OPERATING SYSTEM

LECTURE NOTES (Semester – IV)

for

Bachelor of Science in ***Computer Science***

Department of Computer Science and Applications

School of Arts & Science

Vinayaka Mission's Research Foundation

A V Campus, Chennai – 603104.

Lecture Note Prepared by:

K. PUSHPAVATHI, Associate Professor

SYLLABUS

OPERATING SYSTEM

L T P C

4 0 0 4

OBJECTIVE:

To provide an introduction to the internal operation of modern operating systems.

UNIT-I (12)

Introduction: System Software, Resource Abstraction, OS strategies. **Types of operating systems** - Multiprogramming, Batch, Time Sharing, Single user and Multiuser, Process Control & Real Time Systems.

UNIT-II (12)

Operating System Organization: Factors in operating system design, basic OS functions, implementation consideration; process modes, methods of requesting system services – system calls and system programs.

UNIT-III (12)

Process Management : System view of the process and resources, initiating the OS, process address space, process abstraction, resource abstraction, process hierarchy, Thread model **Scheduling:** Scheduling Mechanisms, Strategy selection, non-pre-emptive and pre-emptive strategies.

UNIT-IV (12)

Memory Management: Mapping address space to memory space, memory allocation strategies, fixed partition, variable partition, paging, virtual memory

UNIT-V (12)

Shell introduction and Shell Scripting :
Shell and various type of shell, Various editors present in linux, Different modes of operation in vi editor, shell script, Writing and executing the shell script, Shell variable (user defined and system variables), System calls, Using system call , Pipes and Filters, Decision making in Shell Scripts (If else, switch), Loops in shell Functions Utility programs (cut, paste, join, tr , uniq utilities), Pattern matching utility (grep)

Total Hours: 60

TEXT BOOK

1. A Silberschatz, P.B. Galvin, G. Gagne, Operating Systems Concepts, 8th Edition, John Wiley Publications 2008.

REFERRENCES

1. A.S. Tanenbaum, Modern Operating Systems, 3rd Edition, Pearson Education 2007.
2. G. Nutt, Operating Systems: A Modern Perspective, 2nd Edition Pearson Education 1997. th
3. W. Stallings, Operating Systems, Internals & Design Principles, 5 Edition, Prentice Hall of India. 2008.
4. M. Milenkovic, Operating Systems- Concepts and design, Tata McGraw Hill 1992.

UNIT I

System software

System software is a type of computer program that is designed to run a computer's hardware and application programs. If we think of the computer system as a layered model, the system software is the interface between the hardware and user applications.

System software is a software that provides platform to other softwares. Some examples can be operating systems, antivirus softwares, disk formating softwares, Computer language translators etc. These are commonly prepared by the computer manufacturers. These softwares consists of programs written in low-level languages, used to interact with the hardware at a very basic level. System software serves as the **interface between the hardware and the end users.**

The most important features of system software include :

1. Closeness to the system
2. Fast speed
3. Difficult to manipulate
4. Written in low level language
5. Difficult to design

An operating system (OS) is a type of system software that manages computer's hardware and software resources. It provides common services for computer programs. An OS acts a link between the software and the hardware. It controls and keeps a record of the execution of all other programs that are present in the computer, including application programs and other system software.

The most important tasks performed by the operating system are

1. **Memory Management**: The OS keeps track of the primary memory and allocates the memory when a process requests it.
2. **Processor Management**: Allocates the main memory (RAM) to a process and de-allocates it when it is no longer required.
3. **File Management:** Allocates and de-allocates the resources and decides who gets the resources.
4. **Security:** Prevents unauthorized access to programs and data by means of passwords.
5. **Error-detecting Aids:** Production of dumps, traces, error messages, and other debugging and error-detecting methods.
6. **Scheduling:** The OS schedules process through its scheduling algorithms.

Compiler : A compiler is a software that translates the code written in one language to some other language without changing the meaning of the program. The compiler is also said to make the target code efficient and optimized in terms of time and space.
A compiler performs almost all of the following operations during compilation: preprocessing, lexical analysis, parsing, semantic analysis (syntax-directed translation), conversion of input

programs to an intermediate representation, code optimization and code generation. Examples of compiler may include gcc(C compiler), g++ (C++ Compiler), javac (Java Compiler) etc.

Interpreter : An interpreter is a computer program that directly executes, i.e. it performs instructions written in a programming or scripting language. Interpreter do not require the program to be previously compiled into a machine language program. An interpreter translates high-level instructions into an intermediate form, which is then executes.
Interpreters are fast as it does not need to go through the compilation stage during which machine instructions are generated. Interpretter continuously translates the program until the first error is met. If an error comes it stops executing. Hence debugging is easy. Examples may include Ruby, Python, PHP etc.

Assembler : An assembler is a program that converts assembly language into machine code. It takes the basic commands and operations and converts them into binary code specific to a type of processor.
Assemblers produce executable code that similar to compilers. However, assemblers are more simplistic since they only convert low-level code (assembly language) to machine code. Since each assembly language is designed for a specific processor, assembling a program is performed using a simple one-to-one mapping from assembly code to machine code. On the other hand, compilers must convert generic high-level source code into machine code for a specific processor.

Other examples of system software include:

- The BIOS (basic input/output system) gets the computer system started after you turn it on and manages the data flow between the operating system and attached devices such as the hard disk, video adapter, keyboard, mouse and printer.
- The boot program loads the operating system into the computer's main memory or random access memory (RAM).
- An assembler takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations.
- A device driver controls a particular type of device that is attached to your computer, such as a keyboard or a mouse. The driver program converts the more general input/output instructions of the operating system to messages that the device type can understand.

Additionally, system software can also include system utilities, such as the disk defragmenterand System Restore, and development tools, such as compilers and debuggers.

Resource

A resource is any object which can be used by any process and can be allocated to any process within a system. Some examples of resources are processors (CPUs), disk space, input/output devices, files, memory (RAM), and so on.

Resource allocation in a computer system by operating system is a difficult task. Actually, the operating system distributes the resources between the competing programs in such a wonderful manner that all programs see the system as if it was dedicated to itself

Resource abstraction

Resource abstraction is the process of hiding the details of how the hardware operates, making computer hardware relatively easy for an application programmer to use.

Resource abstraction provides an abstract model of the operation of hardware components and generalizes hardware behavior.

It also limits the flexibility in which hardware can be manipulated.

It also provides a more convenient working environment for applications, by hiding some of the details of the hardware, and allowing the applications to operate at a higher level of abstraction.

One can view Operating Systems from two points of views: resource manager and extended machines. Form resource manager point of view operating systems manage the different parts of the system efficiently and from extended machines point of view operating systems provide a virtual machine to users that is more convenient to use.

Modern operating systems generally have following three major goals.

- To hide details of hardware by creating abstraction
- To allocate resources to processes (Manage resources)
- Provide a pleasant and effective user interface

OS strategies

Nutt [1997] identifies four common types of operating system strategies on which modern operating systems are built**: batch, timesharing, personal computing, and dedicated**.

According to Nutt, the favored strategy for any given computer depends on

- how the computer is to be used,
- the cost-effectiveness of the strategy implementation in the application environment,
- and the general state of the technology at the time the operating system is developed.

The table below summarizes the characteristics of each operating system strategy as described by Nutt :

Batch	This strategy involves reading a series of jobs (called a batch) into the machine and then executing the programs for each job in the batch. This approach does not allow users to interact with programs while they operate.
Timesharing	This strategy supports multiple interactive users. Rather than preparing a job for

	execution ahead of time, users establish an interactive session with the computer and then provide commands, programs and data as they are needed during the session.
Personal Computing	This strategy supports a single user running multiple programs on a dedicated machine. Since only one person is using the machine, more attention is given to establishing predictable response times from the system. This strategy is quite common today because of the popularity of personal computers.
Dedicated	This strategy supports real-time and process control systems. These are the types of systems which control satellites, robots, and air-traffic control. The dedicated strategy must guarantee certain response times for particular computing tasks or the application is useless.

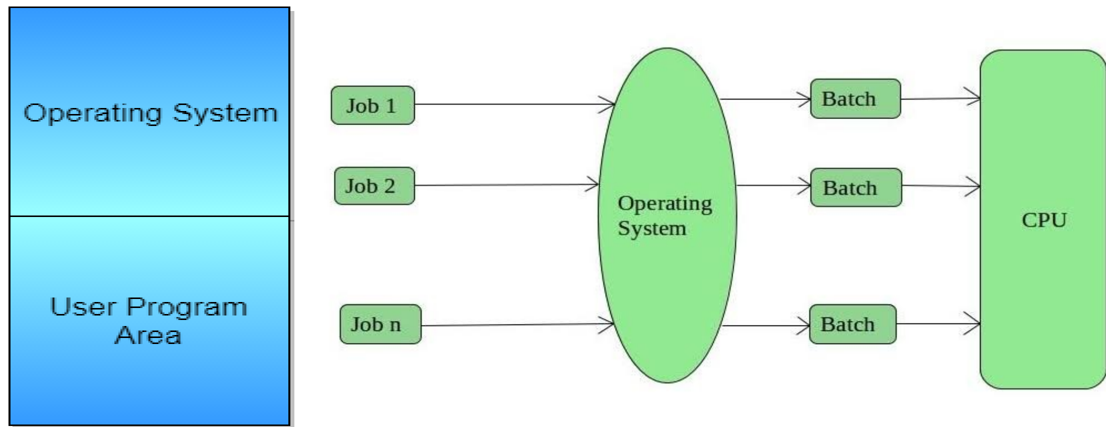
Types of Operating Systems

Simple Batch Systems

- In this type of system, there is **no direct interaction between user and the computer**.
- The user has to submit a job (written on cards or tape) to a computer operator.
- Then computer operator places a batch of several jobs on an input device.
- Jobs are batched together by type of languages and requirement.
- Then a special program, the monitor, manages the execution of each program in the batch.
- The monitor is always in the main memory and available for execution.

Advantages of Simple Batch Systems

1. No interaction between user and computer.
2. No mechanism to prioritise the processes.

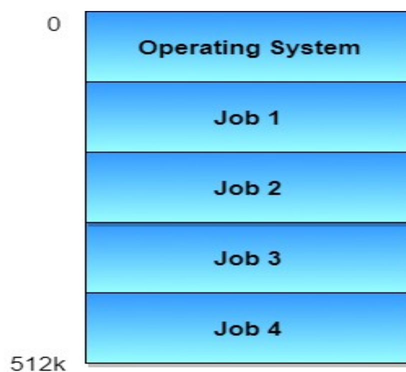


Multiprogramming Batch Systems

- In this the operating system picks up and begins to execute one of the jobs from memory.
- Once this job needs an I/O operation operating system switches to another job (CPU and OS always busy).
- Jobs in the memory are always less than the number of jobs on disk(Job Pool).
- If several jobs are ready to run at the same time, then the system chooses which one to run through the process of **CPU Scheduling**.
- In Non-multiprogrammed system, there are moments when CPU sits idle and does not do any work.
- In Multiprogramming system, CPU will never be idle and keeps on processing.

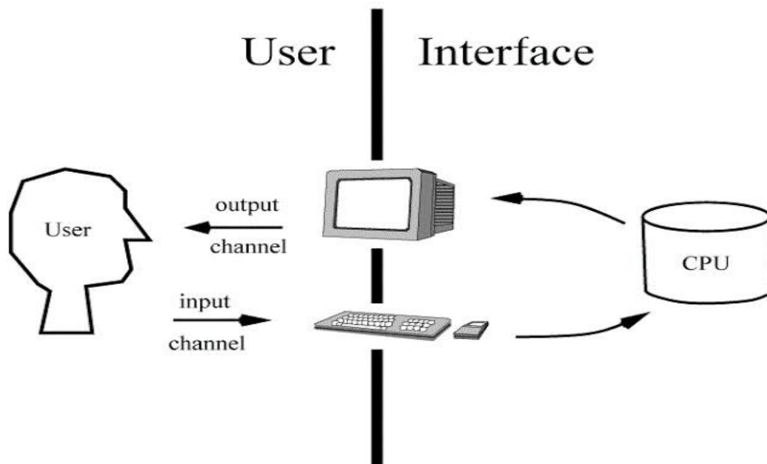
Time Sharing Systems are very similar to Multiprogramming batch systems. In fact time sharing systems are an extension of multiprogramming systems.

In Time sharing systems the prime focus is on **minimizing the response time**, while in multiprogramming the prime focus is to maximize the CPU usage.



Single user operating system

A single user operating system provides the facilities to be used on one computer by only one user. In other words, it supports one user at a time. However, it may support more than one profiles. Single keyboard and single monitor are used for the purpose of interaction. The most common example of a single user operating system is a system that is found in a typical home computer. Also, see the difference between Android and Windows.



Multi-User Operating System

A Multi-user **operating system** is a **computer** operating system which allows multiple users to access the single system with one operating system on it. It is generally used on large **mainframe** computers.
Example: Linux, Unix, Windows 2000, Ubuntu, Mac OS etc.,
In the multi-user operating system, different users connected at different terminals and we can access, these users through the network as shown in the diagram.

Features of the Multi-user Operating System

• Multi-tasking- Using multi-user operating system we can perform multiple tasks at a time, i.e. we can run more than one program at a time.
Example: we can edit a word document while browsing the internet.

• Resource sharing- we can share different peripherals like printers, hard drives or we can share a file or data. For this, each user is given a small time slice of CPU time.

• Background processing- It means that when commands are not processed firstly, then they are

executed in the background while another programs are interacting with the system in the real time.

Types of Multi-user Operating System

A multi-user operating system is of 3 types which are as follows:
1. **Distributed Systems**: in this, different computers are managed in such a way so that they can appear as a single computer. So, a sort of network is formed through which they can communicate with each other.
2. **Time-sliced Systems**: in this, a short period is assigned to each task, i.e. each user is given a time slice of the CPU time. As we know these time slices are tiny, so it appears to the users that they all are using the mainframe computer at the same time.
3. **Multiprocessor Systems**: in this, the operating system utilises more than one processor.
Example: Linux, Unix, Windows XP

Difference Between Single user and Multi user System

Single user Operating System	Multi-user Operating System
It is an operating system in which the user can manage one thing at a time effectively.	It is an operating system in which multiple users can manage multiple resources at a time
Example: MS DOS	**Example**: Linux, Unix, windows 2000, windows 2003 etc.
Single user Operating System has two types: Single user Single task Operating System and Single user Multi task Operating System.	It is of three types: time-sharing operating system, distributed operating system and multiprocessor system.
It is simple.	It is complex.
It provides a platform for one user at a time.	It provides controlled access for the number of users by maintaining a database of known users.
If another user wants to access the computer resources, then he/she has to wait until the current process completes.	There is no need to wait for accessing the computer resources.
This type of operating system is used for single user.	This type of operating system is used for multiple users.
In this, sometimes CPU is utilised to its maximum limit.	The operating system stimulates real-time performance by task switching.
It supports standalone systems.	It doesn't support standalone systems.

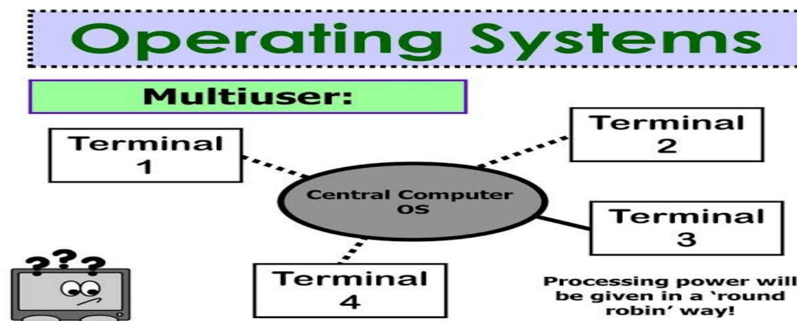
It is the operating system which maximum people use on their personal computers or laptops.	It is the operating system which is most of the time used in mainframe computers.
In this, there is no need to take care of balance between users.	In this, we have to take care of balance between users so that if one problem arises with one user does not affect other users also.

Advantages of the Multi-user Operating System

- When one computer in the network gets affected, then it does not affect another computer in the network. So, the system can be handled efficiently.
- Also, different users can access the same document on their computer. Example: if one computer contains the pdf file which the other user wants to access, then the other user can access that file.
- We use the multi-user operating system in the printing process so that different users can access the same printer and regular operating system can not do this process.
- Airlines also use this operating system for ticket reservation.
- We make use of the multiuser operating system in teachers and library staff for handling and searching for books. In this, the book record is stored in one computer while the other systems which are connected can access that computer for querying of books.

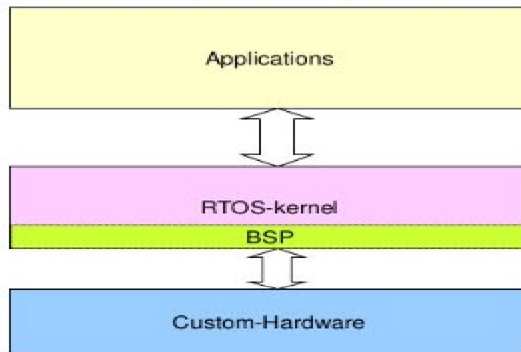
Disadvantages of the Multi-user Operating System

- Sometimes sharing your data becomes dangerous for you as your private Data also gets shared.
- Virus attacking takes place on all computer simultaneously as the computers are shared. So if one computer is affected then other also gets affected.
- Also, computer information is shared.



Real Time Operating System

The RTOS is an operating system, it is a brain of the real-time system and its response to inputs immediately. In the RTOS, the task will be completed by the specified time and its responses in a predictable way to unpredictable events. The structure of the RTOS is shown below.



Types of RTOS

There are three different types of RTOS which are following

- Soft real-time operating system
- Hard real-time operating system
- Firm real-time operating system

Soft Real-Time Operating System

The soft real-time operating system has certain deadlines, may be missed and they will take the action at a time t=0+. The soft real-time operating system is a type of OS and it does not contain constrained to extreme rules. The critical time of this operating system is delayed to some extent. The examples of this operating system are the digital camera, mobile phones and online data etc.

Hard Real-Time Operating System

This is also a type of OS and it is predicted by a deadline. The predicted deadlines will react at a time t = 0. Some examples of this operating system are air bag control in cars, anti-lock brake, and engine control system etc.

Firm Real-Time Operating System

In the firm real-time, an operating system has certain time constraints, they are not strict and it may cause undesired effects.Examples of this operating system are a visual inspection in industrial automation.

UNIT II

FACTORS TO CONSIDER WHEN CHOOSING AN OPERATING SYSTEM

When selecting an operating system for a computer, the following factors may be considered:

1.The hardware configuration of a computer e.g. memory capacity, processor speed andhard disk capacity.

2.The type of computer in terms of size and make. For example, some earlier Applecomputers would not run on Microsoft Operating systems

3.The application software intended for the computer

4.User – friendliness of the operating system

5.The documentation available

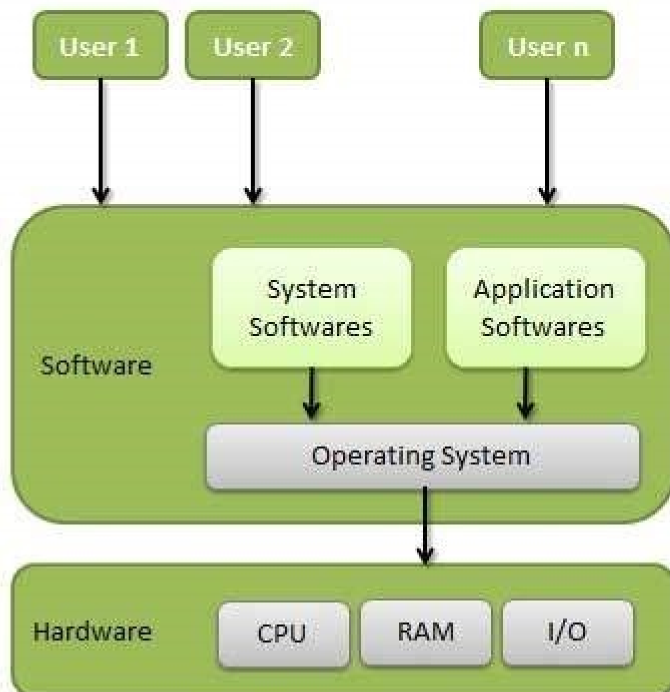
6.The cost of the operating system

7.Reliability and security provided by the operating system

8.The number of processors and hardware it can support

9.The number of users it can support.

BASIC OS FUNCTIONS:



Following are some of important functions of an operating System.

- Memory Management
- Processor Management
- Device Management

- File Management
- Security
- Control over system performance
- Job accounting
- Error detecting aids
- Coordination between other software and users

Memory Management

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.

Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must in the main memory. An Operating System does the following activities for memory management −

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part are not in use.
- In multiprogramming, the OS decides which process will get memory when and how much.
- Allocates the memory when a process requests it to do so.
- De-allocates the memory when a process no longer needs it or has been terminated.

Processor Management

In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called **process scheduling**. An Operating System does the following activities for processor management −

- Keeps tracks of processor and status of process. The program responsible for this task is known as **traffic controller**.
- Allocates the processor (CPU) to a process.
- De-allocates processor when a process is no longer required.

Device Management

An Operating System manages device communication via their respective drivers. It does the following activities for device management −

- Keeps tracks of all devices. Program responsible for this task is known as the **I/O controller**.
- Decides which process gets the device when and for how much time.
- Allocates the device in the efficient way.
- De-allocates devices.

File Management

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

An Operating System does the following activities for file management −

- Keeps track of information, location, uses, status etc. The collective facilities are often known as **file system**.
- Decides who gets the resources.
- Allocates the resources.
- De-allocates the resources.

Other Important Activities

Following are some of the important activities that an Operating System performs −

- **Security** − By means of password and similar other techniques, it prevents unauthorized access to programs and data.
- **Control over system performance** − Recording delays between request for a service and response from the system.
- **Job accounting** − Keeping track of time and resources used by various jobs and users.
- **Error detecting aids** − Production of dumps, traces, error messages, and other debugging and error detecting aids.
- **Coordination between other softwares and users** − Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

Processes in Operating Systems

- A program in execution
- May be stopped and later restarted by the OS
- Process table
  - Records information about each process
  - Program code
  - Data
  - Stack
  - Program counter (PC), Stack pointer (SP), and other registers
- Core image

- Process has a parent [process] and may create child processes
- Communication through messages
- Process ownership determined by the UID and GID
- Process id identifies the process
- Special process id's in Unix
    - 0 Swapper
    - 1 /sbin/init
    - 2 Pagedaemon

Process execution modes in Unix

Two modes of process execution: user mode and kernel mode. Normally, a process executes in the user mode. When a process executes a system call, the mode of execution changes from user mode to kernel mode. The bookkeeping operations related to the user process (interrupt handling, process scheduling, memory management) are performed in kernel mode.

User mode

- Processes can access their own instructions and data but not kernel instructions and data
- Cannot execute certain privileged machine instructions

Kernel mode

- Processes can access both kernel as well as user instructions and data
- No limit to which instructions can be executed
- Runs on behalf of a user process and is a part of the user process

User Mode

The system is in user mode when the operating system is running a user application such as handling a text editor. The transition from user mode to kernel mode occurs when the application requests the help of operating system or an interrupt or a system call occurs.

The mode bit is set to 1 in the user mode. It is changed from 1 to 0 when switching from user mode to kernel mode.

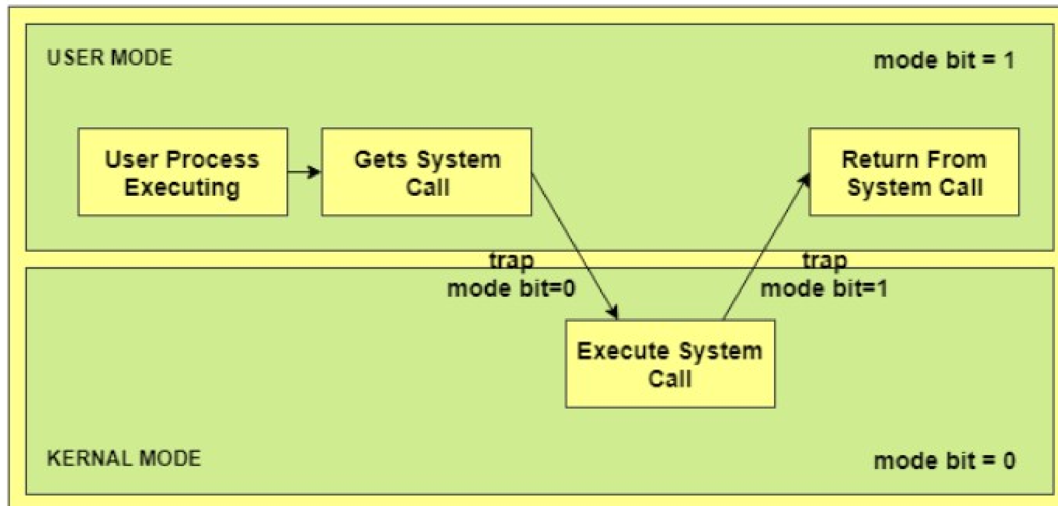
Kernel Mode

The system starts in kernel mode when it boots and after the operating system is loaded, it executes applications in user mode. There are some privileged instructions that can only be executed in kernel mode.

These are interrupt instructions, input output management etc. If the privileged instructions are executed in user mode, it is illegal and a trap is generated.

The mode bit is set to 0 in the kernel mode. It is changed from 0 to 1 when switching from kernel mode to user mode.

An image that illustrates the transition from user mode to kernel mode and back again is:

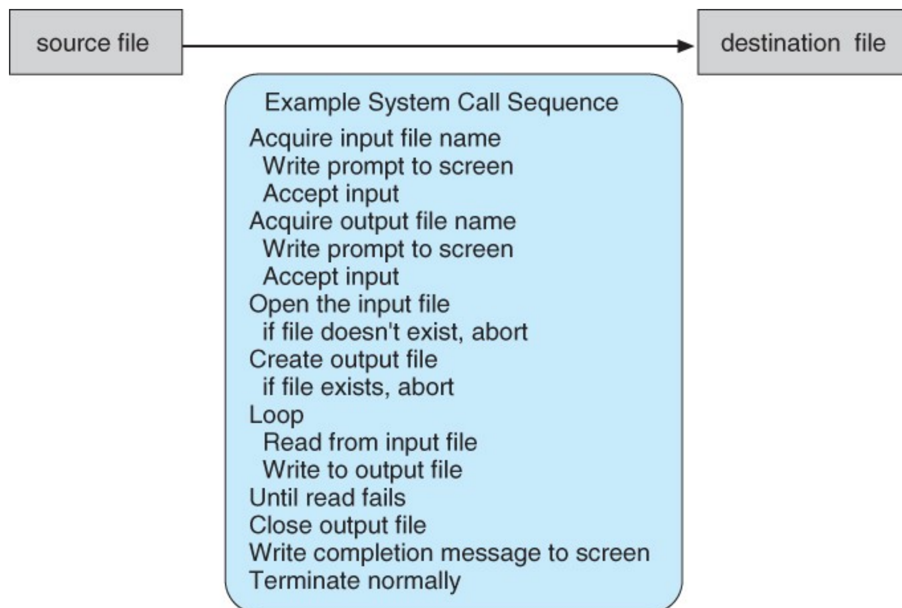


In the above image, the user process executes in the user mode until it gets a system call. Then a system trap is generated and the mode bit is set to zero. The system call gets executed in kernel mode. After the execution is completed, again a system trap is generated and the mode bit is set to 1. The system control returns to kernel mode and the process execution continues.

System Calls:

System calls provide a means for user or application programs to call upon the services of the operating system. The system call provides an interface to the operating system services.

The following figure illustrates the sequence of system calls required to copy a file:



Application developers often do not have direct access to the system calls, but can access them through an application programming interface (API). The functions that are included in the API invoke the actual system calls.

Types of System Calls

There are 5 different categories of system calls:

process control, file manipulation, device manipulation, information maintenance and communication.

Process Control

A running program needs to be able to stop execution either normally or abnormally. When execution is stopped abnormally, often a dump of memory is taken and can be examined with a debugger.

File Management

Some common system calls are *create*, *delete*, *read*, *write*, *reposition*, or *close*. Also, there is a need to determine the file attributes – *get* and *set* file attribute. Many times the OS provides an API to make these system calls.

Device Management

Process usually require several resources to execute, if these resources are available, they will be granted and control returned to the user process. These resources are also thought of as devices. Some are physical, such as a video card, and others are abstract, such as a file.

User programs *request* the device, and when finished they *release* the device. Similar to files, we can *read*, *write*, and *reposition* the device.

Information Management

Some system calls exist purely for transferring information between the user program and the operating system. An example of this is *time*, or *date*.

The OS also keeps information about all its processes and provides system calls to report this information.

Communication

There are two models of interprocess communication, the message-passing model and the shared memory model.

- Message-passing uses a common mailbox to pass messages between processes.
- Shared memory use certain system calls to create and gain access to create and gain access to regions of memory owned by other processes. The two processes exchange information by reading and writing in the shared data.

System Programs

These programs are not usually part of the OS kernel, but are part of the overall operating system.

- System programs may be divided into these categories:
  - **File management** - programs to create, delete, copy, rename, print, list, and generally manipulate files and directories.
  - **Status information** - Utilities to check on the date, time, number of users, processes running, data logging, etc. System **registries** are used to store and recall configuration information for particular applications.
  - **File modification** - e.g. text editors and other tools which can change file contents.
  - **Programming-language support** - E.g. Compilers, linkers, debuggers, profilers, assemblers, library archive management, interpreters for common languages, and support for make.
  - **Program loading and execution** - loaders, dynamic loaders, overlay loaders, etc., as well as interactive debuggers.
  - **Communications** - Programs for providing connectivity between processes and users, including mail, web browsers, remote logins, file transfers, and remote command execution.
  - **Background services** - System daemons are commonly started when the system is booted, and run for as long as the system is running, handling necessary services. Examples include network daemons, print servers, process schedulers, and system error monitoring services.
- Most operating systems today also come complete with a set of **application programs** to provide additional services, such as copying files or checking the time and date.
- Most users' views of the system is determined by their command interpreter and the application programs.

System Calls and System Programs

System calls provide an interface between the process an the operating system. System calls allow user-level processes to request some services from the operating system which process itself is not allowed to do.

System programs provide basic functioning to users so that they do not need to write their own environment for program development (editors, compilers) and program execution (shells). In some sense, they are bundles of useful system calls.

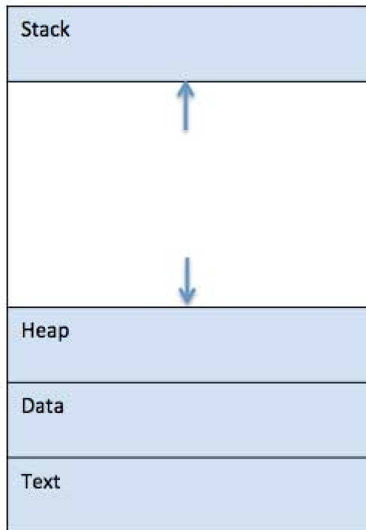
UNIT III

Process

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

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

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



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

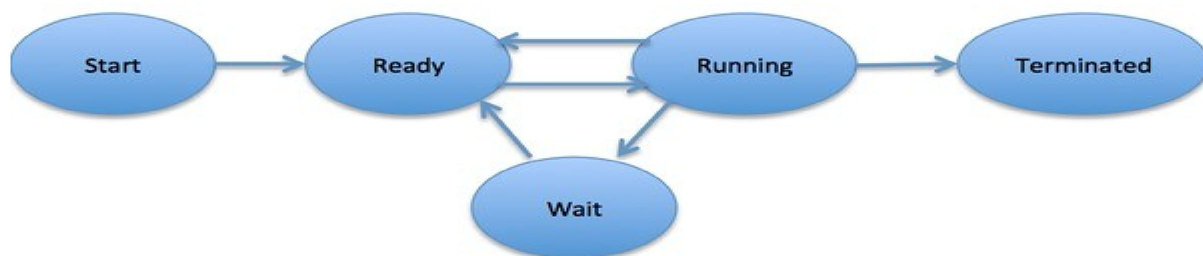
Process Life Cycle

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

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

S.N.	State & Description
1	**Start** This is the initial state when a process is first started/created.

2	**Ready** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3	**Running** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
4	**Waiting** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
5	**Terminated or Exit** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.



The Process Address Space

The process address space consists of the linear address range presented to each process and, more importantly, the addresses within this space that the process is allowed to use. Each process is given a *flat* 32- or 64-bit address space, with the size depending on the architecture. The term "flat" describes the fact that the address space exists in a single range. (As an example, a 32-bit address space extends from the address 0 to 429496729.) Some operating systems provide a *segmented address space*, with addresses existing not in a single linear range, but instead in multiple segments. Modern virtual memory operating systems generally have a flat memory model and not a segmented one. Normally, this flat address space is unique to each process. A memory address in one process's address space tells nothing of that memory address in another process's address space. Both processes can have different data at the same address in their respective address spaces. Alternatively, processes can elect to share their address space with other processes. We know these processes as *threads*.

A memory address is a given value within the address space, such as 4021f000. This particular value identifies a specific byte in a process's 32-bit address space. The interesting part of the address space is the intervals of memory addresses, such as 08048000-0804c000, that the process has permission to access. These intervals of legal addresses are called *memory areas*. The process, through the kernel, can dynamically add and remove memory areas to its address space.

The process can access a memory address only in a valid memory area. Memory areas have associated permissions, such as readable, writable, and executable, that the associated process must respect. If a process accesses a memory address not in a valid memory area, or if it accesses a valid area in an invalid manner, the kernel kills the process with the dreaded "Segmentation Fault" message.

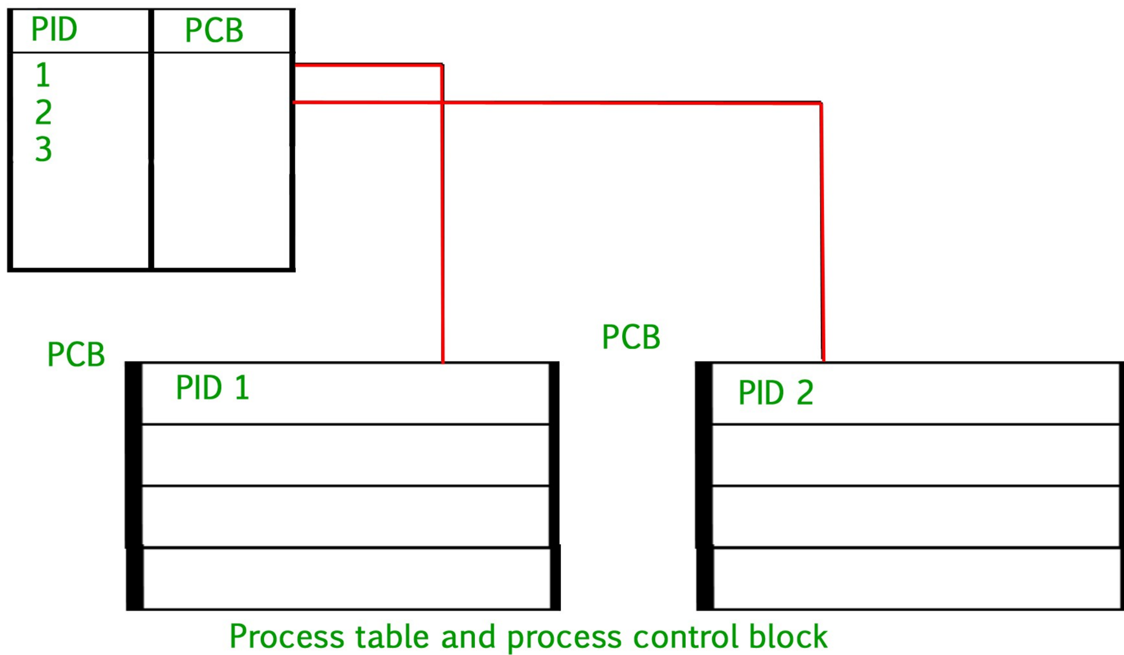
Memory areas can contain all sorts of goodies, such as

- A memory map of the executable file's code, called the *text section*
- A memory map of the executable file's initialized global variables, called the *data section*
- A memory map of the zero page (a page consisting of all zeros, used for purposes such as this) containing uninitialized global variables, called the *bss section*[1]
- A memory map of the zero page used for the process's user-space stack (do not confuse this with the process's kernel stack, which is separate and maintained and used by the kernel)
- An additional text, data, and bss section for each shared library, such as the C library and dynamic linker, loaded into the process's address space
- Any memory mapped files
- Any shared memory segments
- Any anonymous memory mappings, such as those associated with malloc()[2]

Process Table and Process Control Block (PCB)

While creating a process the operating system performs several operations. To identify these process, it must identify each process, hence it assigns a process identification number (PID) to each process. As the operating system supports multi-programming, it needs to keep track of all the processes. For this task, the process control block (PCB) is used to track the process's execution status. Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, scheduling algorithms, etc. All these information is required and must be saved when the process is switched from one state to another. When the process made transitions from one state to another, the operating system must update information in the process's PCB.

A process control block (PCB) contains information about the process, i.e. registers, quantum, priority, etc. The process table is an array of PCB's, that means logically contains a PCB for all of the current processes in the system.



Process table and process control block

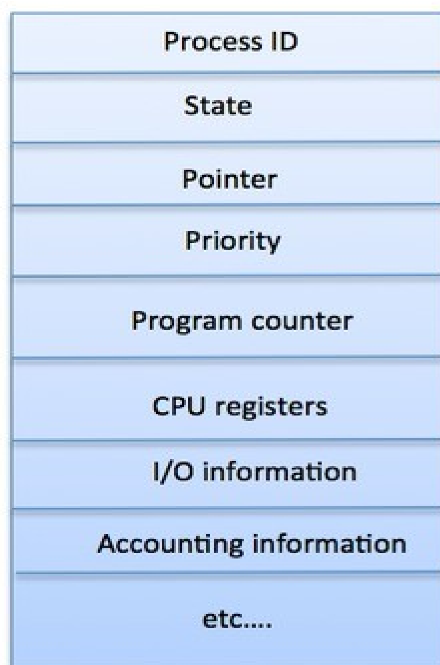
Process Control Block (PCB)

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

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

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

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



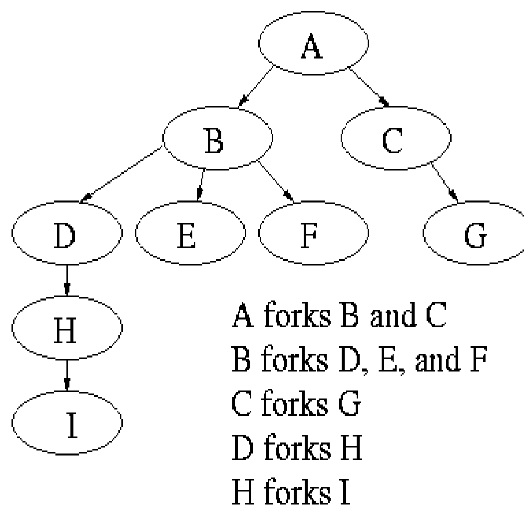
- The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

Process Hierarchies

Modern general purpose operating systems permit a user to create and destroy processes.

- In unix this is done by the **fork** system call, which creates a **child** process, and the **exit** system call, which terminates the current process.
- After a fork both parent and child keep running (indeed they have the *same* program text) and each can fork off other processes.
- A process tree results. The root of the tree is a special process created by the OS during startup.
- A process can *choose* to wait for children to terminate. For example, if C issued a wait() system call it would block until G finished.

Old or primitive operating system like MS-DOS are not multiprogrammed so when one process starts another, the first process is *automatically* blocked and waits until the second is finished.



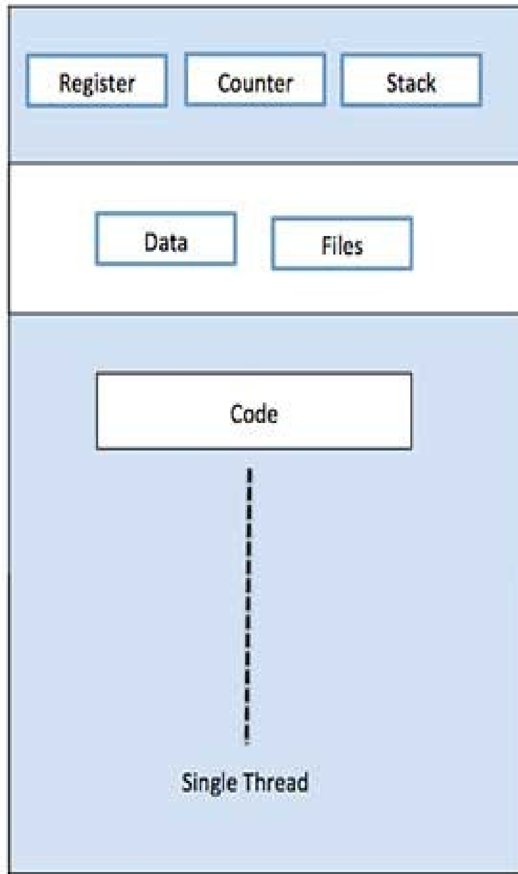
Thread

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

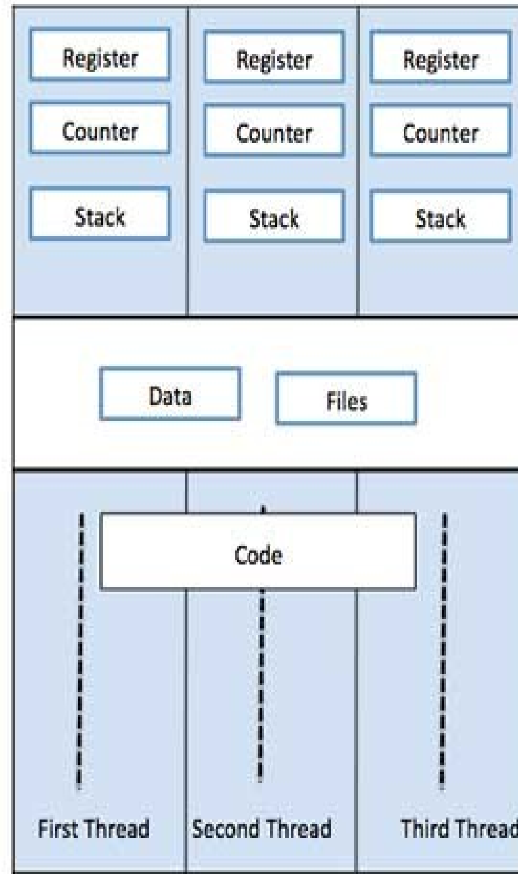
A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

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

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



Single Process P with single thread



Single Process P with three threads

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process	While one thread is blocked and waiting, a second

	can execute until the first process is unblocked.	thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread

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

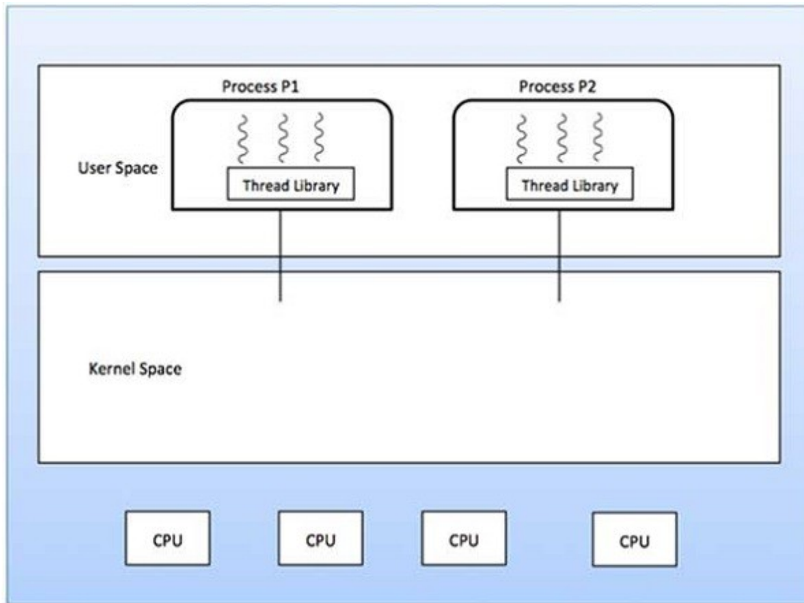
Types of Thread

Threads are implemented in following two ways −

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

User Level Threads

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



Advantages

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

Disadvantages

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

Kernel Level Threads

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

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

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

Disadvantages

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

Multithreading Models

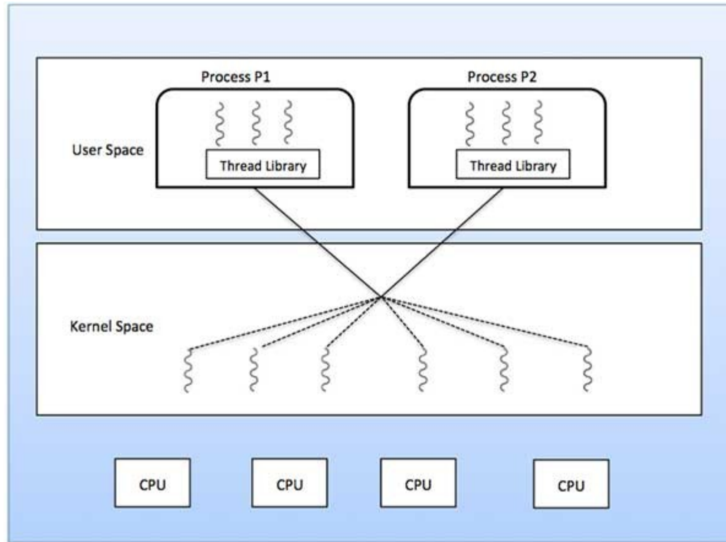
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the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

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

Many to Many Model

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

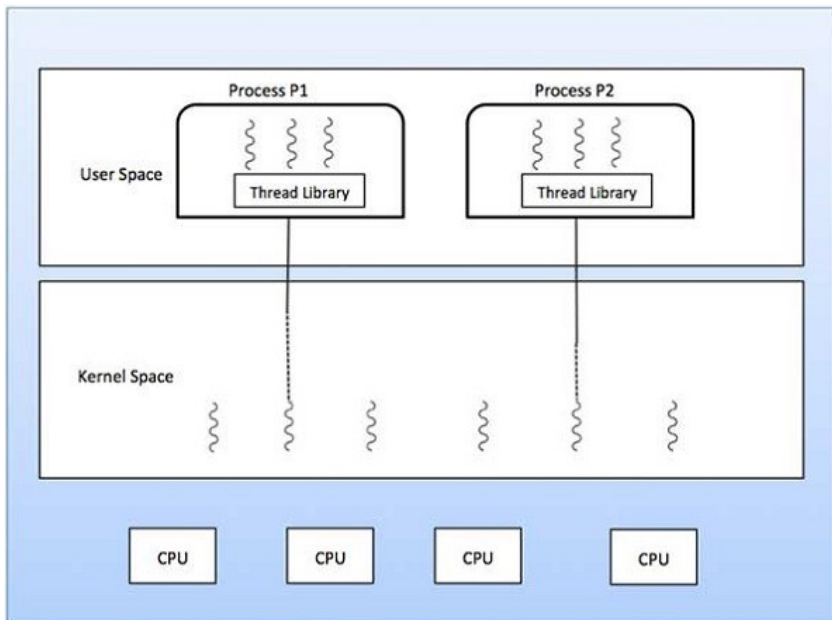
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model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



Many to One Model

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

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

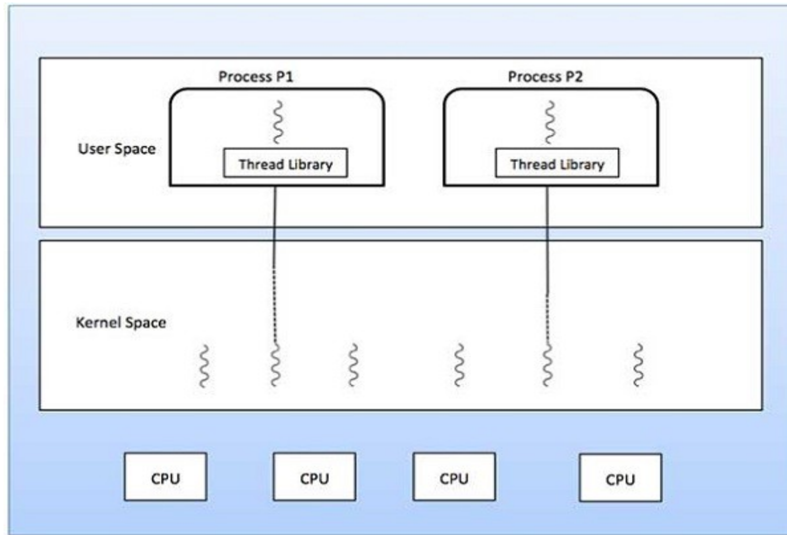


One to One Model

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run

when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

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



Difference between User-Level & Kernel-Level Thread

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

Process Scheduling

The act of determining which process is in the **ready** state, and should be moved to the **running** state is known as **Process Scheduling**.

The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs. For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.

Scheduling fell into one of the two general categories:

- **Non Pre-emptive Scheduling:** When the currently executing process gives up the CPU voluntarily.
- **Pre-emptive Scheduling:** When the operating system decides to favour another process, pre-empting the currently executing process.

Preemptive Scheduling

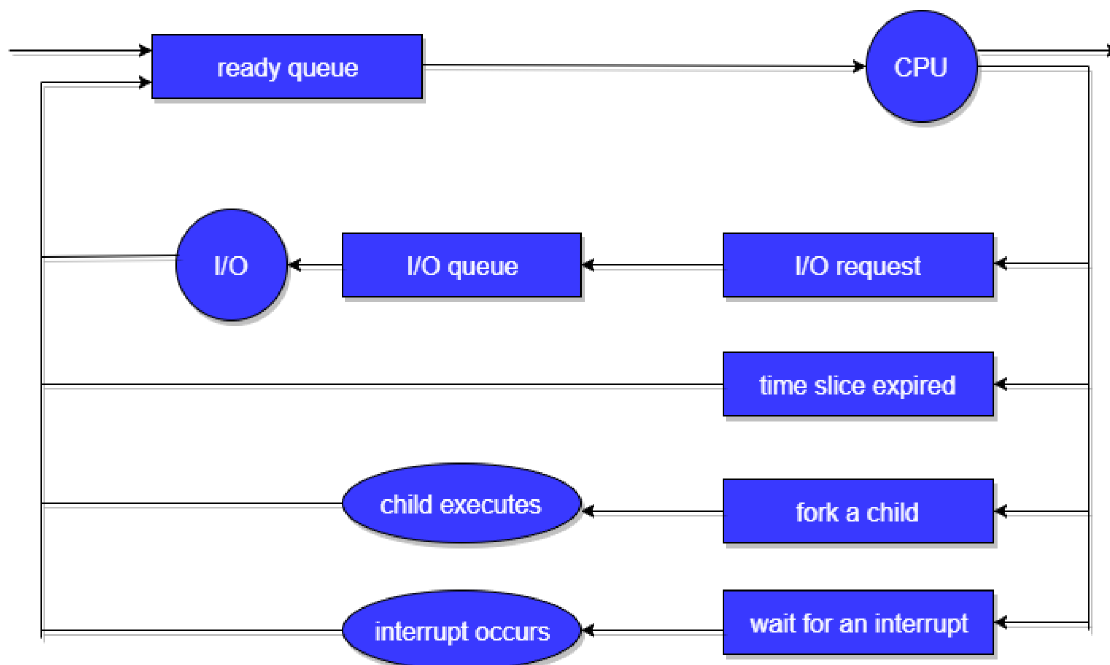
- CPU scheduling decisions take place under one of four conditions:
  1. When a process switches from the running state to the waiting state, such as for an I/O request or invocation of the wait( ) system call.
  2. When a process switches from the running state to the ready state, for example in response to an interrupt.
  3. When a process switches from the waiting state to the ready state, say at completion of I/O or a return from wait( ).
  4. When a process terminates.
- For conditions 1 and 4 there is no choice - A new process must be selected.
- For conditions 2 and 3 there is a choice - To either continue running the current process, or select a different one.
- If scheduling takes place only under conditions 1 and 4, the system is said to be ***non-preemptive***, or ***cooperative***. Under these conditions, once a process starts running it keeps running, until it either voluntarily blocks or until it finishes. Otherwise the system is said to be ***preemptive.***

Scheduling Queues

- All processes, upon entering into the system, are stored in the **Job Queue**.
- Processes in the Ready state are placed in the **Ready Queue**.
- Processes waiting for a device to become available are placed in **Device Queues**. There are unique device queues available for each I/O device.

A new process is initially put in the **Ready queue**. It waits in the ready queue until it is selected for execution(or dispatched). Once the process is assigned to the CPU and is executing, one of the following several events can occur:

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



In the first two cases, the process eventually switches from the waiting state to the ready state, and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Types of Schedulers

There are three types of schedulers available:

1. Long Term Scheduler
2. Short Term Scheduler
3. Medium Term Scheduler

Long Term Scheduler

Long term scheduler runs less frequently. Long Term Schedulers decide which program must get into the job queue. From the job queue, the Job Processor, selects processes and loads them into the memory for execution. Primary aim of the Job Scheduler is to maintain a good degree of Multiprogramming. An optimal degree of Multiprogramming means the average rate of process creation is equal to the average departure rate of processes from the execution memory.

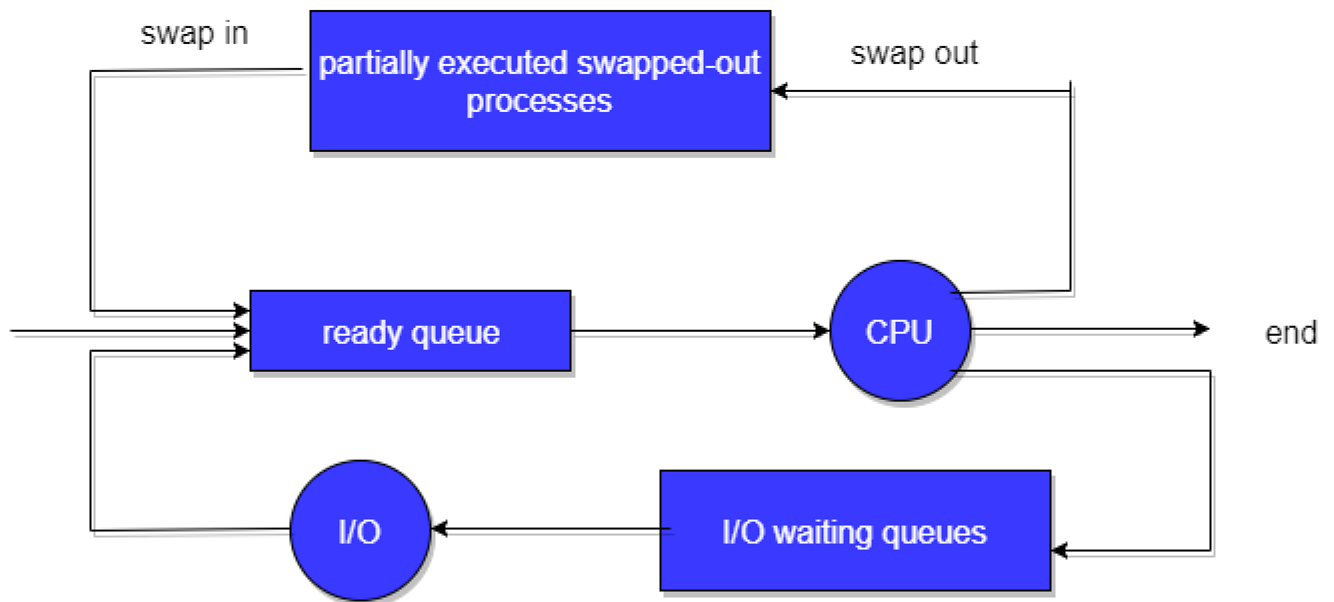
Short Term Scheduler

This is also known as CPU Scheduler and runs very frequently. The primary aim of this scheduler is to enhance CPU performance and increase process execution rate.

Medium Term Scheduler

This scheduler removes the processes from memory (and from active contention for the CPU), and thus reduces the degree of multiprogramming. At some later time, the process can be reintroduced into memory and its execution van be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium term scheduler.

Swapping may be necessary to improve the process mix, or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up. This complete process is descripted in the below diagram:



Comparison among Scheduler

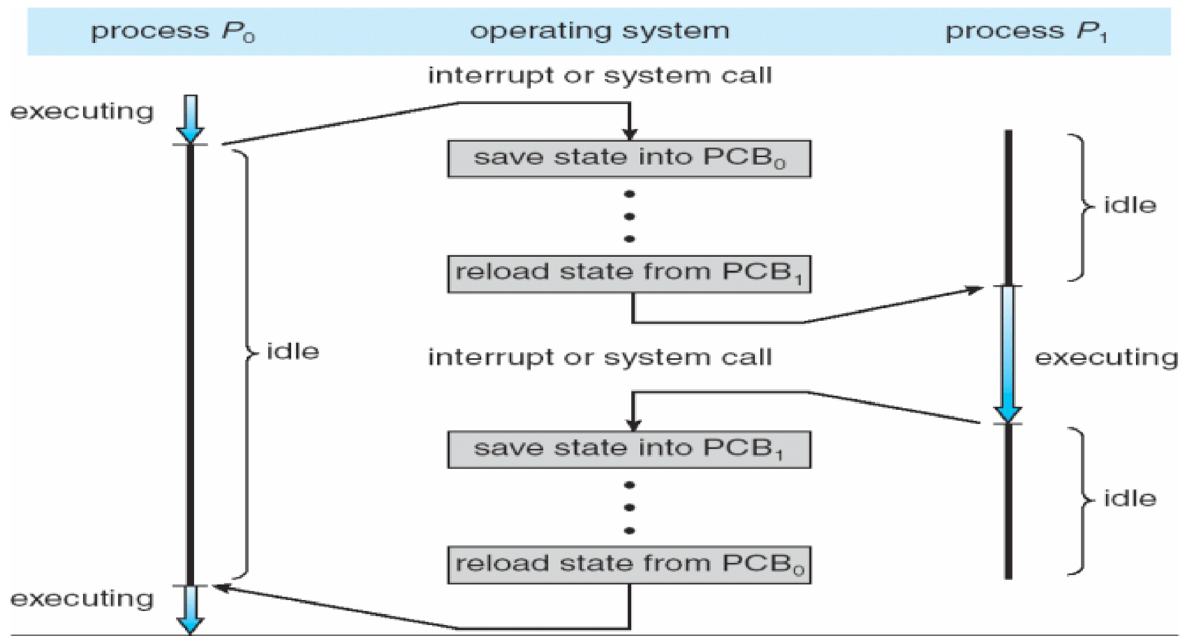
S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping

			scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.



Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

- Program Counter
- Scheduling information
- Base and limit register value
- Currently used register
- Changed State
- I/O State information
- Accounting information

Dispatcher

- The **dispatcher** is the module that gives control of the CPU to the process selected by the scheduler. This function involves:
  - Switching context.
  - Switching to user mode.
  - Jumping to the proper location in the newly loaded program.

- The dispatcher needs to be as fast as possible, as it is run on every context switch. The time consumed by the dispatcher is known as **dispatch latency.**

Scheduling Criteria

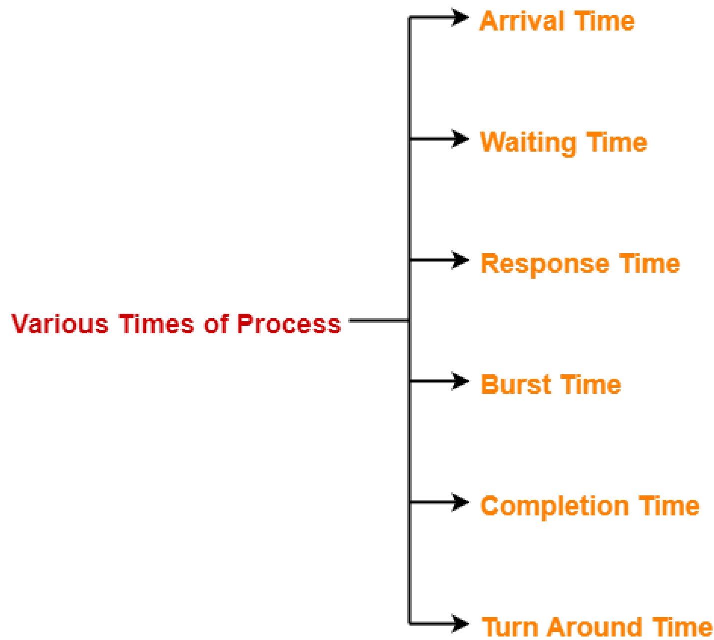
- There are several different criteria to consider when trying to select the "best" scheduling algorithm for a particular situation and environment, including:
    - **CPU utilization** - Ideally the CPU would be busy 100% of the time, so as to waste 0 CPU cycles. On a real system CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)
    - **Throughput** - Number of processes completed per unit time. May range from 10 / second to 1 / hour depending on the specific processes.
    - **Turnaround time** - Time required for a particular process to complete, from submission time to completion. (Wall clock time.)
    - **Waiting time** - How much time processes spend in the ready queue waiting their turn to get on the CPU.
        - (**Load average** - The average number of processes sitting in the ready queue waiting their turn to get into the CPU. Reported in 1-minute, 5-minute, and 15-minute averages by "uptime" and "who".)

▪ Response time - In an interactive system, a process can produce some output early and can continue, computing new results. Previous results are being displayed the user. Thus another measure is the time from the submission to the request until the first response is produced. It is called Response time. The turnaround time is normally limited by speed of output device.

- In general one wants to optimize the average value of a criteria (Maximize CPU utilization and throughput, and minimize all the others.) However some times one wants to do something different, such as to minimize the maximum response time.
- Sometimes it is most desirable to minimize the ***variance*** of a criteria than the actual value. I.e. users are more accepting of a consistent predictable system than an inconsistent one, even if it is a little bit slower.

Various Times Related To Process-

Various times related to process are-



1. Arrival time
2. Waiting time
3. Response time
4. Burst time
5. Completion time
6. Turn Around Time

1. Arrival Time-

Arrival time is the point of time at which a process enters the ready queue.

2. Waiting Time-

Waiting time is the amount of time spent by a process waiting in the ready queue for getting the CPU.

Waiting time = Turn Around time – Burst time

3. Response Time-

Response time is the amount of time after which a process gets the CPU for the first time after entering the ready queue.

Response Time = Time at which process first gets the CPU – Arrival time

4. Burst Time-

Burst time is the amount of time required by a process for executing on CPU.

- It is also called as **execution time** or **running time**.
- Burst time of a process can not be known in advance before executing the process.
- It can be known only after the process has executed.

5. Completion Time-

Completion time is the point of time at which a process completes its execution on the CPU and takes exit from the system.

- It is also called as **exit time**.

6. Turn Around Time-

Turn Around time is the total amount of time spent by a process in the system.

- When present in the system, a process is either waiting in the ready queue for getting the CPU or it is executing on the CPU.

Turn Around time = Burst time + Waiting time

OR

Turn Around time = Completion time – Arrival time

Scheduling Algorithms

The following subsections will explain several common scheduling strategies, looking at only a single CPU burst each for a small number of processes. Obviously real systems have to deal with a lot more simultaneous processes executing their CPU-I/O burst cycles.

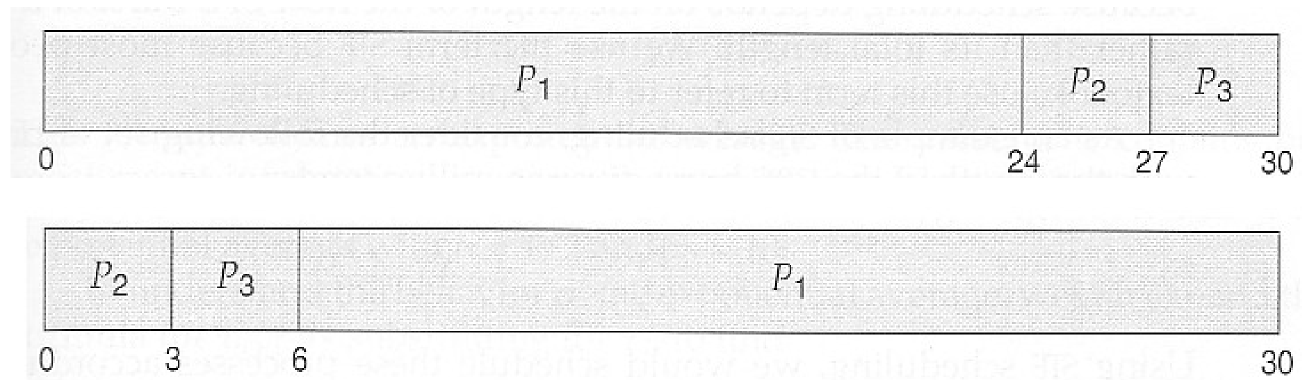
First-Come First-Serve Scheduling, FCFS

- FCFS is very simple - Just a FIFO queue, like customers waiting in line at the bank or the post office or at a copying machine.

- Unfortunately, however, FCFS can yield some very long average wait times, particularly if the first process to get there takes a long time. For example, consider the following three processes:

Process	Burst Time
P1	24
P2	3
P3	3

- In the first Gantt chart below, process P1 arrives first. The average waiting time for the three processes is (0 + 24 + 27) / 3 = 17.0 ms.
- In the second Gantt chart below, the same three processes have an average wait time of (0 + 3 + 6) / 3 = 3.0 ms. The total run time for the three bursts is the same, but in the second case two of the three finish much quicker, and the other process is only delayed by a short amount.



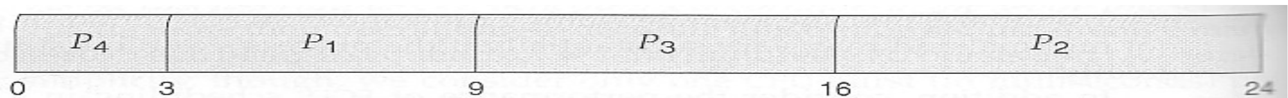
- FCFS can also block the system in a busy dynamic system in another way, known as the ***convoy effect***. When one CPU intensive process blocks the CPU, a number of I/O intensive processes can get backed up behind it, leaving the I/O devices idle. When the CPU hog finally relinquishes the CPU, then the I/O processes pass through the CPU quickly, leaving the CPU idle while everyone queues up for I/O, and then the cycle repeats itself when the CPU intensive process gets back to the ready queue.

Shortest-Job-First Scheduling, SJF

- The idea behind the SJF algorithm is to pick the quickest fastest little job that needs to be done, get it out of the way first, and then pick the next smallest fastest job to do next.
- (Technically this algorithm picks a process based on the next shortest **CPU burst**, not the overall process time.)

- For example, the Gantt chart below is based upon the following CPU burst times, (and the assumption that all jobs arrive at the same time.)

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

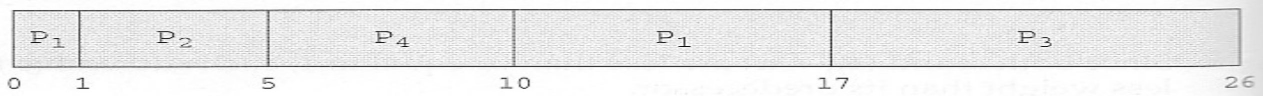


- In the case above the average wait time is (0 + 3 + 9 + 16) / 4 = 7.0 ms, (as opposed to 10.25 ms for FCFS for the same processes.)

- SJF can be either preemptive or non-preemptive. Preemption occurs when a new process arrives in the ready queue that has a predicted burst time shorter than the time remaining in the process whose burst is currently on the CPU. Preemptive SJF is sometimes referred to as **shortest remaining time first scheduling.**

- For example, the following Gantt chart is based upon the following data:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
p4	3	5



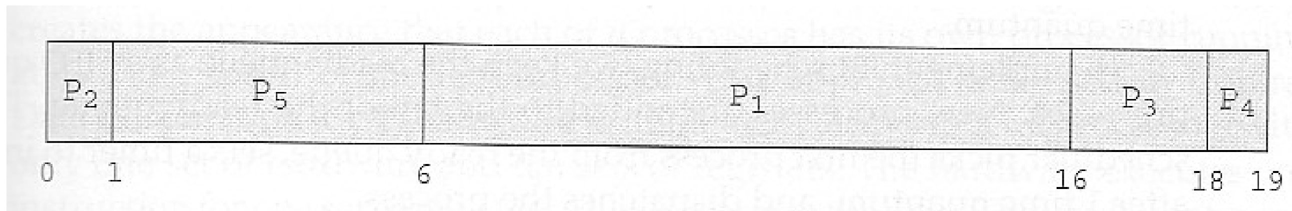
- The average wait time in this case is ((5 - 3) + (10 - 1) + (17 - 2)) / 4 = 26 / 4 = 6.5 ms. (As opposed to 7.75 ms for non-preemptive SJF or 8.75 for FCFS.)

Priority Scheduling

- Priority scheduling is a more general case of SJF, in which each job is assigned a priority and the job with the highest priority gets scheduled first. (SJF uses the inverse of the next expected burst time as its priority - The smaller the expected burst, the higher the priority.)

- Note that in practice, priorities are implemented using integers within a fixed range, but there is no agreed-upon convention as to whether "high" priorities use large numbers or small numbers. This book uses low number for high priorities, with 0 being the highest possible priority.
- For example, the following Gantt chart is based upon these process burst times and priorities, and yields an average waiting time of 8.2 ms:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



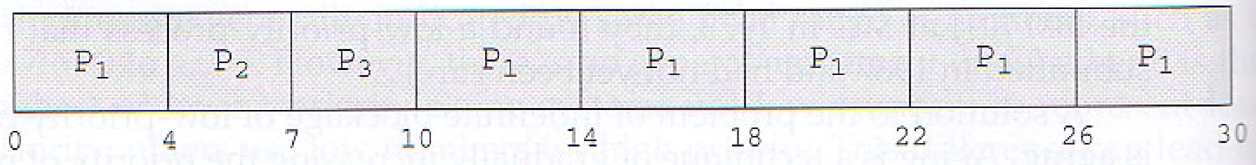
- Priorities can be assigned either internally or externally. Internal priorities are assigned by the OS using criteria such as average burst time, ratio of CPU to I/O activity, system resource use, and other factors available to the kernel. External priorities are assigned by users, based on the importance of the job, fees paid, politics, etc.
- Priority scheduling can be either preemptive or non-preemptive.
- Priority scheduling can suffer from a major problem known as ***indefinite blocking***, or ***starvation***, in which a low-priority task can wait forever because there are always some other jobs around that have higher priority.
    - If this problem is allowed to occur, then processes will either run eventually when the system load lightens (at say 2:00 a.m.), or will eventually get lost when the system is shut down or crashes. (There are rumors of jobs that have been stuck for years.)
    - One common solution to this problem is ***aging***, in which priorities of jobs increase the longer they wait. Under this scheme a low-priority job will eventually get its priority raised high enough that it gets run.

Round Robin Scheduling

- Round robin scheduling is similar to FCFS scheduling, except that CPU bursts are assigned with limits called ***time quantum***.

- When a process is given the CPU, a timer is set for whatever value has been set for a time quantum.
    - If the process finishes its burst before the time quantum timer expires, then it is swapped out of the CPU just like the normal FCFS algorithm.
    - If the timer goes off first, then the process is swapped out of the CPU and moved to the back end of the ready queue.
- The ready queue is maintained as a circular queue, so when all processes have had a turn, then the scheduler gives the first process another turn, and so on.
- RR scheduling can give the effect of all processors sharing the CPU equally, although the average wait time can be longer than with other scheduling algorithms. In the following example the average wait time is 5.66 ms.

Process	**Burst Time**
P1	24
P2	3
P3	3



- The performance of RR is sensitive to the time quantum selected. If the quantum is large enough, then RR reduces to the FCFS algorithm; If it is very small, then each process gets 1/nth of the processor time and share the CPU equally.
- **BUT**, a real system invokes overhead for every context switch, and the smaller the time quantum the more context switches there are. (See Figure 5.4 below.) Most modern systems use time quantum between 10 and 100 milliseconds, and context switch times on the order of 10 microseconds, so the overhead is small relative to the time quantum.

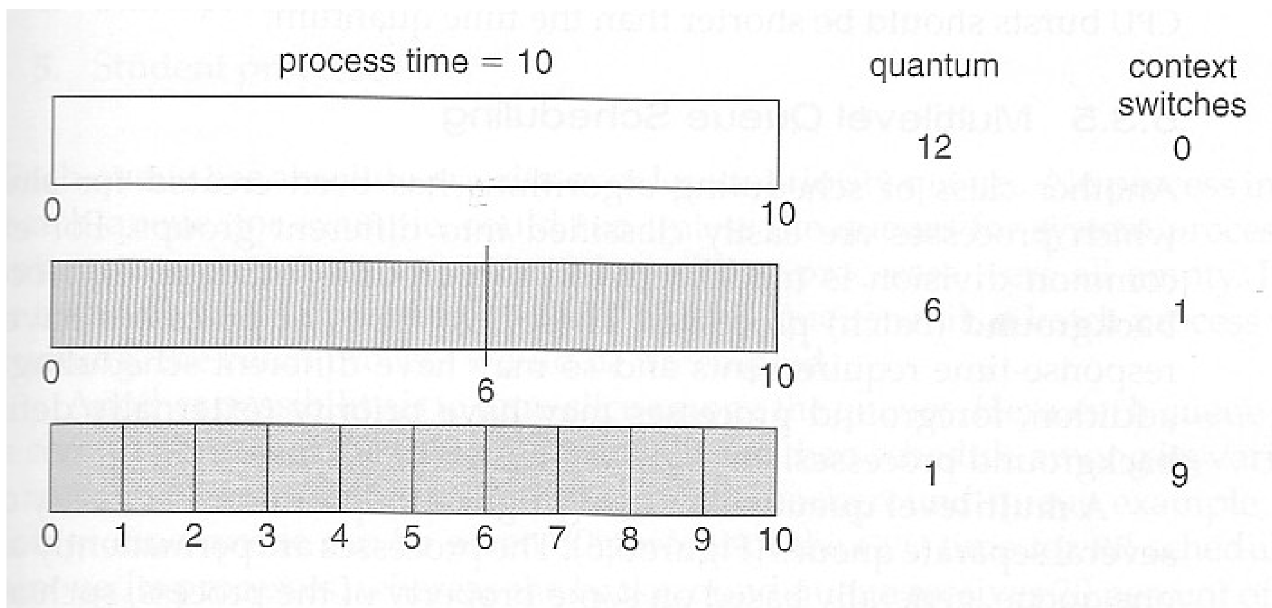
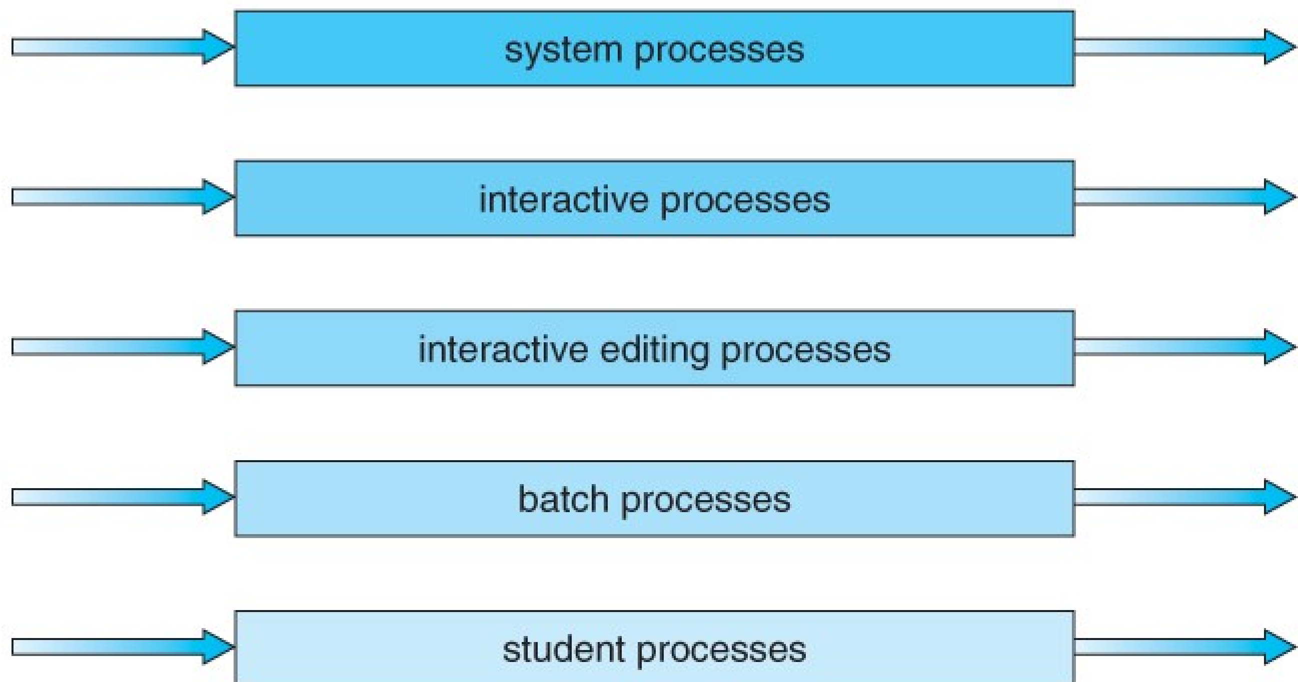


Figure 5.4 The way in which a smaller time quantum increases context switches.

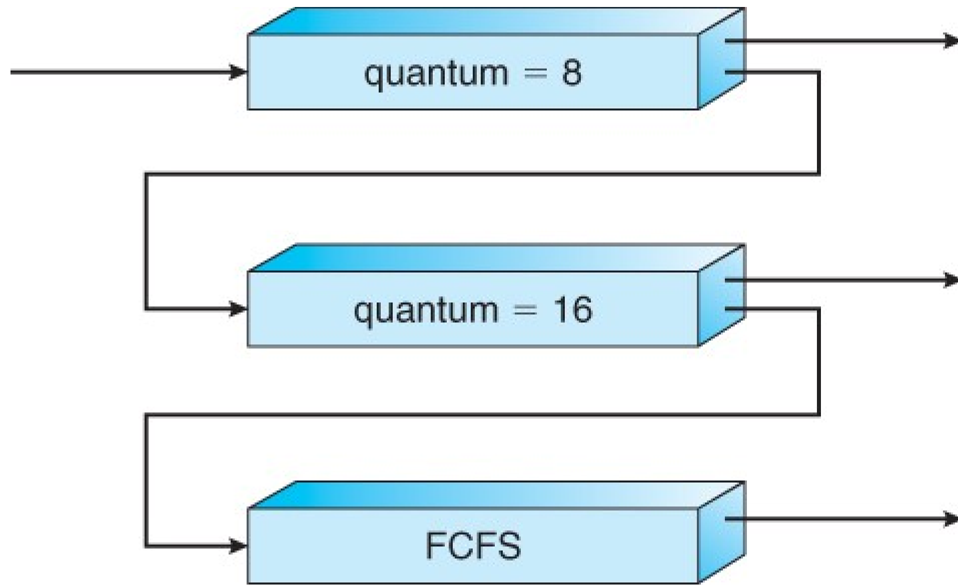
Multilevel Queue Scheduling

- When processes can be readily categorized, then multiple separate queues can be established, each implementing whatever scheduling algorithm is most appropriate for that type of job, and/or with different parametric adjustments.
- Scheduling must also be done between queues, that is scheduling one queue to get time relative to other queues. Two common options are strict priority (no job in a lower priority queue runs until all higher priority queues are empty) and round-robin (each queue gets a time slice in turn, possibly of different sizes.)



Multilevel Feedback-Queue Scheduling

- Multilevel feedback queue scheduling is similar to the ordinary multilevel queue scheduling described above, except jobs may be moved from one queue to another for a variety of reasons:
  - If the characteristics of a job change between CPU-intensive and I/O intensive, then it may be appropriate to switch a job from one queue to another.
  - Aging can also be incorporated, so that a job that has waited for a long time can get bumped up into a higher priority queue for a while.
- Multilevel feedback queue scheduling is the most flexible, because it can be tuned for any situation. But it is also the most complex to implement because of all the adjustable parameters. Some of the parameters which define one of these systems include:
  - The number of queues.
  - The scheduling algorithm for each queue.
  - The methods used to upgrade or demote processes from one queue to another. (Which may be different.)
  - The method used to determine which queue a process enters initially.


quantum = 8
quantum = 16
FCFS

UNIT – IV

MEMORY MANAGEMENT

Main Memory Management:

The main memory is central to the operation of a modern computer system. Main memory is a large array of words or bytes ranging in size from hundreds of thousand to billions. Main memory stores the quickly accessible data shared by the CPU & I/O device. The central processor reads instruction from main memory during instruction fetch cycle & it both reads & writes data from main memory during the data fetch cycle. The main memory is generally the only large storage device that the CPU is able to address & access directly. For example, for the CPU to process data from disk. Those data must first be transferred to main memory by CPU generated E/O calls. Instruction must be in memory for the CPU to execute them. The OS is responsible for the following activities in connection with memory management.

- Keeping track of which parts of memory are currently being used & by whom.
- Deciding which processes are to be loaded into memory when memory space becomes available.
- Allocating &deallocating memory space as needed.

Address Binding

Address binding of instructions and data to memory addresses can happen at three different stages.

- Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes. Example: .COM-format programs in MS-DOS.
- Load time: Must generate relocatable code if memory location is not known at compile time.
- Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., relocation registers).

The following figure shows the various stages of the binding processes and the units involved in each stage:

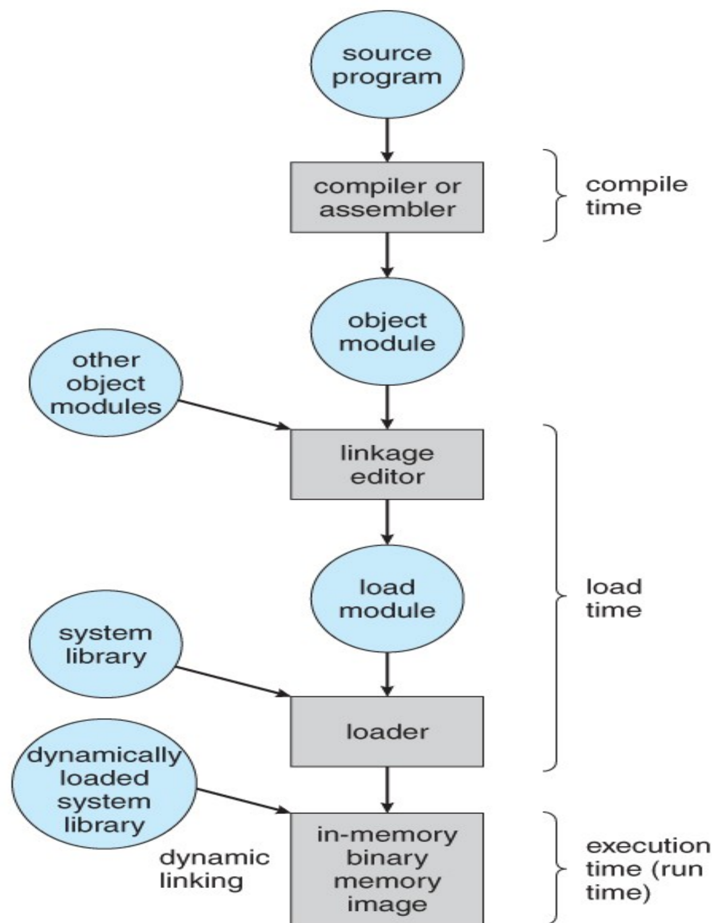


Figure- Multistep processing of a user program

Logical Versus Physical Address Space

- The address generated by the CPU is a logical address, whereas the address actually seen by the memory hardware is a physical address.
- Addresses bound at compile time or load time have identical logical and physical addresses.
- Addresses created at execution time, however, have different logical and physical addresses.
    - In this case the logical address is also known as a **virtual address**, and the two terms are used interchangeably by our text.
    - The set of all logical addresses used by a program composes the **logical address space**, and the set of all corresponding physical addresses composes the **physical address space.**

- The run time mapping of logical to physical addresses is handled by the **memory-management unit, MMU**.
    - The MMU can take on many forms. One of the simplest is a modification of the base-register scheme described earlier.
    - The base register is now termed a **relocation register**, whose value is added to every memory request at the hardware level.

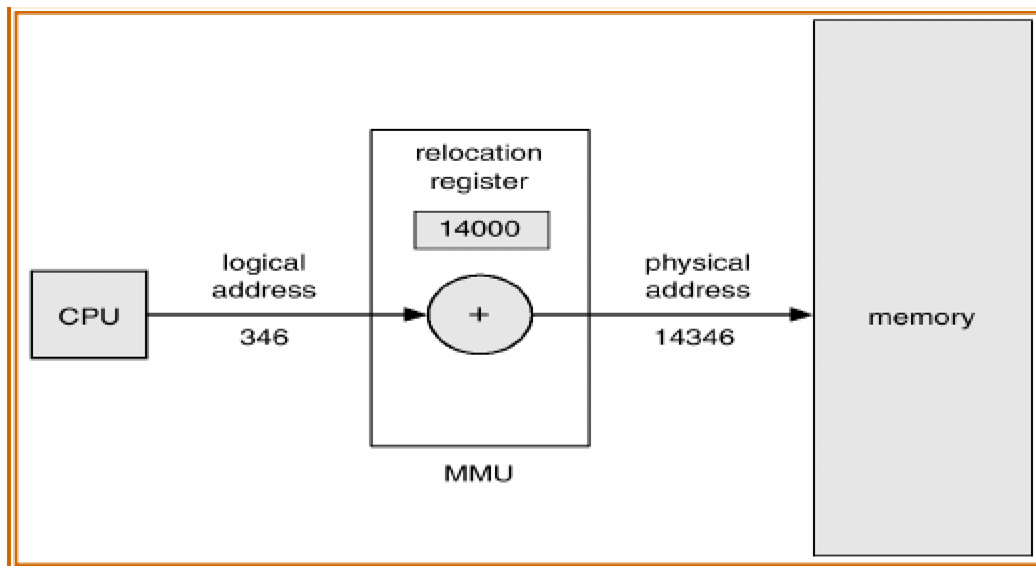


Figure - Dynamic relocation using a relocation register

Dynamic Loading

Rather than loading an entire program into memory at once, dynamic loading loads up each routine as it is called. The advantage is that unused routines need never be loaded, reducing total memory usage and generating faster program startup times.

Dynamic Linking

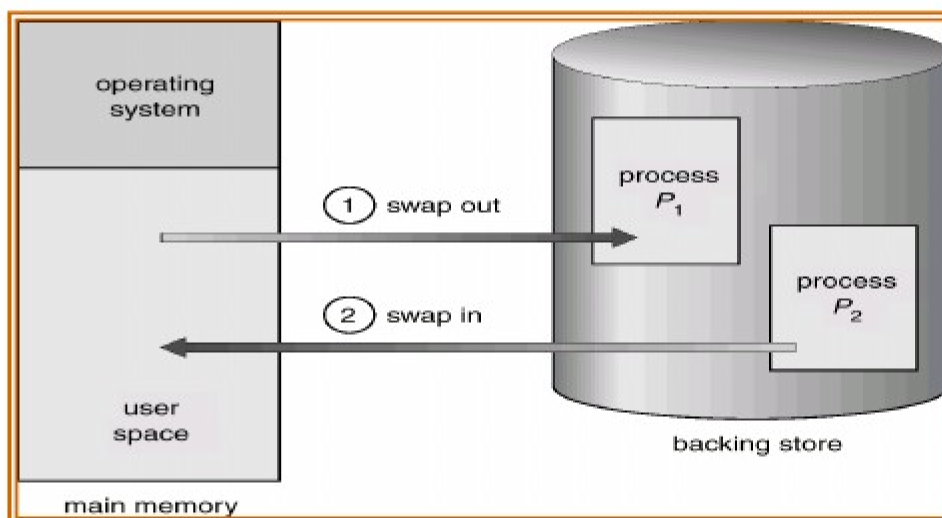
- With **static linking** library modules get fully included in executable modules, wasting both disk space and main memory usage, because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code.

- With **dynamic linking**, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time.

Swapping

- A process must be loaded into memory in order to execute.

- If there is not enough memory available to keep all running processes in memory at the same time, then some processes who are not currently using the CPU may have their memory swapped out to a fast local disk called the **backing store.**
- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. For example, assume a multiprogramming environment with a round robin CPU scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed. In the mean time, the CPU scheduler will allocate a time slice to some other process in memory. When each process finished its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. The quantum must also be sufficiently large that reasonable amounts of computing are done between swaps.
- Roll out, roll in – swapping variant used for priority-based scheduling algorithms. If a higher priority process arrives and wants service, the memory manager can swap out the lower priority process so that it can load and execute lower priority process can be swapped back in and continued. This variant is some times called roll out, roll in. Normally a process that is swapped out will be swapped back into the same memory space that it occupied previously. This restriction is dictated by the process cannot be moved to different locations. If execution time binding is being used, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows).



MEMORY ALLOCATION

Contiguous Memory Allocation

One approach to memory management is to load each process into a contiguous space. The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed. (The OS is usually loaded low, because that is where the interrupt vectors are located, but on older systems part of the OS was loaded high to make more room in low memory (within the 640K barrier) for user processes.)

Memory Protection (was Memory Mapping and Protection)

- The system shown in Figure 8.6 below allows protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change.

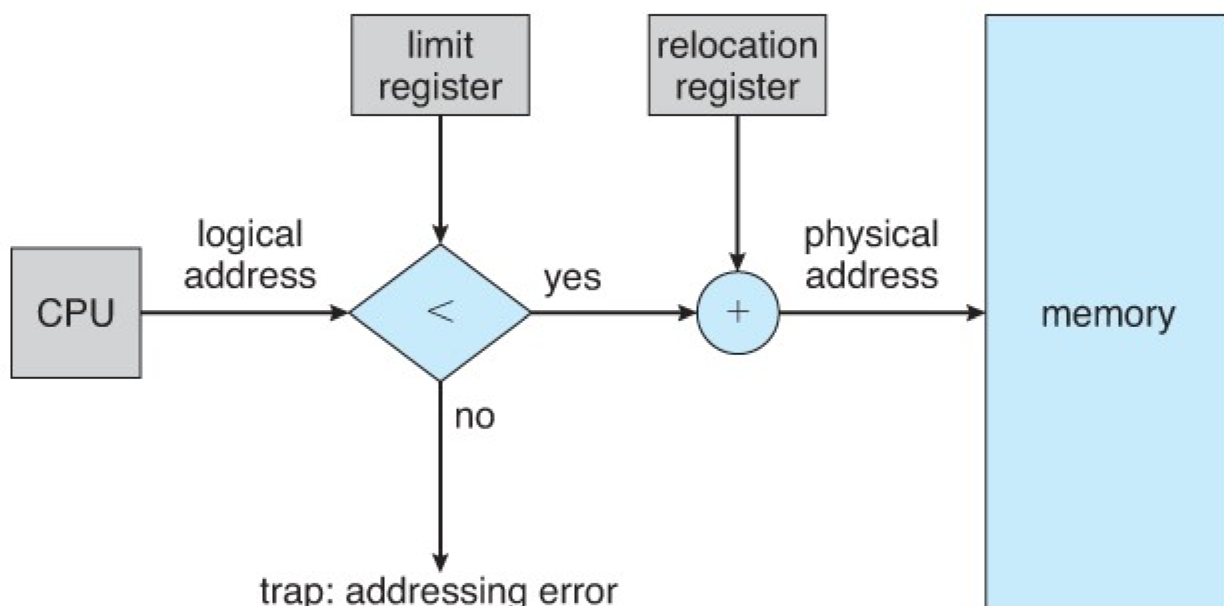


Figure 8.6 - Hardware support for relocation and limit registers

8.3.2 Memory Allocation

- One method of allocating contiguous memory is to divide all available memory into equal sized partitions, and to assign each process to their own partition. This restricts both the number of simultaneous processes and the maximum size of each process, and is no longer used.
- An alternate approach is to keep a list of unused (free) memory blocks (holes), and to find a hole of a suitable size whenever a process needs to be loaded into memory. There

are many different strategies for finding the "best" allocation of memory to processes, including the three most commonly discussed:

1. **First fit** - Search the list of holes until one is found that is big enough to satisfy the request, and assign a portion of that hole to that process. Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole. Subsequent requests may start looking either from the beginning of the list or from the point at which this search ended.
2. **Best fit** - Allocate the *smallest* hole that is big enough to satisfy the request. This saves large holes for other process requests that may need them later, but the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted. Keeping the free list sorted can speed up the process of finding the right hole.
3. **Worst fit** - Allocate the largest hole available, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests.

Fragmentation

Sometimes it happens that memory blocks cannot be allocated to processes due to their small size and memory blocks remain unused. This problem is known as fragmentation.

All the memory allocation strategies suffer from **external fragmentation**, and **internal fragmentation**.

- External fragmentation: it exists when adequate total memory space exists to satisfy a request, but it is not contiguous; storage is fragmented into a large number of holes.
- Internal fragmentation: An approach is to allocate very small holes as part of the larger request. Thus the allocated memory may be larger than the requested memory. Difference between these two number is internal fragmentation – the memory that is internal to any partition but is not being used.
- If the programs in memory are relocatable, (using execution-time address binding), then the external fragmentation problem can be reduced via **compaction**, i.e. moving all processes down to one end of physical memory. This only involves updating the relocation register for each process, as all internal work is done using logical addresses.
- Another solution as we will see in upcoming sections is to allow processes to use non-contiguous blocks of physical memory, with a separate relocation register for each block.

Fixed (or static) Partitioning in Operating System

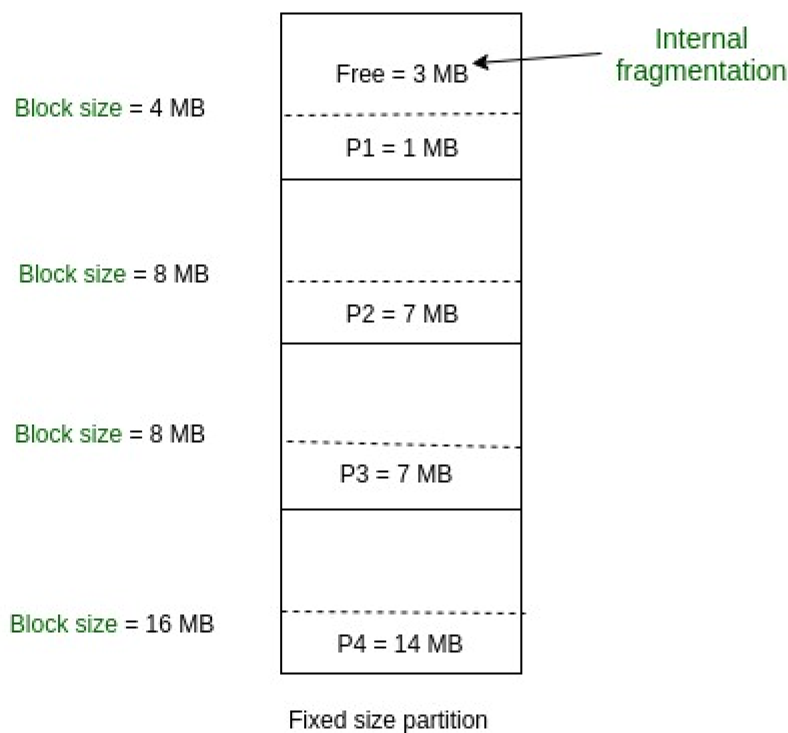
In operating systems, Memory Management is the function responsible for allocating and managing computer's main memory. Memory Management function keeps track of the status of each memory location, either allocated or free to ensure effective and efficient use of Primary Memory.

There are two Memory Management Techniques: **Contiguous**, and **Non-Contiguous**. In Contiguous Technique, executing process must be loaded entirely in main-memory. Contiguous Technique can be divided into:

1. Fixed (or static) partitioning
2. Variable (or dynamic) partitioning

Fixed Partitioning:

This is the oldest and simplest technique used to put more than one processes in the main memory. In this partitioning, number of partitions (non-overlapping) in RAM are **fixed but size** of each partition may or **may not be same**. As it is **contiguous** allocation, hence no spanning is allowed. Here partition are made before execution or during system configure.



Fixed size partition

As illustrated in above figure, first process is only consuming 1MB out of 4MB in the main memory.
Hence, Internal Fragmentation in first block is (4-1) = 3MB.
Sum of Internal Fragmentation in every block = (4-1)+(8-7)+(8-7)+(16-14)= 3+1+1+2 = 7MB.

Suppose process P5 of size 7MB comes. But this process cannot be accommodated inspite of available free space because of contiguous allocation (as spanning is not allowed). Hence, 7MB becomes part of External Fragmentation.

There are some advantages and disadvantages of fixed partitioning.

Advantages of Fixed Partitioning –

1. **Easy to implement:**
   Algorithms needed to implement Fixed Partitioning are easy to implement. It simply

requires putting a process into certain partition without focussing on the emergence of Internal and External Fragmentation.

2. **Little OS overhead:**
   Processing of Fixed Partitioning require lesser excess and indirect computational power.

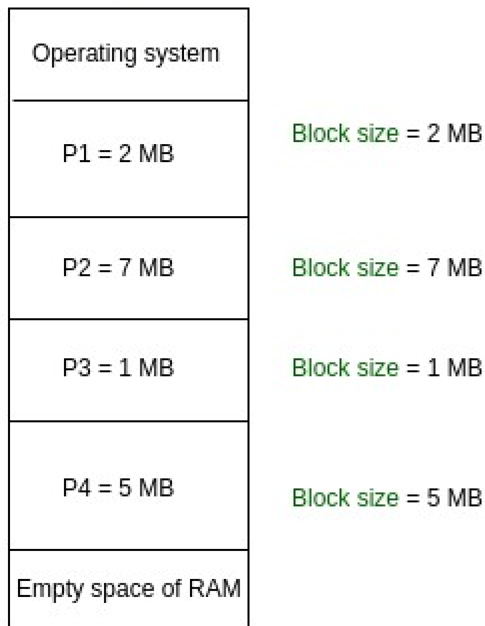
Disadvantages of Fixed Partitioning –

1. **Internal Fragmentation:**
   Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This can cause internal fragmentation.
2. **External Fragmentation:**
   The total unused space (as stated above) of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form (as spanning is not allowed).
3. **Limit process size:**
   Process of size greater than size of partition in Main Memory cannot be accommodated. Partition size cannot be varied according to the size of incoming process's size. Hence, process size of 32MB in above stated example is invalid.
4. **Limitation on Degree of Multiprogramming:**
   Partition in Main Memory are made before execution or during system configure. Main Memory is divided into fixed number of partition. Suppose if there are partitions in RAM and are the number of processes, then condition must be fulfilled. Number of processes greater than number of partitions in RAM is invalid in Fixed Partitioning.

Variable Partitioning –

It is a part of Contiguous allocation technique. It is used to alleviate the problem faced by Fixed Partitioning. In contrast with fixed partitioning, partitions are not made before the execution or during system configure. Various **features** associated with variable Partitioning-

1. Initially RAM is empty and partitions are made during the run-time according to process's need instead of partitioning during system configure.
2. The size of partition will be equal to incoming process.
3. The partition size varies according to the need of the process so that the internal fragmentation can be avoided to ensure efficient utilisation of RAM.
4. Number of partitions in RAM is not fixed and depends on the number of incoming process and Main Memory's size.



There are some advantages and disadvantages of variable partitioning over fixed partitioning as given below.

Advantages of Variable Partitioning –

1. **No Internal Fragmentation:**
   In variable Partitioning, space in main memory is allocated strictly according to the need of process, hence there is no case of internal fragmentation. There will be no unused space left in the partition.
2. **No restriction on Degree of Multiprogramming:**
   More number of processes can be accommodated due to absence of internal fragmentation. A process can be loaded until the memory is empty.
3. **No Limitation on the size of the process:**
   In Fixed partitioning, the process with the size greater than the size of the largest partition could not be loaded and process can not be divided as it is invalid in contiguous allocation technique. Here, In variable partitioning, the process size can't be restricted since the partition size is decided according to the process size.

Disadvantages of Variable Partitioning –

1. **Difficult Implementation:**
   Implementing variable Partitioning is difficult as compared to Fixed Partitioning as it involves allocation of memory during run-time rather than during system configure.
2. **External Fragmentation:**
   There will be external fragmentation inspite of absence of internal fragmentation.
   For example, suppose in above example- process P1(2MB) and process P3(1MB) completed their execution. Hence two spaces are left i.e. 2MB and 1MB. Let's suppose process P5 of size 3MB comes. The empty space in memory cannot be allocated as no

spanning is allowed in contiguous allocation. The rule says that process must be contiguously present in main memory to get executed. Hence it results in External Fragmentation.

Dynamic partitioning

Operating system

P1 (2 MB) executed, now empty — Block size = 2 MB

P2 = 7 MB — Block size = 7 MB

P3 (1 MB) executed — Block size = 1 MB

P4 = 5 MB — Block size = 5 MB

Empty space of RAM

Partition size = process size
So, no internal Fragmentation

Now P5 of size 3 MB cannot be accommodated in spite of required available space because in contiguous no spanning is allowed.

Paging

Paging is a memory management scheme that allows processes physical memory to be discontinuous, and which eliminates problems with fragmentation by allocating memory in equal sized blocks known as ***pages***.

Basic Method

- The basic idea behind paging is to divide physical memory into a number of equal sized blocks called ***frames***, and to divide a programs logical memory space into blocks of the same size called ***pages.***
- Any page (from any process) can be placed into any available frame.
- The ***page table*** is used to look up what frame a particular page is stored in at the moment. In the following example, for instance, page 2 of the program's logical memory is currently stored in frame 3 of physical memory:

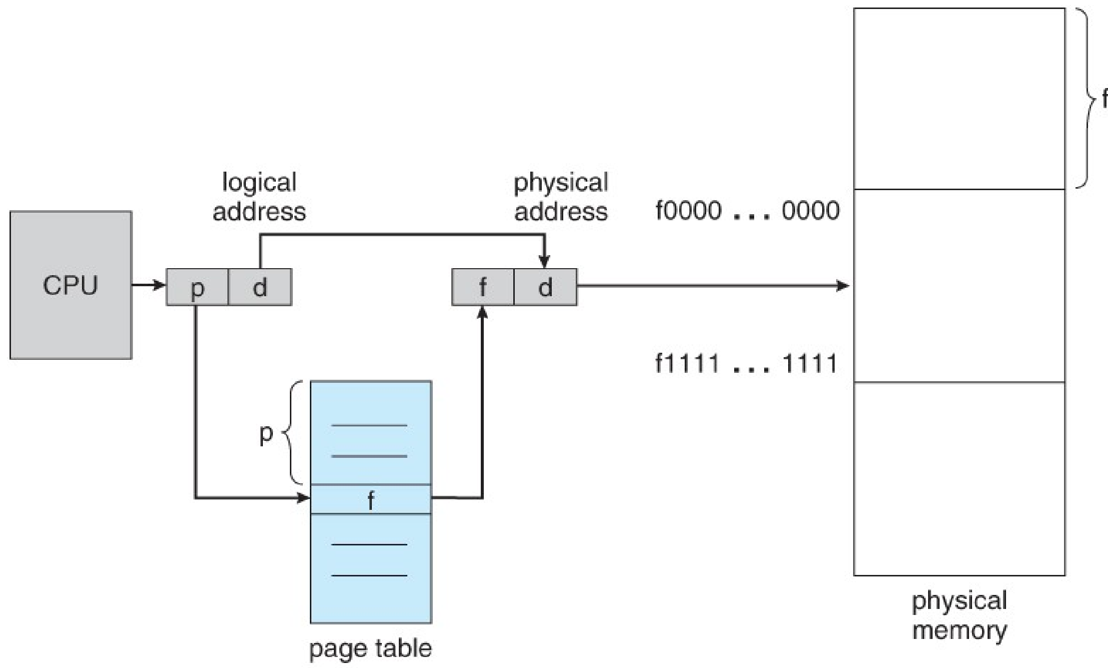


Figure - Paging hardware

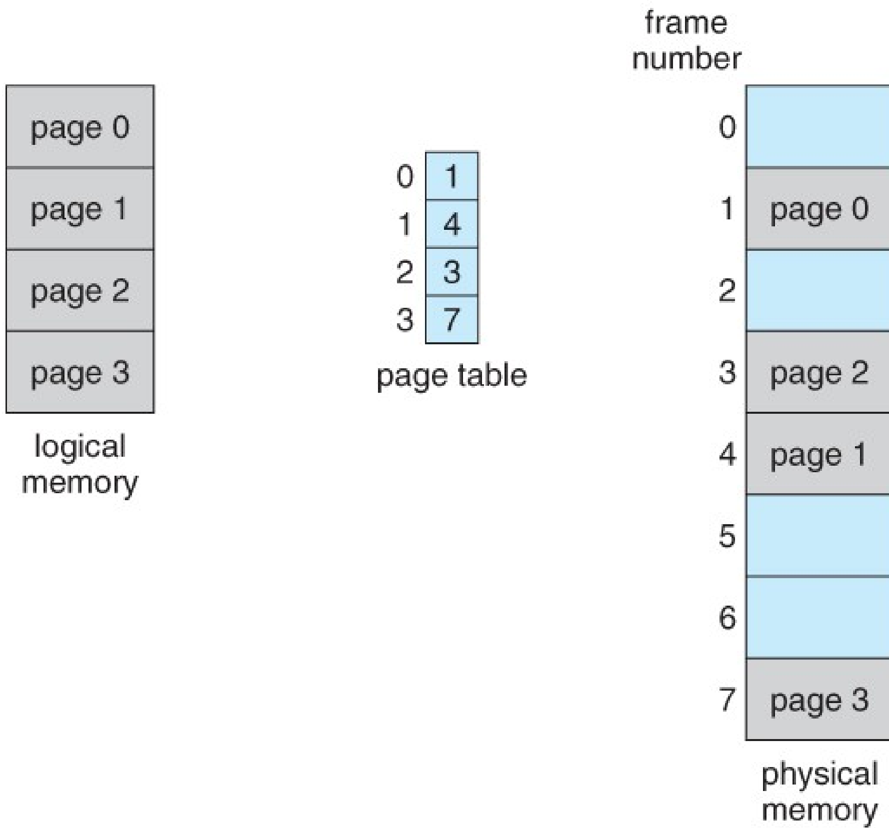
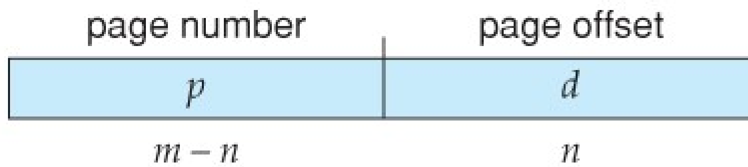


Figure - Paging model of logical and physical memory

- A logical address consists of two parts: A page number in which the address resides, and an offset from the beginning of that page. (The number of bits in the page number limits how many pages a single process can address. The number of bits in the offset determines the maximum size of each page, and should correspond to the system frame size.)
- The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame. The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.

- Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a certain number of bits. For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.

- Note also that the number of bits in the page number and the number of bits in the frame number do not have to be identical. The former determines the address range of the logical address space, and the latter relates to the physical address space.



- Consider the following micro example, in which a process has 16 bytes of logical memory, mapped in 4 byte pages into 32 bytes of physical memory. (Presumably some other processes would be consuming the remaining 16 bytes of physical memory.)

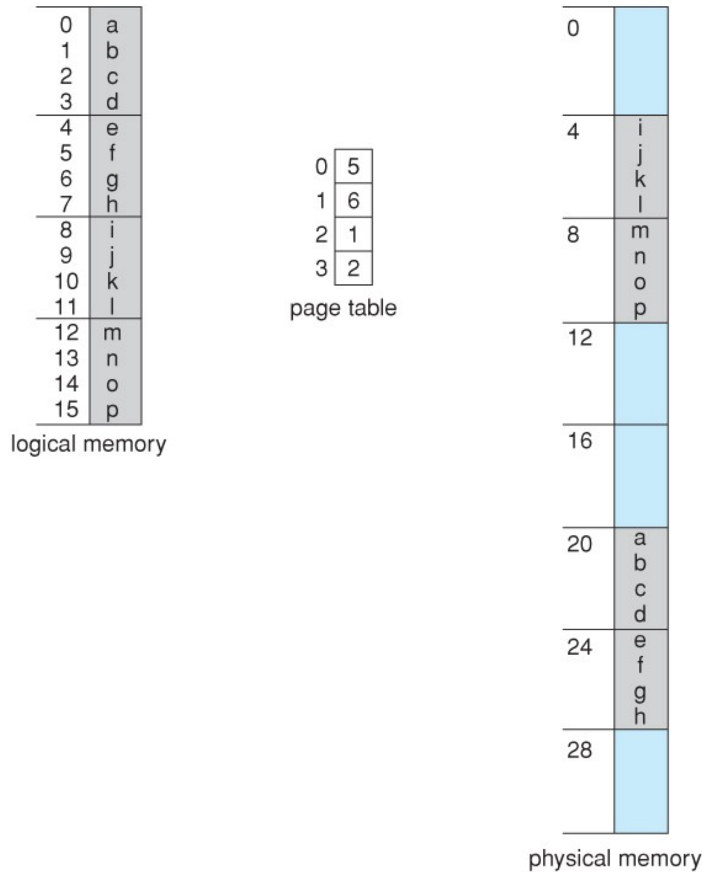


Figure - Free frames (a) before allocation and (b) after allocation

Hardware Support

- Page lookups must be done for every memory reference, and whenever a process gets swapped in or out of the CPU, its page table must be swapped in and out too, along with the instruction registers, etc. It is therefore appropriate to provide hardware support for this operation, in order to make it as fast as possible and to make process switches as fast as possible also.
- One option is to use a set of registers for the page table.
- An alternate option is to store the page table in main memory, and to use a single register (called the page-table base register, PTBR) to record where in memory the page table is located.
    - Process switching is fast, because only the single register needs to be changed.
    - However memory access just got half as fast, because every memory access now requires ***two*** memory accesses - One to fetch the frame number from memory and then another one to access the desired memory location.

- The solution to this problem is to use a very special high-speed memory device called the ***translation look-aside buffer, TLB.***
  - The benefit of the TLB is that it can search an entire table for a key value in parallel, and if it is found anywhere in the table, then the corresponding lookup value is returned.

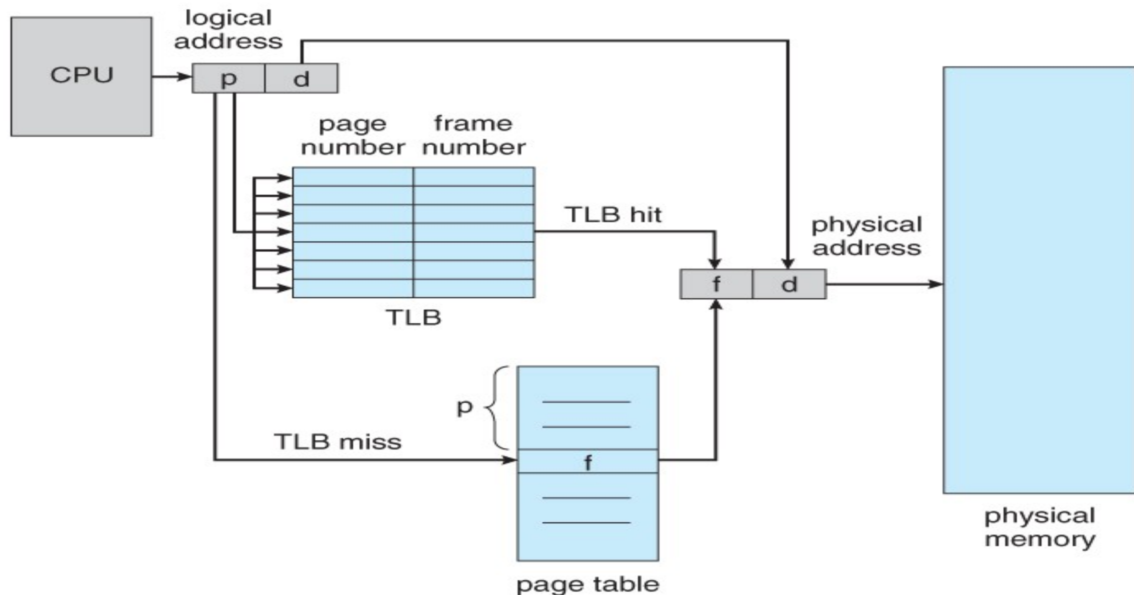


Figure - Paging hardware with TLB

- The TLB is very expensive, however, and therefore very small. (Not large enough to hold the entire page table.) It is therefore used as a cache device.
  - Addresses are first checked against the TLB, and if the info is not there (a TLB miss), then the frame is looked up from main memory and the TLB is updated.
  - If the TLB is full, then replacement strategies range from ***least-recently used, LRU*** to random.
  - Some TLBs allow some entries to be ***wired down***, which means that they cannot be removed from the TLB. Typically these would be kernel frames.
  - Some TLBs store ***address-space identifiers, ASIDs***, to keep track of which process "owns" a particular entry in the TLB. This allows entries from multiple processes to be stored simultaneously in the TLB without granting one process access to some other process's memory location. Without this feature the TLB has to be flushed clean with every process switch.
- The percentage of time that the desired information is found in the TLB is termed the ***hit ratio***.

- For example, suppose that it takes 100 nanoseconds to access main memory, and only 20 nanoseconds to search the TLB. So a TLB hit takes 120 nanoseconds total (20 to find the frame number and then another 100 to go get the data), and a TLB miss takes 220 (20 to search the TLB, 100 to go get the frame number, and then another 100 to go get the data.) So with an 80% TLB hit ratio, the average memory access time would be:

0.80 * 120 + 0.20 * 220 = 140 nanoseconds

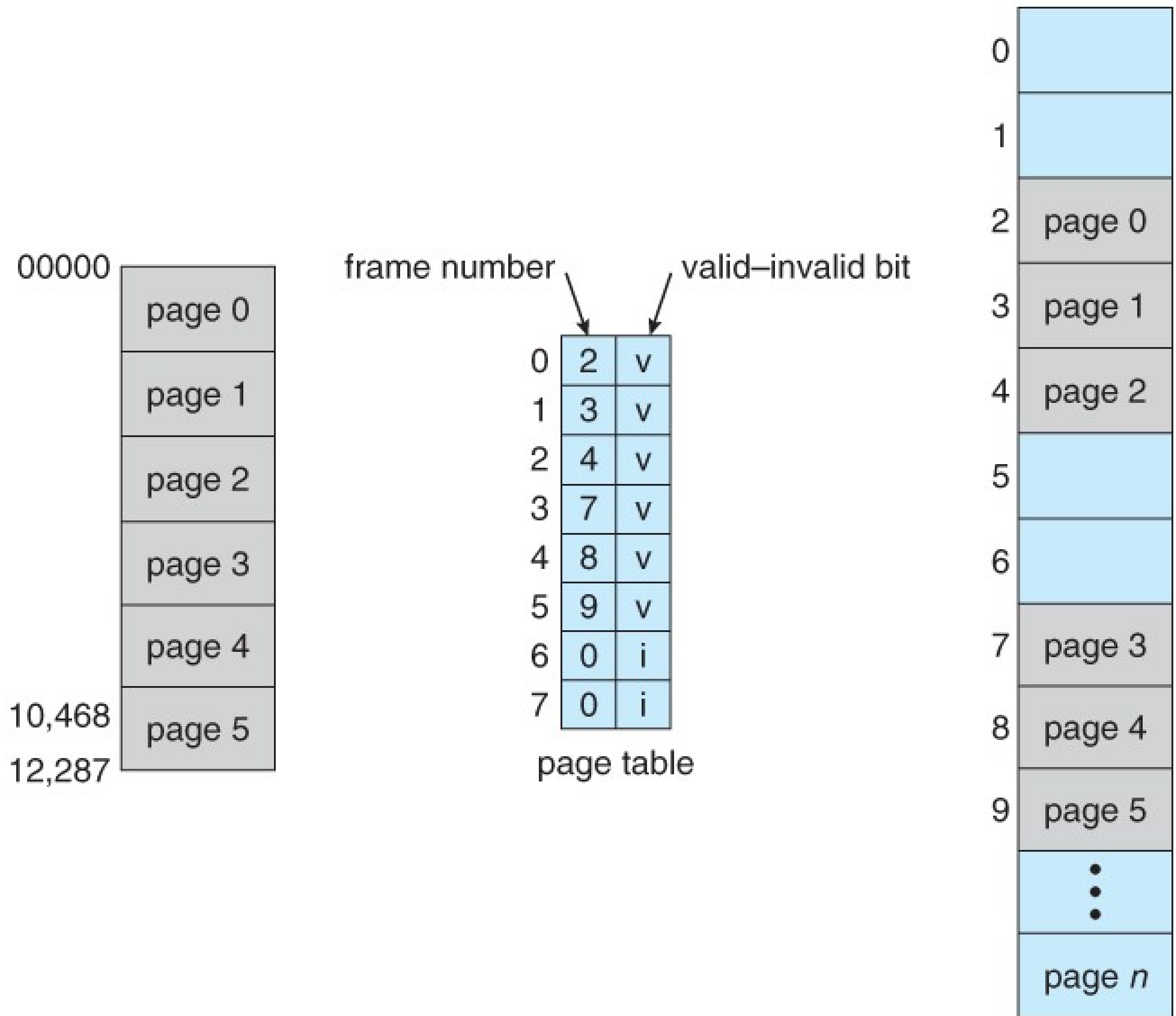
for a 40% slowdown to get the frame number. A 98% hit rate would yield 122 nanoseconds average access time (you should verify this), for a 22% slowdown.

Protection

The page table can also help to protect processes from accessing memory that they shouldn't, or their own memory in ways that they shouldn't.

A bit or bits can be added to the page table to classify a page as read-write, read-only, read-write-execute, or some combination of these sorts of things. Then each memory reference can be checked to ensure it is accessing the memory in the appropriate mode.

Valid / invalid bits can be added to "mask off" entries in the page table that are not in use by the current process, as shown by example in Figure below.



- Many processes do not use all of the page table available to them, particularly in modern systems with very large potential page tables. Rather than waste memory by creating a full-size page table for every process, some systems use a ***page-table length register, PTLR***, to specify the length of the page table.

Address generated by CPU is divided into

Page number(p): Number of bits required to represent the pages in Logical Address Space or Page number
Page offset(d): Number of bits required to represent particular word in a page or page size of Logical Address Space or word number of a page or page offset.

Physical Address is divided into

Frame number(f): Number of bits required to represent the frame of Physical Address Space or Frame number.
Frame offset(d): Number of bits required to represent particular word in a frame or frame size of Physical Address Space or word number of a frame or frame offset.

Virtual Memory in Operating System

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

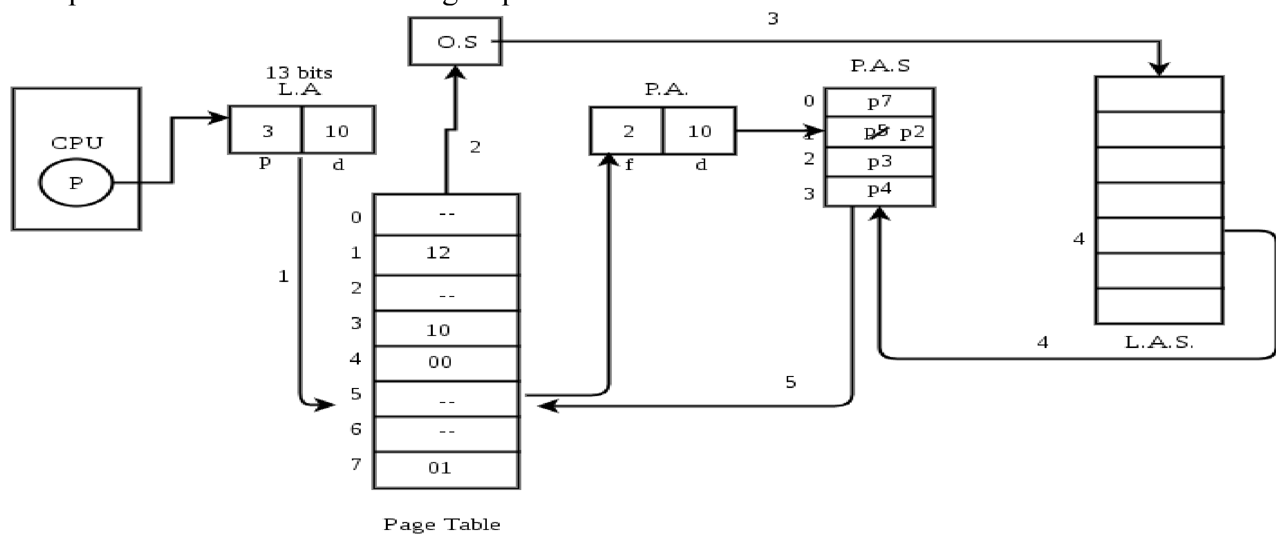
It is a technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory.

1. All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process can be swapped in and out of main memory such that it occupies different places in main memory at different times during the course of execution.
2. A process may be broken into number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and use of page or segment table permits this.

If these characteristics are present then, it is not necessary that all the pages or segments are present in the main memory during execution. This means that the required pages need to be loaded into memory whenever required. Virtual memory is implemented using Demand Paging or Demand Segmentation.

Demand Paging :
The process of loading the page into memory on demand (whenever page fault occurs) is known as demand paging.

The process includes the following steps :



1. If CPU try to refer a page that is currently not available in the main memory, it generates an interrupt indicating memory access fault.
2. The OS puts the interrupted process in a blocking state. For the execution to proceed the OS must bring the required page into the memory.
3. The OS will search for the required page in the logical address space.
4. The required page will be brought from logical address space to physical address space. The page replacement algorithms are used for the decision making of replacing the page in physical address space.
5. The page table will updated accordingly.
6. The signal will be sent to the CPU to continue the program execution and it will place the process back into ready state.

Hence whenever a page fault occurs these steps are followed by the operating system and the required page is brought into memory.

Advantages :

- More processes may be maintained in the main memory: Because we are going to load only some of the pages of any particular process, there is room for more processes. This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in the ready state at any particular time.
- A process may be larger than all of main memory: One of the most fundamental restrictions in programming is lifted. A process larger than the main memory can be executed because of demand paging. The OS itself loads pages of a process in main memory as required.
- It allows greater multiprogramming levels by using less of the available (primary) memory for each process.

Page Fault Service Time :

The time taken to service the page fault is called as page fault service time. The page fault service time includes the time taken to perform all the above six steps.

Let Main memory access time is: m

Page fault service time is: s

Page fault rate is : p

Then, Effective memory access time = (p*s) + (1-p)*m

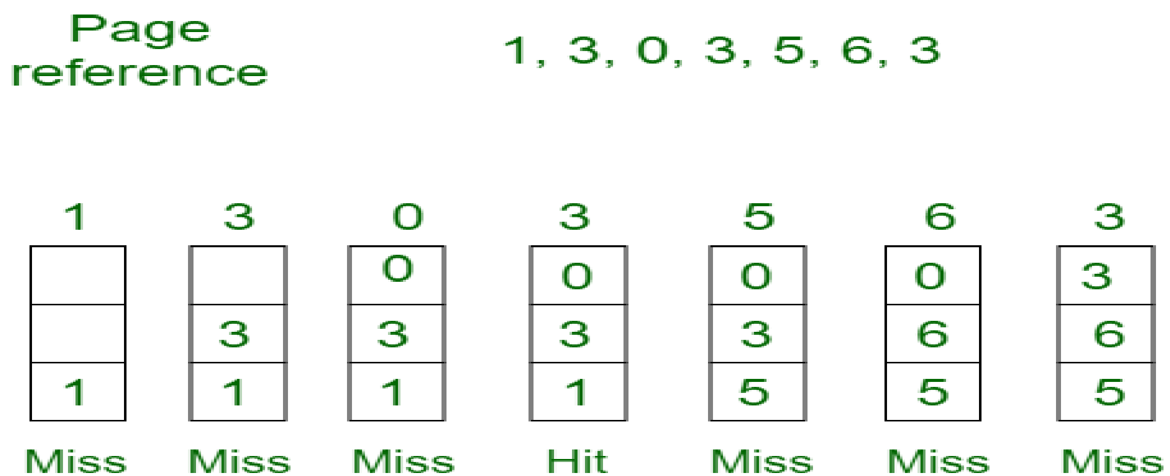
Page Replacement Algorithms in Operating Systems

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

Page Fault – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.
Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

Page Replacement Algorithms :

- **First In First Out (FIFO) –**
  This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.
  Example-1 Consider page reference string 1, 3, 0, 3, 5, 6 with 3 page frames.Find number of page faults.



Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> **3 Page Faults.**
when 3 comes, it is already in memory so —> **0 Page Faults.**
Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>**1 Page Fault.**

6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>**1 Page Fault.**
Finally when 3 come it is not avilable so it replaces 0 **1 page fault**
Belady's anomaly – Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

- **Optimal Page replacement –**
  In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.
  Example-2: Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4 page frame. Find number of page fault.

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4

7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit

Total Page Fault = 6

Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**
0 is already there so —> **0 Page fault.**
when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—>**1 Page fault.**
0 is already there so —> **0 Page fault.**.
4 will takes place of 1 —> **1 Page Fault.**

Now for the further page reference string —> **0 Page fault** because they are already available in the memory.
Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

- **Least Recently Used –**

  In this algorithm page will be replaced which is least recently used.

  Example-3 Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 with 4 page frames.Find number of page faults.

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4

7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit

Total Page Fault = 6

Here LRU has same number of page fault as optimal but it may differ according to question.

Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**

0 is already their so —> **0 Page fault.**

when 3 came it will take the place of 7 because it is least recently used —>**1 Page fault**

0 is already in memory so —> **0 Page fault**.

4 will takes place of 1 —> **1 Page Fault**

Now for the further page reference string —> **0 Page fault** because they are already available in the memory.

UNIT V

Shell and various type of shell

The shell translates your commands and sends them to the system. Every shell has its own features, and some of them are very popular among developers today. These are some of the popular ones:
Sh shell: This is called the Bourne shell, this was developed at AT&T labs in the 70s by a guy named Stephen Bourne. This shell offers many features.
Bash shell: Also called the Bourne again shell, this is very popular and compatible with sh shell scripts, so you can run your sh scripts without changing them.
Ksh shell: Also called the Korn shell, this is compatible with sh and bash. Ksh offers some enhancements over the Bourne shell.
Csh and tcsh: Linux was built using the C language and that drove developers at Berkeley University to develop a C-style shell in which the syntax is similar to the C language. Tcsh adds some minor enhancements to csh.

Various editors present in linux

Text editors can be used for writing code, editing text files such as configuration files, creating user instruction files and many more. In Linux, text editor are of two kinds that is graphical user interface (GUI) and command line text editors (console or terminal).

1. Vi/Vim Editor

Vim is a powerful command line based text editor that has enhanced the functionalities of the old Unix Vi text editor. It is one the most popular and widely used text editors among System Administrators and programmers that is why many users often refer to it as a programmer's editor. It enables syntax highlighting when writing code or editing configuration files.

2. Gedit

This is a general purpose GUI based text editor and is installed by default text editor on Gnome desktop environment. It is simple to use, highly pluggable and a powerful editor with the following features:

1. Support for UTF-8
2. Use of configurable font size and colors
3. Highly customizable syntax highlighting
4. Undo and redo functionalities
5. Reverting of files
6. Remote editing of files
7. Search and replace text
8. Clipboard support functionalities and many more

3. Nano Editor

Nano is an easy to use text editor especially for both new and advanced Linux users. It enhances usability by providing customizable key binding.

Nano has the following features:

1. Highly customizable key bindings
2. Syntax highlighting
3. Undo and redo options
4. Full line display on the standard output
5. Pager support to read form standard input

4. GNU Emacs

This is a highly extensible and customizable text editor that also offers interpretation of the Lisp programming language at its core. Different extensions can be added to support text editing functionalities.

Emacs has the following features:

1. User documentation and tutorials
2. Syntax highlighting using colors even for plain text.
3. Unicode supports many natural languages.
4. Various extension including mail and news, debugger interface, calender and many more

5. Kate/Kwrite

Kate is a feature rich and highly pluggable text editor that comes with KDesktop Environment (KDE). The Kate project aims at development of two main products that is: KatePart and Kate. KatePart is an advanced text editor component included in many KDE applications which may require users to edit text whereas Kate is an multiple document interface(MDI) text editor.
The following are some of its general features:

1. Extensible through scripting
2. Encoding support such as unicode mode
3. Text rendering in bi-directional mode
4. Line ending support with auto detection functionalities

Also remote file editing and many other features including advanced editor features, applications features, programming features, text highlighting features, backup features and search and replace features.

6. Lime Text

This is a powerful IDE-like text editor which is free and open-source successor of popular Sublime Text. It has a few frontends such as command-line interface that you can use with the pluggable backend.

7. Pico Editor

Pico is also a command line based text editor that comes with the Pine news and email client. It is a good editor for new Linux users because of its simplicity in relation to many GUI text editors.

8. Jed Editor

This is also another command line editor with support for GUI like features such as dropdown menus. It is developed purposely for software development and one of its important features is support of unicode mode.

9. gVim Editor

It is a GUI version of the popular Vim editor and it has similar functionalities as the command line Vim.

10. Geany Editor

Geany offers basic IDE-like features with a focus on software development using the GTK+ toolkit.

It has some basic features as listed below:

1. Syntax highlighting
2. Pluggable interface
3. Supports many file types
4. Enables code folding and code navigation
5. Symbol name and construct auto-completion
6. Supports auto-closing of HTML and XML tags
7. Elementary project management functionality plus many more

Different modes of operation in vi editor

The VI editor is the most popular and classic text editor in the Linux family. Below, are some reasons which make it a widely used editor –

- It is available in almost all Linux Distributions
- It works the same across different platforms and Distributions
- It is user-friendly. Hence, millions of Linux users love it and use it for their editing needs

Nowadays, there are advanced versions of the vi editor available, and the most popular one is **VIM** which is **V**i **Im**proved. Some of the other ones are Elvis, Nvi, Nano, and Vile. It is wise to learn vi because it is feature-rich and offers endless possibilities to edit a file.

To work on VI editor, you need to understand **its operation modes**. They can be divided into two main parts.

Command mode:

- The vi editor opens in this mode, and it only **understands commands**
- In this mode, you can, **move the cursor and cut, copy, paste the text**

- This mode also saves the changes you have made to the file
- **Commands are case sensitive.** You should use the right letter case.

Insert mode:

- This mode is for inserting text in the file.
- You can switch to the Insert mode from the command mode **by pressing 'i' on the keyboard**
- Once you are in Insert mode, any key would be taken as an input for the file on which you are currently working.
- To return to the command mode and save the changes you have made you need to press the Esc key

Starting the vi editor

- To launch the VI Editor -Open the Terminal (CLI) and type
- vi <filename_NEW> or <filename_EXISTING>
- &If you specify an existing file, then the editor would open it for you to edit. Else, you can create a new file.

vi Editing commands

Keystrokes	**Action**
i	Insert at cursor **(goes into insert mode)**
a	Write after cursor **(goes into insert mode)**
A	Write at the end of line **(goes into insert mode)**
ESC	Terminate insert mode
u	Undo last change
U	Undo all changes to the entire line
o	Open a new line **(goes into insert mode)**

dd 3dd	Delete line Delete 3 lines.
D	Delete contents of line after the cursor
C	Delete contents of a line after the cursor and insert new text. Press ESC key to end insertion.
dw 4dw	Delete word Delete 4 words
cw	Change word
x	Delete character at the cursor
r	Replace character
R	Overwrite characters from cursor onward
s	Substitute one character under cursor continue to insert
S	Substitute entire line and begin to insert at the beginning of the line
~	Change case of individual character

Moving within a file

Keystroke	**Use**
k	Move cursor up
j	Move cursor down
h	Move cursor left
l	Move cursor right

Saving and Closing the file

Keystroke	Use
Shift+zz	Save the file and quit
:w	Save the file but keep it open
:q	Quit without saving
:wq	Save the file and quit

Shell script

A Shell script can be defined as - "*a series of command(s) stored in a plain text file*". A shell script is similar to a batch file in MS-DOS, but it is much more powerful compared to a batch file. Shell scripts are a fundamental part of the UNIX and Linux programming environment.

Each shell script consists of
• **Shell keywords** such as if..else, do..while.
• **Shell commands** such as pwd, test, echo, continue, type.
• **Linux binary commands** such as w, who, free etc..
• **Text processing utilities** such as grep, awk, cut.
• **Functions** - add frequent actions together via functions. For example, /etc/init.d/functions file contains functions to be used by most or all system shell scripts in the /etc/init.d directory.
• **Control flow** statments such as if..then..else or shell loops to preform repeated actions.

Each script has purpose
• **Specific purpose** - For example, backup file system and database to NAS server.
• **Act like a command** - Each shell script is executed like any other command under Linux.
• **Script code usability** - Shell scripts can be extended from existing scripts. Also, you can use functions files to package frequently used tasks.

Why shell scripting?
• Shell scripts can take input from a user or file and output them to the screen.
• Whenever you find yourself doing the same task over and over again you should use shell scripting, i.e., repetitive task automation.
• Creating your own power tools/utilities.
• Automating command input or entry.
• Customizing administrative tasks.
• Creating simple applications.
• Since scripts are well tested, the chances of errors are reduced while configuring services or system administration tasks such as adding new users.

Advantages
• Easy to use.

• Quick start, and interactive debugging.
• Time Saving.
• Sys Admin task automation.
• Shell scripts can execute without any additional effort on nearly any modern UNIX / Linux / BSD / Mac OS X operating system as they are written an interpreted language.

Disadvantages
• Compatibility problems between different platforms.
• Slow execution speed.
• A new process launched for almost every shell command executed.

Steps to write and execute a script

- Open the terminal. Go to the directory where you want to create your script.
- Create a file with **.sh** extension.
- Write the script in the file using an editor.
- Make the script executable with command **chmod +x** <**fileName**>.
- Run the script using ./<**fileName**>.

Shell variable
variables are used to store data and configuration options. There are two types of variable as follows:
System Variables
Created and maintained by Linux bash shell itself. This type of variable (with the exception of auto_resume and histchars) is defined in CAPITAL LETTERS. You can configure aspects of the shell by modifying system variables such as PS1, PATH, LANG,HISTSIZE,and DISPLAY etc.

Commonly Used Shell Variables
The following variables are set by the shell:

System Variable	Meaning	To View Variable Value Type
BASH_VERSION	Holds the version of this instance of bash.	echo $BASH_VERSION
HOSTNAME	The name of the your computer.	echo $HOSTNAME
CDPATH	The search path for the cd command.	echo $CDPATH
HISTFILE	The name of the file in which command history is saved.	echo $HISTFILE
HISTFILESIZE	The maximum number of lines contained in the history file.	echo $HISTFILESIZE
HISTSIZE	The number of commands to remember in the command history. The default value is 500.	echo $HISTSIZE
HOME	The home directory of the current user.	echo $HOME
IFS	The Internal Field Separator that is used for word splitting after expansion and to split lines into words with the read builtin command. The default value is <space><tab><newline>.	echo $IFS
LANG	Used to determine the locale category for any category not specifically selected with a variable starting with LC_.	echo $LANG
PATH	The search path for commands. It is a colon-separated list of directories in which the shell looks for commands.	echo $PATH
PS1	Your prompt settings.	echo $PS1

User Defined Variables
Created and maintained by user. This type of variable defined may use any valid variable name, but it is good practice to avoid all uppercase names as many are used by the shell.

Creating and setting variables within a script is fairly simple. Use the following syntax:
varName=someValue
someValue is assigned to given **varName** and **someValue** must be on right side of = (equal) sign. If **someValue** is not given, the variable is assigned the null string.

You can display the value of a variable with **echo $varName** or **echo ${varName}**:
echo "$varName"
OR
echo "${varName}"
OR
printf "${varName}"
OR
printf "%s\n" **${**varName**}**
For example, create a variable called vech, and give it a value 'Bus', type the following at a shell prompt:
vech=Bus
Display the value of a variable vech with echo command:
echo "$vech"

OR
echo "${vech}"

System Calls

In computing, a **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to **interact with the operating system**. A computer program makes a system call when it makes a request to the operating system's kernel. System call **provides** the services of the operating system to the user programs via Application Program Interface(API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

Services Provided by System Calls :

1. Process creation and management
2. Main memory management
3. File Access, Directory and File system management
4. Device handling(I/O)
5. Protection
6. Networking, etc.

Types of System Calls : There are 5 different categories of system calls –

1. **Process control:** end, abort, create, terminate, allocate and free memory.
2. **File management:** create, open, close, delete, read file etc.
3. Device management
4. Information maintenance
5. Communication

Examples of Unix System Calls –

	UNIX
Process Control	fork() exit() wait()
File Manipulation	open() read() write() close()
Device Manipulation	ioctl() read() write()
Information Maintenance	getpid()

	alarm() sleep()
Communication	pipe() shmget() mmap()
Protection	chmod() umask() chown()

Pipes and Filters

A ***pipe*** is a means by which the output from one process becomes the input to a second. In technical terms, the standard output (stout) of one command is sent to the standard input (stdin) of a second command.

To make a pipe, put a vertical bar (|) on the command line between two commands.
When a program takes its input from another program, it performs some operation on that input, and writes the result to the standard output. It is referred to as a ***filter***.

The grep Command

The grep command searches a file or files for lines that have a certain pattern. The syntax is −

$grep pattern file(s)

The name **"grep"** comes from the ed (a Unix line editor) command **g/re/p** which means "globally search for a regular expression and print all lines containing it".

A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching.

The simplest use of grep is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output.

```
$ls -l | grep "Aug"
-rw-rw-rw-   1 john  doc     11008 Aug  6 14:10 ch02
-rw-rw-rw-   1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--   1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
$
```

There are various options which you can use along with the **grep** command −

Sr.No.	Option & Description
1	**-v** Prints all lines that do not match pattern.
2	**-n** Prints the matched line and its line number.
3	**-l** Prints only the names of files with matching lines (letter "l")
4	**-c** Prints only the count of matching lines.
5	**-i** Matches either upper or lowercase.

The sort Command

The **sort** command arranges lines of text alphabetically or numerically. The following example sorts the lines in the food file −

```
$sort food
Afghani Cuisine
Bangkok Wok
Big Apple Deli
Isle of Java

Mandalay
Sushi and Sashimi
Sweet Tooth
Tio Pepe's Peppers
$
```

The **sort** command arranges lines of text alphabetically by default. There are many options that control the sorting −

Sr.No.	Description

1	**-n** Sorts numerically (example: 10 will sort after 2), ignores blanks and tabs.
2	**-r** Reverses the order of sort.
3	**-f** Sorts upper and lowercase together.
4	**+x** Ignores first **x** fields when sorting.

More than two commands may be linked up into a pipe. Taking a previous pipe example using **grep**, we can further sort the files modified in August by the order of size.

The following pipe consists of the commands **ls**, **grep**, and **sort** −

```
$ls -l | grep "Aug" | sort +4n
-rw-rw-r--  1 carol doc      1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-rw-  1 john  doc     11008 Aug  6 14:10 ch02
$
```

This pipe sorts all files in your directory modified in August by the order of size, and prints them on the terminal screen. The sort option +4n skips four fields (fields are separated by blanks) then sorts the lines in numeric order.

Decision making in Shell Scripts

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. Two decision-making statements are :

- The **if...else** statement
- The **case...esac** statement

The if...else statements

If else statements are useful decision-making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of **if…else** statement −

- if...fi statement
- if...else...fi statement
- if...elif...else...fi statement

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

Syntax

```
if [ expression ]
then
  Statement(s) to be executed if expression is true
fi
```

The *Shell expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)*are executed. If the *expression* is *false* then no statement would be executed. Most of the times, comparison operators are used for making decisions.

It is recommended to be careful with the spaces between braces and expression. No space produces a syntax error.

If **expression** is a shell command, then it will be assumed true if it returns **0** after execution. If it is a Boolean expression, then it would be true if it returns true.

Example

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
  echo "a is equal to b"
fi

if [ $a != $b ]
then
  echo "a is not equal to b"
fi
```

The above script will generate the following result −

```
a is not equal to b
```

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in a controlled way and make the right choice.

Syntax

```
if [ expression ]
```

```
then
   Statement(s) to be executed if expression is true
else
   Statement(s) to be executed if expression is not true
fi
```

The Shell *expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false*, then no statement will be executed.

Example

The above example can also be written using the *if...else* statement as follows −

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
else
   echo "a is not equal to b"
fi
```

Upon execution, you will receive the following result −

```
a is not equal to b
```

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

Syntax

```
if [ expression 1 ]
then
   Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
then
   Statement(s) to be executed if expression 2 is true
elif [ expression 3 ]
then
   Statement(s) to be executed if expression 3 is true
else
   Statement(s) to be executed if no expression is true
fi
```

This code is just a series of *if* statements, where each *if* is part of the *else* clause of the previous statement. Here statement(s) are executed based on the true condition, if none of the condition is true then *else* block is executed.

Example

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "None of the condition met"
fi
```

Upon execution, you will receive the following result −

```
a is less than b
```

The case...esac Statement

Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated **if...elif** statements.

- case...esac statement

The **case...esac** statement in the Unix shell is very similar to the **switch...case** statement we have in other programming languages like **C** or **C++** and **PERL**, etc.

The basic syntax of the **case...esac** statement is to give an expression to evaluate and to execute several different statements based on the value of the expression.

The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

```
case word in
   pattern1)
      Statement(s) to be executed if pattern1 matches
      ;;
   pattern2)
      Statement(s) to be executed if pattern2 matches
      ;;
   pattern3)
      Statement(s) to be executed if pattern3 matches
      ;;
   *)
```

```
   Default condition to be executed
   ;;
esac
```

Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action.

There is no maximum number of patterns, but the minimum is one.

When statement(s) part executes, the command ;; indicates that the program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

Example

```
#!/bin/sh

FRUIT="kiwi"

case "$FRUIT" in
  "apple") echo "Apple pie is quite tasty."
  ;;
  "banana") echo "I like banana nut bread."
  ;;
  "kiwi") echo "New Zealand is famous for kiwi."
  ;;
esac
```

Upon execution, you will receive the following result −

New Zealand is famous for kiwi.

Loops in shell Functions

A loop is a powerful programming tool that enables you to execute a set of commands repeatedly. The following types of loops available to shell programmers −

- The while loop
- The for loop
- The until loop

The **while** loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

Syntax

```
while command
do
  Statement(s) to be executed if command is true
done
```

Here the Shell *command* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *command* is *false* then no statement will be executed and the program will jump to the next line after the done statement.

Example

Here is a simple example that uses the **while** loop to display the numbers zero to nine −

```
#!/bin/sh

a=0

while [ $a -lt 10 ]
do
  echo $a
  a=`expr $a + 1`
done
```

Upon execution, you will receive the following result −

```
0
1
2
3
4
5
6
7
8
9
```

The **for** loop operates on lists of items. It repeats a set of commands for every item in a list.

Syntax

```
for var in word1 word2 ... wordN
do
  Statement(s) to be executed for every word.
done
```

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Example

Here is a simple example that uses the **for** loop to span through the given list of numbers −

```
#!/bin/sh

for var in 0 1 2 3 4 5 6 7 8 9
do
  echo $var
```

```
done
```

Upon execution, you will receive the following result −

```
0
1
2
3
4
5
6
7
8
9
```

UNTIL LOOP

The while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.

Syntax

```
until command
do
   Statement(s) to be executed until command is true
done
```

Here the Shell *command* is evaluated. If the resulting value is *false*, given *statement(s)* are executed. If the *command* is *true* then no statement will be executed and the program jumps to the next line after the done statement.

Example

Here is a simple example that uses the until loop to display the numbers zero to nine −

```
#!/bin/sh

a=0

until [ ! $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

Upon execution, you will receive the following result −

```
0
1
2
3
4
```

5
6
7
8
9

UTILITY PROGRAMS

Cut

The cut command in UNIX takes the section from a file and outputs it to standard out. However, it does not delete any part of the file it extracts text from. cut is powerful in that it may accept multiple files for its standard input.

With the options listed, there are several ways you can specify a cut.

-b Select only the bytes specified. May be a single, set or range of bytes, separated by a comma.
-c Specify the number of characters from each line.
-f Extract a set of specified fields.
-d Used with the -f option. Use a specified delimiter rather than default tab.
Each of these options come with a list that is made up of an integer, range of integers, or multiple integer ranges separated by a comma. A list is defined as follows:

N The *n*th byte, character or field. Count starts at 1.
n- From the *n*th byte, character or field forward.
n-m From the *n*th to the *m*th byte, character or field (inclusive).
-m From the first to the *m*th byte.

```
$ cat test.txt
doh  re  me  fa  so
1 2 3 4 5
$ cut -f 3-5 test.txt
me  fa  so
3 4 5
$ cut -f 1,3-4 test.txt
doh  me  fa
1 3 4
```

Paste

The paste command is used to merge lines of files together. With this command, you can add one or more columns (or fields) of text to a file. There are two options you should be aware of:

-d Specify the delimiter to be used instead of tabs.

-s Append in serial instead of parallel. (Horizontal pasting instead of vertical.)

```
$ cat names.txt
Billy
Bob
Chase
Jon
Jonathan
$ cat birthdates.txt
09/21/1992
08/12/1982
05/24/1999
04/23/1974
08/09/2001
$ paste -d ',' names.txt birthdates.txt
Billy,09/21/1992
Bob,08/12/1982
Chase,05/24/1999
Jon,04/23/1974
Jonathan,08/09/2001
```

Join

Join takes a common column between two tables, and joins them together based on that attribute.

-t Specify a delimiter

-1 *n* Use the *n*th column as the join key for the first column.

-2 *n* Use the *n*th column as the join key for the second column.

-a *n* Also print the unprintable lines from *n*, where *n* is 1 or 2 (first or second file).

```
$ cat birthdates.txt
05/24/1999,4
04/23/1974,2
```

```
08/09/2001,5
11/24/1991,3
01/23/1975,1
$ cat names.txt
Billy,1
Bob,2
Chase,3
Jon,4
Jonathan,5
```

First, we must have both lists sorted according to the column we want to join on. names.txt is already sorted, but birthdates.txt is not

```
$ sort -t ',' -k 2 birthdates.txt > sortedBirthdates.txt
$ join -t ',' -1 2 -2 2 names.txt sortedBirthdates.txt
1,Billy,01/23/1975
2,Bob,04/23/1974
3,Chase,11/24/1991
4,Jon,05/24/1999
5,Jonathan,08/09/2001
```

Right/Left outer join

In some cases, you'll want to join two tables even though there are some rows without a corresponding value in the other row. Joining the right table with missing corresponding values is called a *right outer join*, and joining the left table with missing corresponding rows is called a *left outer join*.

A left outer join would have the option `-a1` and a right outer join would have option `-a2`.

Full outer join

In a *full outer join*, both table rows are included, even if they don't have a corresponding row. Expect many null cell values when using this option.

Use the `-a` option for a full outer join.

tr command

The tr command in UNIX is a command line utility for translating or deleting characters. It supports a range of transformations including uppercase to lowercase, squeezing repeating characters, deleting specific characters and basic find and replace. It can be used with UNIX pipes to support more complex translation. **tr stands for translate.**

Syntax :
$ tr [OPTION] SET1 [SET2]

Options
-c : complements the set of characters in string.i.e., operations apply to characters not in the given set
-d : delete characters in the first set from the output.
-s : replaces repeated characters listed in the set1 with single occurrence
-t : truncates set1

1. How to convert lower case to upper case
To convert from lower case to upper case the predefined sets in tr can be used.
$cat greekfile
Output:

WELCOME TO

GeeksforGeeks

$cat greekfile | tr “[a-z]” “[A-Z]”
Output:

WELCOME TO

GEEKSFORGEEKS

2. How to translate white-space to tabs
The following command will translate all the white-space to tabs
$ echo "Welcome To GeeksforGeeks" | tr [:space:] '\t'
Output:

Welcome	To	GeeksforGeeks

3. How to translate braces into parenthesis
You can also translate from and to a file. In this example we will translate braces in a file with parenthesis.
$cat greekfile
Output:

{WELCOME TO}

GeeksforGeeks

$ tr '{}' '()'   newfile.txt
Output:

(WELCOME TO)

GeeksforGeeks

The above command will read each character from “geekfile.txt”, translate if it is a brace, and write the output in “newfile.txt”.

4. How to use squeeze repetition of characters using -s
To squeeze repeat occurrences of characters specified in a set use the -s option. This removes repeated instances of a character.
OR we can say that,you can convert multiple continuous spaces with a single space
$ echo "Welcome   To   GeeksforGeeks" | tr -s [:space:] ' '
Output:

Welcome To GeeksforGeeks

5. How to delete specified characters using -d option
To delete specific characters use the -d option.This option deletes characters in the first set specified.
$ echo "Welcome To GeeksforGeeks" | tr -d 'w'
Output:

elcome To GeeksforGeeks

6. To remove all the digits from the string, use
$ echo "my ID is 73535" | tr -d [:digit:]
Output:

my ID is

7. How to complement the sets using -c option
You can complement the SET1 using -c option. For example, to remove all characters except digits, you can use the following.
$ echo "my ID is 73535" | tr -cd [:digit:]
Output:

73535

Uniq command

Uniq command in unix or linux system is used to suppress the duplicate lines from a file. It discards all the successive identical lines except one from the input and writes the output.

The syntax of uniq command is

uniq [option] filename

The options of uniq command are:

c : Count of occurrence of each line.

d : Prints only duplicate lines.

D : Print all duplicate lines

f : Avoid comparing first N fields.

i : Ignore case when comparing.

s : Avoid comparing first N characters.

u : Prints only unique lines.

w : Compare no more than N characters in lines

Uniq Command Examples:

First create the following example.txt file in your unix or linux operating system.

```
> cat example.txt
Unix operating system
unix operating system
unix dedicated server
linux dedicated server
```

1. Suppress duplicate lines

The default behavior of the uniq command is to suppress the duplicate line. Note that, you have to pass sorted input to the uniq, as it compares only successive lines.

```
> uniq example.txt
unix operating system
unix dedicated server
linux dedicated server
```

If the lines in the file are not in sorted order, then use the sort command and then pipe the output to the uniq command.

```
> sort example.txt | uniq
```

2. Count of lines.

The -c option is used to find how many times each line occurs in the file. It prefixes each line with the count.

```
> uniq -c example.txt
   2 unix operating system
   1 unix dedicated server
```

1 linux dedicated server

3. Display only duplicate lines.

You can print only the lines that occur more than once in a file using the -d option.

```
> uniq -d example.txt
unix operating system
```

```
> uniq -D example.txt
unix operating system
unix operating system
```

The -D option prints all the duplicate lines.

4. Skip first N fields in comparison.

The -f option is used to skip the first N columns in comparison. Here the fields are delimited by the space character.

```
> uniq -f2 example.txt
unix operating system
unix dedicated server
```

In the above example the uniq command, just compares the last fields. For the first two lines, the last field contains the string "system". Uniq prints the first line and skips the second. Similarly it prints the third line and skips the fourth line.

5. Print only unique lines.

You can skip the duplicate lines and print only unique lines using the -u option

```
> uniq -u example.txt
unix dedicated server
linux dedicated server
```

The grep Command

The grep command searches a file or files for lines that have a certain pattern. The syntax is −

$grep pattern file(s)

The name **"grep"** comes from the ed (a Unix line editor) command **g/re/p** which means "globally search for a regular expression and print all lines containing it".

A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching.

The simplest use of grep is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output.

```
$ls -l | grep "Aug"
-rw-rw-rw-   1 john  doc     11008 Aug  6 14:10 ch02
-rw-rw-rw-   1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--   1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
$
```

There are various options which you can use along with the **grep** command −

Sr.No.	Option & Description
1	**-v** Prints all lines that do not match pattern.
2	**-n** Prints the matched line and its line number.
3	**-l** Prints only the names of files with matching lines (letter "l")
4	**-c** Prints only the count of matching lines.
5	**-i** Matches either upper or lowercase.

VINAYAKA MISSIONS RESEARCH FOUNDATION
(Deemed to be University)

DEPARTMENT OF COMPUTER SCIENCE

OPERATING SYSTEM
QUESTION BANK-2019

UNIT -I

Part-A (2 Marks)

1. What is an Operating System?
2. What is a system software?
3. Define resource abstraction.
4. What is a device driver?
5. Define booting.
6. What is device controller?
7. List out the strategies on which modern operating system is built on?
8. List out the types of operating system.
9. What is batch processing?
10. What is time sharing?
11. Expand BIOS and write its function.
12. Define POST.
13. List out any two features of multiprogramming.
14. What is multiprocessing?
15. List the two types of real time system.

Part-B (5 Marks)

1. Discuss in detail about system software.
2. Explain about resource abstraction.
3. Explain in detail about booting process with a neat diagram.
4. Discuss about the single user operating system with a suitable example.
5. Discuss about the concept of batch processing.
6. Discuss about the concept of timesharing with an example.
7. Mention the two basic purposes of operating system.
8. Describe the different types of real time system.

9. List out the advantages of multiprocessor system.
10. List out the advantages of multiprogramming system.

Part-C (10 marks)

1. Explain in detail about the different strategies in building a modern operating system.
2. Explain in detail about the multi programming operating system with neat diagram.
3. Differentiate between a single user and a multi user operating system.
4. Explain in detail about batch and time sharing operating system.
5. Discuss in detail about real time operating system.

UNIT -II

Part-A (2 Marks)

1. List the factors for designing an operating system
2. What is the use of mechanisms in operating system?
3. Define policies in terms of operating system.
4. Define trap.
5. What are the two different modes of operations in operating system?
6. What is the use of mode bit?
7. Define process.
8. Define system process.
9. Define user process.
10. Define file.
11. What is a system call?
12. List any two functions of operating system.
13. Give two examples for tertiary storage devices.
14. Define protection.
15. Mention the use of system program.

Part-B (5 Marks)

1. Discuss in detail about design goal of an operating system.
2. Discuss about mechanisms and policies in designing an operating system.
3. Write the advantages of designing operating system using highlevel languages.
4. List the disadvantages of designing operating system using highlevel languages.
5. Discuss in detail about different process modes in operating system.
6. Explain in detail how system calls are used with a suitable example.
7. Explain in detail about the use of system calls in file management.
8. Explain in detail about role of operating system in memory management.
9. List out the system calls provided by operating system for information maintenance.
10. Explain in detail about role of operating system in I/O system management.

Part-C (10 marks)

1. Explain in detail about the factors to be considered in designing an operating system.
2. Discuss in detail about the functions of operating system.

3. Explain in detail about the use of system calls in process management.
4. Discuss in detail about the use of system calls in device management and communication.
5. Explain in detail about system programs.

UNIT-III
Part-A(2 marks)

1. What is a process?
2. Define process state.
3. What is the use of process control block?
4. What is the use of job queues, ready queues and device queues?
5. What is meant by context switch?
6. Define thread.
7. Define CPU scheduling.
8. What is preemptive and non-preemptive scheduling?
9. What is a dispatcher?
10. What is dispatch latency?
11. What are the various scheduling criteria for CPU scheduling?
12. Define throughput.
13. What is turnaround time?
14. What is meant by cascading termination?
15. Differentiate long term scheduler and short term scheduler.

Part-B (5 marks)

1. With a neat diagram, explain different states of a process.
2. Discuss in detail about process hierarchies.
3. Explain the various scheduling criteria for CPU scheduling algorithms.
4. Explain in detail about CPU Scheduling mechanism.
5. Give an overview about threads.
6. Differentiate process and threads.
7. Explain with an example about Round Robin scheduling algorithm.
8. Discuss in detail about context switching with a neat diagram.
9. Explain with an example about First Come First Serve (FCFS) scheduling algorithm.
10. Consider following processes with length of CPU burst time in milliseconds

Process	**Burst time**
P1	5
P2	10
P3	2
P4	1

All process arrived in order p1, p2, p3, p4 all time zero

a) Draw Gantt charts illustrating execution of these processes for SJF and round robin (quantum=1)
b) Calculate waiting time for each process for each scheduling algorithm
c) Calculate average waiting time for each scheduling algorithm

Part-C (10 marks)

1. Explain process control block in details.
2. Explain different levels of threads with advantages and disadvantages.
3. Explain the following process scheduling algorithm
   a) Priority scheduling
   b) Shortest job first scheduling
4. Explain in detail about process creation and process termination.
5. Consider the following set of processes with their arrival and burst times as shown

Process	**ArrivalTime**	**Burst Time**
P0	0	10HR
P1	0	05HR
P2	1	02HR
P3	2	01HR

Compute the turn around time and waiting time of each job using the following scheduling algorithms. i) FCFS ii) S.J.F iii) Round-Robin (choose time quantum=1)

UNIT-IV

Part-A (2 marks)

1. What is the main function of the memory-management unit?
2. Define dynamic loading.
3. Define dynamic linking.
4. What are overlays?
5. Define swapping.
6. What are the common strategies to select a free hole from a set of available holes?
7. Mention the use of page table.
8. What do you mean by best fit?
9. What is virtual memory?
10. What is Demand paging?
11. What are the various page replacement algorithms used for page replacement?
12. What are the major problems to implement demand paging?
13. What are pages and frames?
14. Define fragmentation.
15. What is page fault?

Part-B (5 marks)

1. Explain the difference between Physical and logical address.
2. Explain in detail about fixed and variable partition of memory allocation.
3. Explain with neat diagram internal and external fragmentation.
4. Explain in detail about hierarchical page table.
5. Discuss in detail about shared page with a suitable example.
6. What is dynamic storage allocation problem? Mention the names of different methods used to solve the above problem.
7. What is page fault and how it is handled?
8. What is thrashing and explain its cause.
9. What is demand paging? Explain it with address translation mechanism used.
10. Discuss LRU-Approximation page Replacement.

Part-C (10 marks)

1. Explain about contiguous memory allocation.
2. Give the basic concepts about paging.
3. Write about the techniques for structuring the page table.
4. Explain any two page replacement algorithms
5. Consider the following page reference string
   1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6
   Find out the number of page faults if there are 4 page frames, using the following page replacement algorithm i) LRU ii) FIFO iii) Optimal

UNIT-V

Part-A (2 marks)

1. What is a shell?
2. Define shell script.
3. List the different types of shell.
4. What are the various types of editors present in linux?
5. What are the different modes of operations in vi editor?
6. Mention the uses of ls –l command.
7. What is the use of system variable?
8. How will you declare a user defined variable?
9. What is a pipeline?
10. What is the use of 'for' in shell script?
11. Write the purpose of cut command.
12. What is the use of the options c and v in grep command.
13. What is the use of test command?
14. differentiate rmdir and rm command.
15. List out the logical operators used in shell script.

Part-B (5 marks)

1. Construct a pipeline for the following jobs:-

   (1) List all files beginning with the char “B” on the screen and also store them in a file called 1.

   (2) All the files present in dir-dir 1 should be deleted. Any error, if it occurs while carrying out this operation, should be stored in a file “error file”.

2. Explain each column of the output of ls –l command.
3. Explain following commands with suitable examples.
   Pwd, wc.
4. Explain the commands search for pattern and search & replace in Vi editor.
5. Write a short note on filters command.
6. Explain the following with examples : cut and paste
7. What is ‘for’ loop in a shell script? Explain the different ways of making the lists.
8. Write the output of the following script and explain it.
   ```
   for var in one "This is two" "Now three" "We'll check four"
   do
   echo "Value: $var"
   done
   ```
9. Write short note on vi editor.
10. Write short note on types of shell in linux.

Part-C (10 marks)

1. What are different modes of vi editor? Explain.
2. Explain the system variables with example.
3. Explain in detail about decision making in shell script with examples.
4. How does grep help in searching for a pattern? Explain each option with an example.
5. Write a shell script to accept a directory name and check for its existing. If exists, display number of directories and ordinary files in it. Otherwise display message as “directory does not exist”.