Figure 1-8 Device status table
1.4.3 Storage Hierarchy

Very fast storage is very costly. So the Operating System manages a ranking of storage devices in order to make the best use of resources. In fact, considerable effort goes into this support.

![Storage-device hierarchies](image)

**Figure 1-9 Storage-device hierarchies**

1.4.4 Caching

Caching is an important integrity of computer systems. Information is generally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system-the cache-on a interim basis. When we need a particular piece of
information, we first check whether it is in the cache. If it is, we use the information straightly from the cache; if it is not, we use the information from the main storage system, putting a copy in the cache under the assumption that we will need it again soon.

In addition, internal programmable registers, for example index registers, provide a high-speed cache for main memory. The programmer (or compiler) performs the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory. There are also caches that are executed totally in hardware. For example, most systems have an instruction cache to hold the next instructions expected to be executed. Without this cache, the CPU would have to wait various cycles while an instruction is fetched from main memory. For similar reasons, most systems have one or more high-speed data caches in the memory ranking.

We are not concerned with these hardware-only caches in this text, since they are exterior of the control of the operating system.

Because caches have restricted size, cache management is an important design problem. Careful selection of the cache size and of a replacement policy can result in 80 to 99 percent of all ingress being in the cache, greatly increasing performance.

Main memory can be prospected as a fast cache for secondary storage, since data in secondary storage must be copied into main memory for use, and data must be in main memory before being carried to secondary storage for safekeeping. The file-system data, which occupies permanently on secondary storage, may appear on several levels in the storage hierarchy. At the highest level, the operating system may prolong a cache of file-system data in main memory. Also, electronic RAM disks (also known as solid-state disks) may be used for high-speed storage that is obtained through the file-system interface.

The size of secondary storage is on magnetic disks. The magnetic-disk storage, in turn, is often backed up onto magnetic tapes or removable disks to defend against data loss in case of a hard-disk failure. Some systems automatically records old file data from secondary storage to tertiary storage, such as tape jukeboxes, to lower the storage cost. The movement of information between levels of a storage ranking may be either explicit
or implicit, depending on the hardware design and the controlling operating-system software. For example, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention. On the other hand, transfer of data from disk to memory is generally controlled by the operating system.

### 1.4.5 Hardware Protection

The aim is protecting the Operating System and others from malicious or ignorant users. Concurrent threads might obstruct with others. This heads to protection of resources by user/supervisor mode.

### 1.4.6 Dual-Mode Operation

To ensure actual operation, we must protect the operating system and all other programs and their data from any malfunctioning program. Protection is needed for any shared resource. The approach taken by many operating systems gives hardware support that allows us to differentiate among various modes of execution.

At the very least, we need two separate modes of operation: user mode and monitor mode (also called supervisor mode, system mode, or privileged mode). A bit, known as the mode bit, is added to the hardware of the computer to indicate the current mode: monitor (0) or user (1). With the mode bit, we are able to differentiate between a task that is executed on behalf of the operating system, and one that is executed on behalf of the user. As we shall see, this architectural enhancement is useful for many other facet of system operation. At system boot time the hardware starts in monitor mode. The operating system is then loaded and starts user processes in user mode. Whenever a trap or interrupt occurs, the hardware shifts from user mode to monitor mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gets control of the computer, it is in monitor mode. The system always shifts to user mode (by setting the mode bit to 1) before passing control to a user program. The dual mode of operation gives us with the means for protecting the operating system from errant users, and errant users from one another. We achieve this protection by designating some of the machine instructions that may cause harm as privileged instructions. The hardware allows privileged instructions to be released only in monitor mode. If an attempt is made to release a privileged instruction
in user mode, the hardware does not release the instruction, but rather treats the instruction as illegal and catches it to the operating system.

The concept of privileged instructions also gives us with the means for the user to interact with the operating system by questioning the system to perform some designated tasks that only the operating system should do. Each such request is invoked by the user releasing a privileged instruction. Such a request is known as a system call (also called a monitor call or an operating-system function call).

When a system call is released, it is treated by the hardware as software interrupt. Control proceeds through the interrupt vector to a service routine in the operating system, and the mode bit is set to monitor mode. The system-call service routine is a part of the operating system. The monitor inspects the interrupting instruction to determine what system call has occurred; a parameter shows what type of service the user program is requesting. Additional information needed for the request may be progressed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The monitor authenticates that the parameters are correct and legal, releases the request, and returns control to the instruction following the system call as shown in figure 1.10.
Figure 1-10 Use of a system call to perform I/O
1.4.7 I/O Protections

A user program may damage the normal operation of the system by issuing illegal I/O instructions, by accessing memory locations within the operating system itself, or by refusing to give up the CPU. We can use various mechanisms to ensure that such distortion cannot take place in the system.

To prevent users from performing illicit I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions straightly; they must do it through the operating system. For I/O protection to be complete, we must be sure that a user program can never gets control of the computer in monitor mode. If it could, I/O protection could be compromised.

Consider a computer executing in user mode. It will switch to monitor mode whenever an interrupt or trap arises, jumping to the address determined from the interrupt vector. If a user program, as part of its execution, stores a new address in the interrupt vector, this new address could overwrite the foregoing address with an address in the user program. Then, when a corresponding trap or interrupt happened, the hardware would switch to monitor mode, and would transfer control through the (modified) interrupt vector to the user program.

The user program could acquire control of the computer in monitor mode. In fact, user programs could get control of the computer in monitor mode in many other ways. In addition, new bugs are uncovered every day that can be exploited to bypass system protections. Thus, to do I/O, a user program executes a system call to request that the operating system execute I/O on its behalf. The operating system, executing in monitor mode, checks that the request is viable, and (if the request is valid) does the I/O requested. The operating system then returns to the user. I/O Define I/O instructions as confidential; they can be executed only in Supervisor mode.

Memory Protection: A user program can only ingress its own logical memory. For example, it can't modify supervisor code. To ensure accurate operation, we must protect the interrupt vector from modification by a user program. In addition, we must also defend the interrupt-service routines in the operating system from modification. Even if
the user did not get unauthorized control of the computer, modifying the interrupt service routines would probably distort the proper operation of the computer system and of its spooling and buffering.

We see then that we must provide memory safeguard at least for the interrupt vector and the interrupt-service routines of the operating system. In general, we want to protect the operating system from ingress by user programs, and, in addition, to protect user programs from one another. This protection must be given by the hardware.

This protection is proficient by the CPU hardware comparing every address generated in user mode with the registers. Any attempt by a program executing in user mode to access monitor memory or other users' memory results in a ploy to the monitor, which treats the attempt as a fatal error. This scheme prevents the user program from (accidentally or deliberately) altering the code or data structures of either the operating system or other users.

The base and limit registers can be loaded by only the operating system, which uses a special affluent instruction. Since privileged instructions can be executed in only monitor mode, and since only the operating system executes in monitor mode, only the operating system can charge the base and limit registers.

This scheme allows the monitor to change the value of the registers, but stops user programs from changing the registers' contents. The operating system, executing in monitor mode, is given unrestricted ingress to both monitor and users' memory. This contingency allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls.

1.4.8 CPU Protection

In addition to protecting I/O and memory, we must ensure that the operating system retain control. We must prevent a user program from getting pushed in an infinite loop or not calling system services, and never returning control to the operating system. To achieve this goal, we can use a timer. A timer can be set to interrupt the computer after a defined period. The period may be fixed (for instance, 1/60 second) or variable (for
instance, from 1 millisecond to 1 second). A variable timer is commonly implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt happens. For example, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

Before turn-off over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control moves automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Clearly, instructions that alter the operation of the timer are privileged. Thus, we can use the timer to avert a user program from running too long. A simple method is to initialize a counter with the amount of time that a program is allowed to run. A program with a 7-minute time limit, for instance, would have its counter initialized to 420. Every second, the timer interrupts and the counter is decremented by 1. As long as the counter is positive, command is returned to the user program. When the counter becomes negative, the operating system terminates the program for be more than the assigned time limit.

A more general use of a timer is to implement time sharing. In the most undemanding case, the timer could be set to interrupt every N millisecond, where N is the time slice that each user is allowed to execute before the next user gains control of the CPU. The operating system is invoked at the end of each time slice to perform various
housekeeping tasks, for example adding the value \( N \) to the record that specifies (for accounting purposes) the amount of time the user program has executed thus far. The operating system also saves registers, internal variables, and buffers, and changes several other variables to prepare for the next program to run. Following a context switch, the next program pursues with its execution from the point at which it left off (when its previous time slice ran out).

Another use of the timer is to calculate the current time. A timer interrupt signals the passage of some period, allowing the operating system to calculate the current time in reference to some initial time. If we have interrupts every 1 second, and we have had 1427 interrupts since we were told that it was 1:00 P.M., then we can calculate that the current time is 1:23:47 P.M. Some computers determine the current time in this manner, but the calculations must be done carefully for the time to be kept correctly, since the interrupt-processing time (and other times when interrupts are disabled) tends to cause the software clock to slow down. Most computers have a different hardware time-of-day clock that is independent of the operating system.

**1.4.9 CPU-I/O Burst Cycle**

A clock blocks programs from using all the CPU time. This clock causes an interrupt that causes the operating system to get control from a user program. For machines connected together, this protection must expand across: Shared resources, Multiprocessor Architectures, Clustered Systems and the practice of this is called “distributed operating systems”.

**1.5 Emergence of CPU**

**1.5.1 CPU Scheduling**

The goal of CPU schedulers is to provide a vision that each process or thread has its own dedicated CPU. The mechanisms needed to virtualize the CPU are fairly simple. Generating a policy to divide the physical CPU amongst competing processes and threads is a far more difficult problem. Understanding this problem in some feature is critical to appreciating the importance of increasing the transparency of CPU schedulers, whether through improved interfaces or empirical observations.
CPU scheduling is not programming; there is not an optimal solution. Rather CPU scheduling is about balancing objects and making difficult tradeoffs. Identifying the underlying objects of a CPU scheduler is key to appreciating its complex behavior. Appreciating the tradeoffs schedulers make to accomplish their objects is critical to understanding the evolution of commodity schedulers. A new scheduler either places an emphasis on different objects, or provides a simpler way to achieve the same objects as its predecessors. Understanding commodity schedulers is the first step to helping them. One must work hard to evade applying value judgments to scheduling policies. The only poor scheduling policy is one that trades something for nothing, one that de-emphasizes one object without an improvement in another. For example, it can be tempting to dismiss commodity scheduler’s out-of-hand as too complicate. Butler Lampson once advocated returning to a basic three-level, round-robin scheduler. Lampson’s scheduler would not be better than commodity schedulers; it would merely place a greater emphasis on simplicity.

System designers create CPU scheduling policies to match a particular environment. Each tradeoff in a scheduling policy returns assumptions about workloads and hardware configurations. Therefore, a CPU scheduler can only be assessed with respect to how well it works in a given environment. It is essential to note that general-purpose is an environment choice; it merely encapsulates all other choices.

We start this chapter by describing the hardware, operating systems, and applications that make up a classical distributed computing environment. We then categorize CPU schedulers, discussing how every category achieves their primary goal and at what cost. Next, we describe how multiprocessors systems introduce a new set of conflicting scheduling goals. We also present two categories of multiprocessor scheduler in this section. We finish with an overview of CPU scheduling policies in commodity operating systems, with an emphasis on Linux.

An operating system communicates between the user and the computer hardware. The aim of an operating system is to provide a platform in which a user can implement programs in well-located and efficient manner. Time Sharing systems are more composite they have developed from a single task to a multitasking environment in which
processes run in line manner. CPU scheduling is a mandatory operating system task; therefore its scheduling is central to operating system design. When there is number of process in the ready queue or job pool waiting for the CPU, the operating system must determine through the scheduler the order of execution. Allocating CPU to a process needs careful awareness and avoids process starvation for CPU. Scheduling decision try to minimize the following: turnaround time, response time and average waiting time for processes and the number of context switches.

Scheduling algorithms are the mechanism by which a resource is allocated to a process or task. CPU scheduling is the mechanism by which a resource is allocated to a process or task and accomplishes in different ways. For scheduling many scheduling algorithms are used like FCFS, SJF, RR, and Priority scheduling algorithm. The processes are scheduled according to the given burst time, occurrence time and priority. The implementation of processes used number of resources such as Memory, CPU etc.

A scheduling decision refers to the concept of selecting the next process for implementation. During each scheduling decision, a context switch occurs, meaning that the current process will stop its implementation and put back to the ready queue and another process will be dispatched. We define the scheduling raised cost when more context switches and all process are switching the finally CPU performance will be decreased.

Scheduling algorithms are widely used in transmissions networks and in operating systems to allocate resources to competing tasks. In operating systems, the scheduler must order the set of running processes for access to the CPU and other system resources. This research explores several well known CPU scheduling algorithms by means of analysis and compares their performance under different workloads.

The aim of process scheduling is to assign the processor or processors to the set of processes in a way that meets system and user objectives such as response time, throughput or processor efficiency. Various scheduling algorithms have been proposed; many are well understood. This research focuses to enable a quantitative differentiation of the performance of the Round Robin Algorithm under a range of workloads.
Schedulers can be either work conserving or non-work conserving. Work conserving schedulers always schedule a task if one is run-able, whereas non-work conserving schedulers may inactive the CPU even if there is a run-able task, typically to meet some performance constraint for the application, such as ensuring that it runs with a stringent periodicity. [1]

![Figure 1-12 A schematic of scheduling](image)

All requests waiting to be serviced are store in a list of pending requests. An arriving request is added to this list. When-ever scheduling is to be performed; the scheduler inspects the pending requests and selects one for servicing (this is shown by the dashed arrow in Figure 1-14). This request is handed over to the server. A request discards the server when it completes or when it is preempted by the scheduler, in which case it is put back into the list of pending requests. In either situation, scheduler presents scheduling to select the next request to be serviced. Thus, four events associated to scheduling are occurrence, scheduling, preemption and completion.

A request is the implementation of a job or a process. A User submits a request and waits for its completion. In an interactive environment, a user may interact with a process during its execution-the user makes a sub request to a process and the process responds by executing an action or by computing a result.

Table 1-3 summarizes scheduling-related concepts and idioms. Many time of a job or process is the total of CPU time and I/O time needs by it to complete its operation. It is an
inherent property of a job or process. The total time consumed by a job or a process in the OS may be longer than its service time because there might be times when it is neither executing on the CPU nor performing I/O. Consequently, its completion time depends on its occurrence time, service time, and the kind of service it is gives by the OS. Scheduling associated concepts can be grouped into user-centric concepts and system-centric concepts. Let we discuss these in the following:

**User-centric scheduling concepts**-Response time, deadline overrun, WM around time and weighted turn around are user-centric or application-centric perspective of a scheduler's performance. Response time is the time since submission of a sub-request to the time its processing is finished. It is an absolute measure of service.

<table>
<thead>
<tr>
<th>Term or concept</th>
<th>Definition or description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request related</td>
<td></td>
</tr>
<tr>
<td>Occurrence time</td>
<td>Time when a user submits a job/process</td>
</tr>
<tr>
<td>Admission time</td>
<td>Time when the system starts considering a job/process for scheduling</td>
</tr>
<tr>
<td>Completion time</td>
<td>Time when a job/process is completed</td>
</tr>
<tr>
<td>Deadline</td>
<td>Time by which a job/process must be completed to meet the response requirement of a real time application</td>
</tr>
<tr>
<td>Service time</td>
<td>The total of CPU time and I/O time required by a job/process or subsequent to complete its operation.</td>
</tr>
<tr>
<td>Preemption</td>
<td>Forced deallocation of CPU from a job or process</td>
</tr>
<tr>
<td>Priority</td>
<td>Priority is a tie-breaking rule used to select a request when many requests await service</td>
</tr>
<tr>
<td>User service</td>
<td></td>
</tr>
</tbody>
</table>
### related: individual request

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadline overrun</td>
<td>The amount of time by which the completion time of a job/process exceeds its deadline. Deadline overruns can be both positive or negative.</td>
</tr>
<tr>
<td>Fair share</td>
<td>A specified share of CPU time that should be devoted to execution of a process or a group of processes.</td>
</tr>
<tr>
<td>Response ratio</td>
<td>The ratio ( \frac{(\text{time since occurrence} + \text{service time of the process})}{\text{service time of the process}} )</td>
</tr>
<tr>
<td>Response time (rt)</td>
<td>Time between the submission of a subsequent for processing to the time its result becomes available. This concept is applicable to interactive processes.</td>
</tr>
<tr>
<td>Turnaround time</td>
<td>Time between the submission of a job/process and its completion by the system. This concept is meaningful for non-interactive jobs or processes only.</td>
</tr>
<tr>
<td>Weighted turnaround</td>
<td>Ratio of the turnaround time of a job/process to its own service time</td>
</tr>
<tr>
<td>User service related:</td>
<td></td>
</tr>
<tr>
<td>average service</td>
<td>Mean response time Average of the response times of all subrequests serviced by the system.</td>
</tr>
<tr>
<td>Mean turnaround time</td>
<td>Average of the turnarounds times of all subrequests serviced by the system.</td>
</tr>
</tbody>
</table>

### Scheduling related

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule length</td>
<td>The average number of jobs/processes or subrequests</td>
</tr>
</tbody>
</table>
1.5.2 Histogram of CPU-burst Times

The periods of CPU bursts have been extensively measured. Although they differ greatly by process and by computer, they tend to have a frequency curve similar to that shown in Figure 6.2. The curve is usually characterized as exponential or hyper exponential, with many short CPU bursts, and a few long CPU bursts. An I/O bound program would usually have many very short CPU bursts. A CPU-bound program might have some very long CPU bursts. This distribution can help us choose an appropriate CPU-scheduling algorithm.

![Histogram of CPU-burst times](image)

*Figure 1-13 Histogram of CPU-burst times*

1.5.3 Alternating Sequence of CPU and I/O Bursts

The success of CPU scheduling pivots on the following observed property of processes: Process execution subsists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution starts with a CPU burst. That is followed by...
an I/O burst, then further CPU burst, then another I/O burst, and so on. Eventually, the last CPU burst will end with a system request to terminate execution, rather than with further I/O burst.

![Diagram showing alternating sequence of CPU and I/O bursts.](image)

**Figure 1-14 Alternating sequence of CPU and I/O bursts.**

### 1.6 Motivation

The most important aspect of job scheduling is the capacity to multiprogramming. A individual user cannot, in general, keep either the CPU or the I/O devices busy at all times. Multiprogramming rises CPU utilization by organizing jobs so that the CPU always has one to execute. The idea is as follows: The operating system keeps various jobs in memory simultaneously. This set of jobs is a subset of the jobs remain in the job pool-since the number of jobs that can be kept simultaneously in memory is usually much
compact than the number of jobs that can be in the job pool. The operating system picks and starts to execute one of the jobs in the memory. Finally, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multi programmed system, the CPU would sit inactive. In a multiprogramming system, the operating system simply shifts to, and executes, another job. When that job needs to wait, the CPU is shifts to another job, and so on. Finally, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

In CPU scheduling, when number of processes have arrived in the ready queue for the execution the number of variable should be known. So, the processes are need the number of resources, their burst time, occurrence time, priority number to the associated process and the parameters needs for the execution of process. When first process is scheduled on the CPU then it should be automatically executed or timer lapse or it will preempt by the next upcoming process. Our goal of CPU scheduling is to achieve high degree of multiprogramming and decreases the response time, average waiting time turnaround time and number of context switches.

Different CPU scheduling algorithms are considered and verified. Like FCFS (first come first serve) schedules the number of processes observing to their given burst time. SJF (Shortest Job first) organizes the processes according to the burst time that means it executes the process which has small burst time. So it esteem shorter job first. The priority scheduling schedules the number of processes following to their given priorities. The priorities is allocated according to a integer number smaller the integer higher the priorities. So it essential to note that the average waiting time, turnaround time response time and number of context switches will not high.

The round robin scheduling is the one of scheduling algorithm which resolves the problem of SJF and Priority scheduling using time quantum. The FCFS, SJF and priority scheduling algorithms are not set into the time sharing systems. The RR scheduling algorithm is designed mostly for the time sharing system and it is starvation free. Its performance depends on the time quantum. If time quantum is very high then it is alike to FCFS. If time quantum is very low then it provides more context switches and lowers the
CPU efficiency. The weakness of RR scheduling algorithm is that it provides higher average waiting time, higher turnaround time and more context switches.

To enhance the average waiting time, turnaround time and number of context switches we introduce our proposed algorithm which is based on execution time i.e. on SJF and the problem of Time Quantum is also solved.

1.7 Problem Definition and Proposed Solution

The main goal of this thesis is to gives the feasible or optimal solution for the disadvantages of Round Robin that means minimize the average waiting time, turnaround time and number of context switches for the maximum CPU utilization.

Proposed Solution:

A new efficient algorithm is proposed in this thesis to reduce the average waiting time, average turnaround time and context switches using adaptive method based on smart time slice. The approach is similar to the RR but some changes arises here for improvements.

1.8 Preliminaries

I. Program

Program is assigns to the set of instructions that are executed in pipeline fashion.

II. Process

Sequential execution of a program is known as process. A process contain a text section in which program code is written, it also comprises current activity as represented by the program counter, it also include stack that clasp temporary data, it also comprise data section which contain global variables. A process may include a heap; which is memory that is dynamically granted during process at run time.

In computing, a process is an example of a computer program that is being executed. It contains the program code and its present activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions synchronously.
A computer program is a passive group of instructions; a process is the actual execution of those instructions. Several processes may be associated with the same program; for instance, opening up several instances of the same program often means more than one process is being executed.

![Diagram of Process State](image)

**Figure 1-15 Diagram of Process State**

The several process states, displayed in a state diagram above, with arrows indicates possible transitions between states.

An operating system kernel that permits multi-tasking needs processes to have certain states. Names for these states are not standardized, but they have same functionality.

- Firstly, the process is "created" - it is loaded from a secondary storage device (hard disk or CD-ROM...) into main memory. After that the process scheduler allots it the state "waiting".

- While the process is "waiting" it stays for the scheduler to do a so-called context switch and load the process into the processor. The process state then becomes "running", and the processor performs the process instructions.
- If a process needs to wait for a resource (wait for user input or file to open ...), it is assigned the "blocked" state. The process state is altered back to "waiting" when the process no longer needs to wait.

- Once the process finishes execution, or is terminated by the operating system, it is no longer needed. The process is removed immediately or is moved to the "terminated" state. When removed, it just stays to be removed from main memory.

III. Process Control Block

In operating system each process is illustrated by a process control block (PCB). The PCB contain many information about the process like process state, process number, program counter, list of open files, registers and CPU scheduling information.

Process Control Block (PCB, also called Task Controlling Block, Task Struct, or Switch frame) is a data structure in the operating system kernel containing the information needed to manage a particular process. The PCB is "the presentation of a process in an operating system".

If the mission of the operating system is to control computing resources on behalf of processes, then it must be continuously informed about the status of each process and resource. The approach generally followed to represent this information is to create and update status tables for each relevant entity, such as memory, I/O devices, files and processes. Memory tables, for example, may contain information about the allocation of main and secondary (virtual) memory for every process, authorization attributes for accessing memory areas shared among dissimilar processes, etc. I/O tables may have entries stating the availability of a device or its assignment to a process, the status of I/O operations being achieved, the location of memory buffers used for them, etc. File tables give info about location and status of files (of course, what else? more on this later). Finally, process tables keep the data the OS needs to manage processes. At least part of the process control data structure is always preserved in main memory, though its exact location and configuration varies with the OS and the memory management technique it uses. In the following we'll refer by process image to the total physical manifestation of a
process, which comprises instructions, program data areas (both static and dynamic - e.g. at least a stack for procedure calls and parameter passing) and the process management information. We'll call this last set the process control block (PCB).

The role of the PCBs is central in process management: they are accessed and/or altered by most OS utilities, including those involved with scheduling, memory and I/O resource access and performance monitoring. It can be said that the set of PCBs defines the current state of the operating system. Data structuring for processes is usually done in terms of PCBs. For example, pointers to other PCBs inside a PCB allow the creation of those queues of processes in several scheduling states ("ready", "blocked", etc.) that we previously mentioned.

In modern sophisticated multitasking systems the PCB stores many distinct items of data, all needed for correct and efficient process management. Though the descriptions of these structures are obviously system-dependent, we can identify some very common parts, and classify them in three main categories:

Process identification data; Processor state data; Process control data; Process identification data always comprise a unique identifier for the process (almost invariably an integer number) and, in a multiuser-multitasking system, data like the identifier of the parent process, user identifier, user group identifier, etc. The process id is especially relevant, since it's often used to cross-reference the OS tables defined above, e.g. to permit identifying which process is using which I/O devices, or memory areas.

Processor state data are those pieces of information that specify the status of a process when it's suspended, allowing the OS to restart it later and still enforce correctly. This always comprises the content of the CPU general-purpose registers, the CPU process status word, stack and frame pointers etc.

Process control information is used by the OS to maintain the process itself. This comprises:

The process scheduling state (different from the task state above discussed), example in terms of "ready", "suspended", etc., and different scheduling information as well, like a priority value, the number of time elapsed since the process derived control of the CPU or
since it was suspended. Also, in case of a suspended process, event identification data must be recorded for the event the process is waiting for. Process structuring information: process’s children id's, or the id's of other processes cognated to the current one in some functional way, which may be constituted as a queue, a ring or other data structures. Inter process communication information: various flags, signals and messages associated with the communication among independent processes may be stored in the PCB.

I. Accounting information

![Process Control Block (PCB)](image)

**Process state**: The state may be new, ready, running, waiting, halted, and so on.

**Program counter**: The counter shows the address of the next instruction to be executed for this process.

**CPU registers**: The registers vary in number and type, relying on the computer architecture. They comprise accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to permit the process to be continued correctly afterward.

**CPU-scheduling information**: This information comprises a process priority, pointers to scheduling queues, and any other scheduling parameters.
**Memory-management information:** This information may comprise such information as the value of the base and limit registers, the page tables, or the segment tables, relying on the memory system used by the operating system.

**Accounting information:** This information comprises the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

**Status information:** The information comprises the list of I/O devices allocated to this process, a list of open files, and so on.

The PCB simply works as the repository for any information that may vary from process to process.

**Included information**

Implementations differ, but in general a PCB will comprise, directly or indirectly:

- The identifier of the process (a process identifier, or PID)
- Register values for the process considering, notably, the program counter and stack pointer values for the process.
- The address space for the process
- Priority (in which maximum priority process gets first preference. e.g., nice value on Unix operating systems)
- Process accounting information, such as when the process was last run, how much CPU time it has collected, etc.
- Pointer to the following PCB i.e. pointer to the PCB of the next process to run
- I/O Information (i.e. I/O devices allotted to this process, list of opened files, etc.)
- During context switch, the running process is stopped and another process is given a possibility to run. The kernel must stop the execution of the running process, copy out the values in hardware registers to its PCB, and modify the hardware registers with the values from the PCB of the new process.
A process in an operating system is presented by a data structure known as a process control block (PCB) or process descriptor. The PCB contains major information about the specific process including:

- The present state of the process i.e., whether it is ready, running, waiting etc.
- Unique identification of the process in order to route "which is which" information.
- A pointer to parent process.
- Similarly, a pointer to child process (if it exists).
- The priority of process (a part of CPU scheduling information).
- Pointers to locate memory of processes.
- A register save area.
- The processor it is running on.

The PCB is a definite store that allows the operating systems to locate key information about a process. Thus, the PCB is the data structure that explains a process to the operating systems.
II. **Scheduling Queues**

- **Job Queue**: When processes join the system they are put into a job pool. This job pool refers to the job queue or place of all processes in the system. In system software, a job queue (sometimes batch queue), is a data structure managed by job scheduler software containing jobs to run. Users submit their programs that they want implemented, "jobs", to the queue for batch processing. The scheduler software maintains the queue as the pool of jobs accessible for it to run. Multiple batch queues might be used by the scheduler to differentiate types of jobs depending on parameters like:
  
  - job priority
  - estimated execution time
  - resource requirements
  - The use of a batch queue provides these benefits:
    - sharing of computer resources among many users
    - time-shifts job processing to when the computer is less busy

![Figure 1-17 CPU Switch From Process to Process](image)
- Avoids idle the compute resources without minute-by-minute human supervision allows around-the-clock high utilization of costly computing resources.

- Ready queue: The processes that are residing in main memory and are ready and waiting to execute are kept on a list is known as ready queue. Ready queue is set of all processes occupy in main memory, ready and waiting to execute. This queue is usually stored as a associated list. A ready-queue header holds pointers to the first and final PCBs in the list. Each PCB comprises a pointer field that points to the next PCB in the ready queue. Suppose the process makes an I/O request to a shared device, like a disk. Since there are many processes in the system, the disk may be engaged with the I/O request of some other process. The process therefore may have to hold back for the disk. The list of processes waiting for a particular I/O device is known as a device queue. Every device has its own device queue.

- Device Queue: Device queue refers to the when a process allots to the CPU, it executes or while waiting for some I/O devices for completing particular I/O event. So, the list of processes waiting in queue for specific I/O devices is called a device queue. Device queue is set of processes waiting for an I/O device. Figure 1.18 shows the Process migration between the different queues.
Figure 1-18 Ready queue and various I/O device queues.

Figure 1-19 Queueing-diagram representation of process scheduling.
III. Process Schedulers

Long-term scheduler (or job scheduler) – selects which processes should be showed into the ready queue.

Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allotted to CPU.

Processes can be described as either:

- I/O-bound process – spends more time doing I/O than computations, many small CPU bursts.
- CPU-bound process – expend more time doing computations; few very long CPU bursts.

An operating system has to give a suitable combination of user-centric and system-centric features. It also has to alter to the nature and number of user requests that are expected to arise in its environment, and to the availability of resources. A single scheduler and a single scheduling policy cannot label all its concerns. Therefore, an OS uses an arrangement consisting of three schedulers called the long-term scheduler, the medium-term scheduler and the short-term scheduler to address dissimilar user-centric and system-centric issues. Below table summarizes characteristic of these schedulers.

<table>
<thead>
<tr>
<th><strong>Long-term scheduling</strong></th>
<th>Decides when to disclose an arrived process for scheduling based on its nature, whether CPU-bound or I/O-bound, and accessibility of resources like kernel data structures, user terminals and disk space for swapping.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Medium-term scheduling</strong></td>
<td>Moves processes between the memory and the disk to optimize use of the memory. Manages a sufficient number of ready processes in the memory.</td>
</tr>
<tr>
<td><strong>Short-term</strong></td>
<td>Decides which ready process to accomplish next and for how</td>
</tr>
</tbody>
</table>
Figure 1-20 Event handling and scheduling

Figure 1-20 shows an overview of scheduling and related actions. As considered, the kernel operates in an interrupt-driven manner. Every event that needs the kernel's attention causes an interrupt. The interrupt handler executes a context Save function and Impresses an event handler. The event handler processes the event and changes the state of
the process pretentious by the event. It then intances the long-term, medium-term or short-term scheduler as appropriate. For example the event handler that creates a new process invokes the long-term scheduler, and the medium-term scheduler is invoked when a process is to be rejected or resumed. Eventually the short-term scheduler gains control and a process for execution. The process should be executed and arranges to produce a timer interrupt when the time elapses. It now hands over the chosen process to the dispatching mechanism.

In summary, the long-term scheduler chooses processes that should be considered for scheduling by the medium-term scheduler. The medium-term scheduler chooses processes that should be considered for scheduling by the short-term scheduler, and the short-term scheduler selects processes that should be executed on the CPU. Each scheduler applies its own criteria concerning use of resources and quality of service to choose processes for the next stage of scheduling. We consider the nature and relevance of long, medium and short-term scheduling activities in dissimilar OSs in the following Sections.

a. Long Term Scheduler: The long term scheduler is also known as job scheduler that means it chooses the process from the job pool and loads into the memory for execution.

The long-term, or admission scheduler, determine which jobs or processes are to be admitted to the ready queue (in the Main Memory); that is, when an attempt is made to implement a program, its admission to the set of currently executing processes is either authorized or delayed by the long-term scheduler. Thus, this scheduler control what processes are to run on a system, and the degree of concurrency to be beard at any one time - i.e.: whether a high or low amount of processes are to be accomplished concurrently, and how the split between input output intensive and CPU intensive processes is to be handled. In modern operating systems, this is worn to make sure that real time processes get enough CPU time to finish their tasks. Without actual real time scheduling, modern GUIs would seem sluggish. The long term queue exists in the Hard Disk or the "Virtual Memory".
Long-term scheduling is also significant in large-scale systems such as batch processing systems, computer clusters, supercomputers and render farms. In these cases, special purpose job scheduler software is typically used to help these functions, in addition to any underlying admission scheduling support in the operating system.

b. Short Term Scheduler: The short term scheduler is known as CPU scheduler that means it chooses the processes that are ready to execute and allocates the CPU to one of them.

The short-term scheduler (also known as the CPU scheduler) decides which of the ready, in-memory processes are to be accomplished (allocated a CPU) after a clock interrupt, an I/O interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions much more oftenly than the long-term or mid-term schedulers - a scheduling decision will at a minimum have to be made after every time slice, and these are very short. This scheduler can be preemptive, implying that it is effective of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or non-preemptive (also known as "voluntary" or "co-operative"), in which case the scheduler is ineffectual to "force" processes off the CPU.

A preemptive scheduler depends upon a programmable interval timer which invokes an interrupt handler that runs in kernel Mode and executes the scheduling function.

c. Medium Term Scheduler: The medium term scheduler is used in time sharing system. The basic concept of medium term scheduler is that sometimes it can be advantageous to remove processes from memory and thus reduce degree of multi-programming. Later, the processes can be reintroduced into memory, and its execution can be continued where it suspended. This scheme is called as swapping. So, the process is swapped out, and is later swapped in, by the medium term scheduler.
Scheduler temporarily removes processes from main memory and places them on secondary memory (such as a disk drive) or vice versa. This is commonly referred to as "swapping out" or "swapping in" (also incorrectly as "paging out" or "paging in"). The medium-term scheduler may decide to swap out a process which has not been active for some time, or a process which has a low priority, or a process which is page faulting frequently, or a process which is taking up a large amount of memory in order to free up main memory for other processes, swapping the process back in later when more memory is available, or when the process has been unblocked and is no longer waiting for a resource.

In many systems today (those that support mapping virtual address space to secondary storage other than the swap file), the medium-term scheduler may actually perform the role of the long-term scheduler, by treating binaries as "swapped out processes" upon their execution. In this way, when a segment of the binary is required it can be swapped in on demand, or "lazy loaded".

d. Dispatcher: Another component involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the
process selected by the short-term scheduler. This function involves the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program.

Dispatcher analyses the values from Program counter and fetches instructions, data into registers.

The dispatcher should be as fast as possible, since it is invoked during every process switch. During the context switches, the processor is idle for a fraction of time. Hence, unnecessary context switches should be avoided. The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

Burst Time: The time for which a process holds the CPU is known as burst time. Burst time is an assumption of how long a process requires the CPU between I/O waits. It cannot be predicted exactly, before a process starts. It means the amount of time a process uses the CPU for a single time. (A process can use the CPU several times before complete the job)

Occurrence Time: The time at which a process arrives is its occurrence time. The instant in time when a job becomes available

- For execution
- May not be exact: Release time jitter so ri is in the interval \([ri-, ri+]\)
- A job can be scheduled and executed at any time at, or after, its release time, provided its resource dependency conditions are met.

Turnaround Time: Turnaround time is the amount of time to execute a particular process. In Computing, turnaround time is the total time taken between the submission of a program/process/thread/task (Linux) for execution and the return of the complete output
to the customer/user. It may vary for various programming languages depending on the developer of the software or the program. Turnaround time may simply deal with the total time it takes for a program to provide the required output to the user after the program is started.

Turnaround time is one of the metrics used to evaluate operating system scheduling algorithms. In case of batch systems, turnaround time will include time taken in forming batches, batch execution and printing results.

- Waiting Time: Waiting time is the amount of time a process has been waiting in the ready queue.
- Response Time: Time from the submission of a request by the process till its first response is the response time.

IV. Context Switch

When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process. Context-switch time is overhead; the system does no useful work while switching and is dependent on hardware support.

1.9 Scope of the study

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst distribution

1.10 Objectives of CPU Scheduling

1. To introduce CPU scheduling, which is the basis for multi programmed operating systems

2. To describe various CPU-scheduling algorithms
3. To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

1.10.1 Performance Criteria

1. CPU Utilization: We want to keep the CPU as busy as possible that means CPU is not free during the execution of processes. Conceptually the CPU utilization can range from 0 to 100 percent.

2. Throughput: If the CPU is executing processes, then work is being completed. One measure work is the number of processes that are completed per time unit that means the number of tasks per second which the scheduler manages to complete the tasks.

3. Response Time: In Time Sharing System, turnaround time is not the measurement for the performance. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, response time is the time from the submission of a request until the first response is produced that means when the task is submitted until the first response is received. So the response time should be low for best scheduling.

The length of time from the release time of the job to the time instant when it completes - Not the same as execution time, since may not execute continually

![Release and Response Time](image)

**Figure 1-22 Release and Response Time**

4. Turnaround Time: Turnaround time introduces to the total time which is spend to complete the process and is how long it takes the time to implement that process. The time interval from the time of submission of a process to the time of accomplishment is the turnaround time. Total turnaround time is calculation is the
sum total of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

5. Waiting Time: The waiting time is not the measurement of time when a process implements or does I/O completion; it affects only the amount of time of submission of a process expends waiting in the ready queue. So the Waiting time is the period of spent waiting in the ready queue to agree the new arriving process for the CPU.

6. Correctness: For algorithms which allow the user to attach deadlines to a task, this measures the proportion of time or proportion of total scheduling time unit when a task fails to gather its deadline that means all the task should be encounter at the given deadline. Correctness of an algorithm is asserted when it is said that the algorithm is proper with respect to a specification. Functional correctness mentions to the input-output behavior of the algorithm (i.e., for each input it produces the correct output). A distinction is made between total correctness, which additionally needs that the algorithm terminates, and partial correctness, which simply needs that if an answer is returned it will be correct. Since there is no general solution to the halting problem, a total correctness declaration may lie much deeper. A abortion proof is a type of mathematical proof that plays a critical role in formal confirmation because total correctness of an algorithm depends on termination. For example, successively searching through integers 1, 2, 3, … to see if we can find an exemplar of some phenomenon — say an odd perfect number — it is quite simple to write a partially correct program (using long division by two to check $n$ as perfect or not). But to say this program is totally proper would be to assert something currently not known in number theory. A proof would have to be a mathematical proof, accepting both the algorithm and specification are given formally. In particular it is not expected to be a correctness declaration for a given program implementing the algorithm on a given machine. That would involve such considerations as restrictions on memory. A deep result in proof theory, the Curry-Howard correspondence, states that a proof of functional correctness in constructive logic communicates to a certain program in
the lambda calculus. Converting a proof in this way is known as program extraction. Hoare is a specific formal system for reasoning rigorously about the accurateness of computer programs. In computer science, real-time computing (RTC), or reactive computing, is the learning of hardware and software systems that are subject to a “real-time constraint”— e.g. operational deadlines from event to system response. Real-time programs must warranty response within strict time constraints. Often real-time response times are understood to be in the order of milliseconds and occasionally microseconds. In contrast, a non-real-time system is one that cannot warranty a response time in any situation, even if a fast response is the usual result. The use of this word should not be demented with the two other legitimate uses of ‘real-time’. In the realm of simulations, the term means that the simulation’s clock runs as speedy as a real clock would; and in the domain of data convey, media processing and enterprise systems, the expression is used to mean ‘without perceivable delay’. Real-time software may use one or more of the below: synchronous programming languages, real-time operating systems, and real-time networks, each of which give essential frameworks on which to build a real-time software application. A real-time system may be one where its application can be contemplate (within context) to be mission critical. The anti-lock brakes on a car are a easy example of a real-time computing system — the real-time constraint in this system is the time in which the brakes must be released to stop the wheel from locking. Real-time computations can be said to have failed if they are not finished before their deadline, where their deadline is comparative to an event. A real-time deadline must be met, regardless of system load.

7. Overhead: Overhead refers to the portion of time wasted due to computation of the schedule, and the system overhead due to surrounding switching the tasks when new task is arriving.

8. Efficiency: Efficiency mention to the respective of system when CPU is busy for scheduling of new arriving tasks. OS makes two related kinds of resolution about resources i.e. Allocation and Scheduling. Allocation means who gets what. Given
a set of requests for resources, which processes should be given which resources in order to make most important use of the resources. Implication is that resources are not easily preemptible. Scheduling means how long can they keep it? When more resources are requested than can be allowed immediately, in which order they should be serviced. Specimen are processor scheduling (one processor, many processes), memory scheduling in virtual memory systems. Implication is that resource is preemptible.

9. Fairness: In the absence of user or system supplied criteria for choice, the scheduler should allot a fair amount of the resource to each task. Fairness or waiting time is equal CPU time to each process (or more generally suitable times according to each process’ priority). It is the time for which the process remains in the ready queue. In practice, these goals often dispute (e.g. throughput versus latency), thus a scheduler will implement a suitable compromise. Preference is given to any one of the above stated concerns depending upon the user’s needs and objectives. [1]

1.10.2 Performance metrics

Metrics that guide scheduling decision relys on the application area. Classical scheduling theory typically uses metrics such as:

- minimizing the sum of collection time
- minimizing the weighted sum of completion time
- minimizing scheduling length
- minimizing number of processors needed
- minimizing the maximum lateness
- minimizing response time and increasing throughput
1.10.3 Problem with classical theory performance metric

- In most cases, deadlines are not even rated. In contrast, there are several key constraints in real-time systems that make the problem substantially hard.

- The main aim of real-time systems is to meet the timing-constraints of the individual task. This introduces require of quite different metrics for real-time systems

- The variation of metrics has been suggested depending upon the type of real-time systems that exist in the real world

- There are different demands for different scheduling algorithms. This makes it hard to compare them.

Suitability of performance metrics

- The sum of accomplishment times is generally not of interest because there is no direct assessment of timing properties.

- The weighted sum is very influential when tasks have different values that they impart to the systems upon completion

- Value is generally overlooked in the real-time systems, many of which simply concentrate on deadline rather than on a combination of value and deadlines.

- Minimizing scheduling length has secondary significance in minimizing required system resources, but it does not straightly address the fact that individual task has deadline

- The same is true for minimizing the number of processors needed. Minimizing the maximum lateness metrics can be functional at design time when resources can be added until the maximum lateness is less than or equal to zero. In that case, no task misses its deadline.
• In dynamic real–time systems it cannot be apriori warranted that all deadlines will be met, maximizing the number of tasks that meet their deadlines is generally used as metrics.

• The level of predictability provided by a particular scheduling scheme may be another metrics in real-time systems.

• An optimal scheduling algorithm is one that may vanish to meet a deadline only if no other scheduling algorithm can meet it.

• Similarly, some algorithms are planned to deal with only pre-emptible tasks while others can handle only non-preemptive tasks.

• Task independence, resource and placement constraints, criticalness and strictness of deadline (soft, firm or hard) are examples of some other features of real-time tasks, which affect the nature of the scheduling algorithms.

• Scheduling algorithms also rely on the type of computer system they are intended for. Few algorithms are for uni-processor systems while other is for multiprocessor systems.

• Finally, there can be other parameters and aim [imprecise] of scheduling algorithms along which they can be differentiated.

1.10.4 Performance measures

Per process:

• Waiting time 20 seconds

• Turnaround time 40 seconds

• Required time 20 seconds

• Penalty ratio (1/Response ratio) \(\frac{40}{20} = 2\)

System measures:

• Average waiting time
• Throughput k processes per min.
• Average penalty ratio (Response ratio)
• Average Turnaround time

Requirements of CPU Scheduling:
• CPU and IO cycles
• Real Time vs. non-real time tasks
• Preemption vs. no preemption
• Short vs. long tasks
• Utilization of idle cycles
• Priorities of tasks

The many algorithms can be categorized into set of paradigms depending on conditions:
• Whether a system executes schedulability analysis
• If it does, Whether it is done statically or dynamically
• Whether the result of the analysis itself makes a schedule of plan according to which tasks are displaced at run time

Scheduling Policies:
• Non-preemptive policies
• Scheduled till completion
• Once a process is scheduled, it remains
• A scheduled process may be preempted and another may be scheduled
• Preemptive policies

When is a scheduler invoked:
• Creation

• Voluntary withdrawal

• Wait for a slower device

• Completion

• Policy dependent events

• Device Ready

**Scheduling**

Scheduling is a Process of the allotment of resources and time to tasks in such a way that certain performance needs are met. The most broadly researched topic within real-time systems and in the past few years’ extensive work has been done in this area. Comprehensive and united scheduling approaches are required and should be able to handle the following matters: Preemptive and non preemptive tasks; Periodic and non periodic tasks; Tasks with various level of importance or (value function); No of tasks with a simple deadline; End to End timing constraints, Precedence limitations; Communication limitations /requirements, Resource needs, Placement constraints, Fault tolerance needs, Tight and loose deadlines, Normal and overload conditions.

The solution must be integrated enough to handle the interface between

• CPU Scheduling and resource allotment

• I/O Scheduling and CPU Scheduling

• Local and Distributed scheduling and

• Static scheduling and dynamic scheduling

• CPU Scheduling and real time communication scheduling
Significance of CPU scheduling

For the last 15 years, advances in CPU scheduling algorithms have been made largely unrelated by the rapidly increasing speed of processors (some exceptions stand out. Various interesting works in the area of real-time scheduling for multimedia applications were simply outpaced by the speed of new processors. Each modern desktop computer can easily play streaming media, and all without the help of research techniques designed solely for that purpose. In fact, several of these desktops play real-time media using timesharing scheduling technology developed in 80’s and early 90’s.

Why, then, should the systems research company invest time and money in scheduling research?

The concept is the free hardware ride is over; processor speeds peaked some time in 2003. The present trend in hardware is more processors instead of faster processors, and this means that CPU scheduling is relevant again. Improved CPU scheduling is needed to generate application performance improvements from multicore processors. Multicore processors do not automatically give performance improvements to applications the way faster processors did. Instead applications must be redesigned to grow their parallelism. Similarly, CPU schedulers must be redesigned to maximize the performance of this fresh application parallelism. CPU scheduling policy (and in a large part mechanism) is insignificant to a serial application running on its own machine. Now, however, an application may be competing/cooperating with various concurrent instances of itself on a single machine. In this scenario, CPU scheduling is vitally important. An application may act unresponsive, flaky, or even schizophrenic if some portions of the application starve while others thrive. CPU scheduling is mostly unrelated if the CPU is underutilized; on an underutilized CPU a scheduler can only truly affect scheduling latency. One initially expects that the increased number of processors per machine would minimize the potential for CPU contention, and thereby the need for good CPU scheduling. However, the increase in redundant hardware has happened concurrently with an increase in server consolidation, which reintroduces the potential for CPU contention.
To rise profits and improve output (of the cluster), server consolidation may reduce an application’s hardware allocation until the application is running at near its allotted hardware capacity. This close tailoring of resource allotted shifts part of the scheduling problem to the application’s operating system. Running at near capacity means that even less increases in load may place the system in an overload state, and CPU scheduling becomes critical when systems are overloaded. Under overload, the CPU scheduler must make cautious decisions about how to divide its limited resources.

1.11 Organization of Thesis

A concise summary of the rest of the thesis work is explained below:

- In the chapter 1, the background of operating system and types of operating system was examined.

- In the chapter 2, the background of scheduling and CPU scheduling is examined. Definitions of few terms which are involved CPU scheduling are reviewed. The different type of CPU scheduling was also examined.

- Chapter 3 mainly emphasis on the detailed study of literature survey related to RR CPU scheduling policies used in operating system. It leads with an introduction to the RR.

- Chapter 4 covers the new proposed CPU scheduling algorithm and real example of it.

- Chapter 5 reviews the waiting rime and context switches i.e. performance aspects of RR and ERR. It comprises experimental results and comparisons with RR.

- Chapter 6 comprises simulation results and performance of the algorithm.

- Chapter 7 comprises conclusion of the whole thesis.

- Chapter 8 comprises the future scope of the research so that further research work can be carried out on this topic.
• Chapter 9 comprises all references considering research paper, books and web links for the research work.

• Chapter 10 comprises paper published on thesis work.