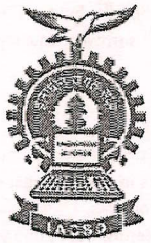
17 //



Institute for Advanced Computing And Software Development (IACSD) Akurdi, Pune

Operating Systems

Dr. D.Y. Patil Educational Complex, Sector 29, Behind Akurdi Railway Station,
Nigdi Pradhikaran, Akurdi, Pune - 411044.

Introduction to OS

Functions of OS

Memory Management

Memory management refers to management of Primary Memory or Main Memory. It's keeping track of all different applications and processes running on your computer and all the data they're using. 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. Operating systems can easily check which bytes of main memory are empty and which are not. It allocates the main memory for the program execution, and when the program is completed or terminated, then it deallocates the memory. Operating systems also keep a record that which byte of memory is assigned to which program.

Processor Management

The operating system's responsibility is to manage the processes running on your computer. This includes starting and stopping programs, allocating resources, and managing memory usage. The operating system manages all the processes so that each process gets the CPU for a specific time to execute itself, and there will be less waiting time for each process.

Device Management

Operating systems provide essential functions for managing devices connected to a computer. These functions include *allocating memory, processing input and output requests, and managing storage devices.* An Operating System manages device communication via their respective drivers.

File Management

Operating systems are responsible for managing the files on a computer. This includes creating, opening, closing, and deleting files.

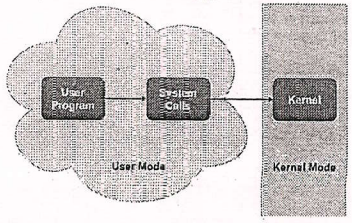
What is Interrupts and System call?

System Interrupt

System interrupts are a way for a process to alert the kernel that an event has occurred. Once interrupted, the kernel can process the event and return to the process where it left off. System interrupts are also used to suspend the execution of a program temporarily.

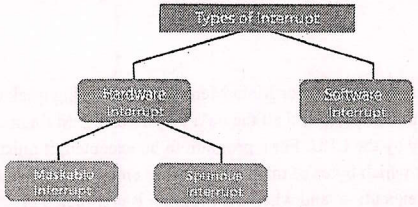
System call

A system call is a method that enables a user process to interact with the kernel of Operating System. It is a call from user mode to kernel mode.



Types of Interrupt

Interrupt signals may be issued in response to hardware and software events. Basically, these are classified as hardware interrupt and software interrupt.



Software Interrupt:

Software interrupts generally take place when *there are exceptions in the process or by using special instructions that cause the interrupts.*

Hardware Interrupt:

When an external device wants the attention of the operating system to service a certain request, they raise an interrupt which is called hardware interrupts.

Maskable Interrupt:

Processors typically have an internal interrupt mask register which allows selective enabling and disabling of hardware interrupts. Each interrupt signal is associated with a bit in the mask register; on some systems, the interrupt is enabled when the bit is set and disabled when the bit is clear, while on others, a set bit disables the interrupt. When the interrupt is disabled, the associated interrupt signal will be ignored by the processor. Signals which are affected by the mask are called *maskable interrupts*.

The interrupt mask does not affect some interrupt signals and therefore cannot be disabled; these are called *non-maskable interrupts* (NMI).

Spurious Interrupts:

A spurious interrupt is a hardware interrupt for which no source can be found. The term phantom interrupt or ghost interrupt may also use to describe this phenomenon.

Difference between System Call and System Interrupt

System Call	System Interrupt
A procedural method through which a computer software requests the operating system's kernel for assistance	An interrupt is an event in which the CPU is asked to do a specific action by outside components in an operating system
Enables an application to interact with the kernel in order to access resources, such as memory or hardware devices	Informs the CPU to pause running the currently executing programs in order to perform some urgent actions
A system call is initiated by the program calling the kernel by executing a special instruction	A system interrupt is initiated by hardware or software
Requires the kernel's attention	Don't require the kernel's attention

Introduction to Linux

Linux is an open-source Unix-like operating system-based family on the Linux kernel, and the OS kernel was first published on 17 September 1991 by *Linus Torvalds*. Typically, Linux is packaged as the Linux distribution, which contains the supporting libraries and system software and kernel, several of which are offered by the GNU Project. Several Linux distributions use the term *"Linux"* in the title, but the Free Software Foundation uses the *"GNU/Linux"* title to focus on the necessity of GNU software, causing a few controversies.

What is Linux File system?

Linux file system is the collection of data and/or files stored in a computer's hard disk or storage, your computer relies on this file system to ascertain the location and positioning of files in your storage, were it not there, the files would act as if they are invisible, obviously causing many problems.

There are actually many different file systems that exist for Linux, if you're wondering which one you should use, we will provide a comprehensive list of the file systems that are supported by Linux.

Linux File system Types:

Ext: "ext" is an acronym that stands for "extended file system" and was created in 1992 and is the very first file system designed specifically for Linux. Its functionality was designed partly based on the UNIX file system. The purpose of its creation originally was to innovate beyond the file system used before it (the MINIX file system) and overcome its limitations.

Ext2: also referred to as "second extended system". Created in 1993, ext2 was designed to be the successor of the original extension system for Linux. It innovated in areas such as storage capacity, and general performance. This file system notably allows for up to 2 TB of data. Like ext, this file system is very old, so it really should just be avoided.

Ext3: ext3, or third extended system, created in 2001, surpasses ext2 in that it is a journaling file system. A journaling file system is a system that records in a separate log changes and updates to files and data before such actions have been completed.

Ext4: ext4, standing for "fourth extended system", was created in 2006. Because this file system overcomes numerous limitations that the third extended system had, it is both widely used, and the default file system that most Linux distros use. While it may not be the most cutting edge, it is absolutely reliable and stable – which
is really valuable in Linux.

JFS: The file system JFS was created by IBM in 1990 and the name JFS is an acronym standing for Journaling File System, as we've already covered this concept with the number 3 file system in this article, you should already be quite familiar with what exactly this means.

XFS: xfs, an acronym that stands for "Extent File System", was created by Silicon Graphics and originally made for their OS "IRIX", but was later given to Linux.

Btrfs: btrfs, which is yet another acronym standing for B Tree File System, created by Oracle in 2009. It is regarded as a rival file system to ext4, though it's consensus that overall ext4 is the better file system, as it transfers data faster and offers more stability but although this is the case, that does not mean btrfs isn't worth looking into.

Some Basic Commands:

1. **mkdir Command**

   The mkdir command is used to create a new directory under any directory.
   mkdir **<directory** name

2. **rmdir Command**

   The rmdir command is used to delete a directory.
   rmdir **<directory** name**>**

3. **ls Command**

   The ls command is used to display a list of content of a directory.
   ls

4. **cd Command**
   The cd command is used to change the current directory.
   cd **<directory** name**>**

5. **touch Command**
   The touch command is used to create empty files. We can create multiple empty files by executing it once.
   touch **<file** name**>**
   touch **<file1> <file2>** ....

6. **cat Command**

   The cat command is a multi-purpose utility in the Linux system. It can be used to create a file, display content of the file, copy the content of one file to another file, and more.
   cat **<file** name**>**

7. **rm Command**

   The rm command is used to remove a file.
   rm <file name>

8. **cp Command**

   The cp command is used to copy a file or directory.
   cp **<existing** file name**> <new** file name**>**

9. **mv Command**

   The mv command is used to move a file or a directory form one location to another location.
   mv **<file** name**> <directory** path**>**

10. **rename Command**

    The rename command is used to rename files. It is useful for renaming a large group of files.
    rename 's/old-name/new-name/' files

11. **head Command**

    The head command is used to display the content of a file. It displays the first 10 lines of a file.
    head **<file** name**>**

12. **tail Command**

    The tail command is similar to the head command. The difference between both commands is that it displays the last ten lines of the file content. It is useful for reading the error message.
    tail **<file** name**>**

13. **tac Command**

    The tac command is the reverse of cat command, as its name specified. It displays the file content in reverse order (from the last line).
    tac **<file** name**>**

14. **more command**

    The more command is quite similar to the cat command, as it is used to display the file content in the same way that the cat command does. The only difference between both commands is that, in case of larger files, the more command displays screenful output at a time.
    more **<file** name**>**

15. **less Command**

    The less command is similar to the more command. It also includes some extra features such as 'adjustment in width and height of the terminal.' Comparatively, the more command cuts the output in the width of the terminal.
    less **<file** name**>**

16. **useradd Command**

    The useradd command is used to add or remove a user on a Linux server.
    useradd username

17. **passwd command**

    The passwd command is used to create and change the password for a user.
    passwd **<username>**

18. **groupadd Command**

    The groupadd command is used to create a user group.
    groupadd **<group** name**>**

19. **grep Command**

The grep is the most powerful and used filter in a Linux system. The 'grep' stands for "**global *regular expression print.*" *It is useful for searching* the content from a file. Generally, it is used with the pipe.
command | grep **<searchWord>**

20. **sed command**

The sed command is also known as **stream editor**. It is used to edit files using a regular expression. It does not permanently edit files; instead, the edited content remains only on display. It does not affect the actual file.
command | sed 's/**<oldWord>**/**<newWord>**/'

21. **find Command**

The find command is used to find a particular file within a directory. It also supports various options to find a file such as byname, by type, by date, and more.
find . -name "*.pdf"

22. **cal Command**

The cal command is used to display the current month's calendar with the current date highlighted.
Cal

23. **mount Command**

The mount command is used to connect an external device file system to the system's file system.
mount -t type **<device>** **<directory>**

24. **mail Command**

The mail command is used to send emails from the command line.
mail -s "Subject" **<recipient** address>

25. **ping Command**

The ping command is used to check the connectivity between two nodes, that is whether the server is connected. It is a short form of "Packet Internet Groper."
ping **<destination>**

26. **SSH command**

SSH which stands for Secure Shell, It is used to connect to a remote computer securely. Compare to Telnet, SSH is secure wherein the client /server connection is authenticated using a digital certificate and passwords are encrypted. Hence it's widely used by system administrators to control remote Linux servers.
SSH username@ip-address or hostname

27. **FTP command**

FTP **is file transfer protocol.** It's the **most preferred protocol for data transfer** amongst computers.
You can use FTP to –

- Logging in and establishing a connection with a remote host
- Upload and download files
- Navigating through directories
- Browsing contents of the directories

The syntax to establish an FTP connection to a remote host is –
ftp hostname="" or=""

28. **Telnet command**

Telnet helps to –

- connect to a remote Linux computer
- run programs remotely and conduct administration

telnet hostname="" or=""
Example:
telnet localhost

Linux File Permissions

Every file and directory in your UNIX/Linux system has following 3 permissions defined for all the 3 owners discussed above.

- **Read:** This permission give you the authority to open and read a file. Read permission on a directory gives you the ability to lists its content.
- **Write:** The write permission gives you the authority to modify the contents of a file. The write permission on a directory gives you the authority to add, remove and rename files stored in the
- directory. Consider a scenario where you have to write permission on file but do not have write permission on the directory where the file is stored. You will be able to modify the file contents. But
- you will not be able to rename, move or remove the file from the directory.
- **Execute:** In Windows, an executable program usually has an extension ".exe" and which you can
- easily run. In Unix/Linux, you cannot run a program unless the execute permission is set. If the
- execute permission is not set, you might still be able to see/modify the program code(provided read & write permissions are set), but not run it.

Permissions are as follows

permission	on a file	on a directory
r (read)	read file content (cat)	read directory content (ls)
w (write)	change file content (vi)	create file in directory (touch)
x (execute)	execute the file	enter the directory (cd)

Permission set:

position	characters	ownership
1	-	denotes file type
2-4	rw-	permission for user
5-7	rw-	permission for group
8-10	r--	permission for other

You can change file permissions in Linux using the chmod command. The basic syntax is chmod [permissions] [filename]. For example, chmod u+w file.txt will add write permission to the owner of the file.

System Variable

Your interaction with Linux Bash shell will become very pleasant, if you use PS1, PS2, PS3, PS4, and PROMPT_COMMAND effectively. PS stands for prompt statement. This article will give you a jumpstart on the Linux command prompt environment variables using simple examples.

PS0 The value of this parameter is expanded (see PROMPTING below) and displayed by interactive shells after reading a command and before the command is executed.

PS1T he value of this parameter is expanded (see PROMPTING below) and used as the primary prompt string. The default value is \s-\v\$.

PS2 The value of this parameter is expanded as with PS1 and used as the secondary prompt string. The default is >

PS3 The value of this parameter is used as the prompt for the select command

PS4 The value of this parameter is expanded as with PS1 and the value is printed before each command bash displays during an execution trace. The first character of PS4 is replicated multiple times, as necessary, to indicate multiple levels of indirection.

Shell Programming

The shell can be defined as a command interpreter within an operating system like Linux/GNU or Unix. It is a program that runs other programs. Simply put, the shell is a program that takes commands from the keyboard and gives them to the operating system to perform.

The shell sends the result to the user over the screen when it has completed running a program which is the common output device. That's why it is known as **"command interpreter"**.

The shell is not just a command interpreter. Also, the shell is a programming language with complete constructs of a programming language such as **functions, variables, loops, conditional execution**, and many others.

Types of Linux Shells

Linux offers different shell types for addressing various problems through unique features. The shells developed alongside Unix often borrowed features from one another as development progressed. Below is a brief overview of different shell types and their features.

1. Bourne Shell (sh)

The **Bourne shell** was the first default shell on Unix systems, released in 1979. The shell program name is **sh**, and the traditional location is */bin/sh*. The prompt switches to **$**, while the root prompt is **#**.

2. C Shell (csh)

The **C shell (csh)** is a Linux shell from the late 1970s whose main objective was to improve interactive use and mimic the C language. Since the Linux kernel is predominantly written in C, the shell aims to provide stylistic consistency across the system.

3. TENEX C Shell (tcsh)

The **TENEX C shell (tcsh)** is an extension of the C shell (**csh**) merged in the early 1980s. The shell is backward compatible with csh, with additional features and concepts borrowed from the TENEX OS.

4. KornShell (ksh)

The **KornShell (ksh)** is a Unix shell and language based on the Bourne shell (sh) developed in the early 1980s. The location is in */bin/ksh* or */bin/ksh93*, while the prompt is the same as the Bourne shell.

5. Debian Almquist Shell (dash)

The **Debian Almquist Shell (dash)** is a Unix shell developed in the late 1990s from the Almquist shell (ash), which was ported to Debian and renamed.Dash is famous for being the default shell for Ubuntu and Debian. The shell is minimal and POSIX compliant, making it convenient for OS startup scripts.

6. Bourne Again Shell (bash)

The **Bourne Again shell** is a Unix shell and command language created as an extension of the Bourne shell (**sh**) in 1989. The shell program is the default login shell for many Linux distributions and earlier versions of macOS. he shell name shortens to **bash**, and the location is */bin/bash*. Like the Bourne shell, the bash prompt is **$** for a regular user and **#** for root.

Shell Variable:

A shell variable is a variable that is available only to the current shell. In contrast, an environment variable is available system wide and can be used by other applications on the system. A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

Variable Names
The name of a variable can contain only letters (a to z or A to Z), numbers (0 tc 9) or the underscore character (_).By convention, Unix shell variables will have their names in UPPERCASE.

The following examples are valid variable names –

_ALI
TOKEN_A
VAR_1
VAR_2

Following are the examples of invalid variable names –

2_VAR
-VARIABLE
VAR1-VAR2
VAR_A!
The reason you cannot use other characters such as !, *, or - is that these characters have a specia meaning for the shell.

Defining Variables
Variables are defined as follows –
variable_name=variable_value
For example –
NAME="Deepika"
Var2=29
Variables of this type are called **scalar variables.** A scalar variable can hold only one value at a time.

Accessing Values
To access the value stored in a variable, prefix its name with the dollar sign ($) –

For example, the following script will access the value of defined variable NAME and print it on STDOUT –

```
#!/bin/sh
NAME="Deepika"
```

Echo $NAME

The above script will produce the following value –
Deepika

Read-only Variables
Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.
For example, the following script generates an error while trying to change the value of NAME –

```
#!/bin/sh
NAME="Deepika"
Readonly NAME
```


NAME="Aparna"

The above script will generate the following result –

/bin/sh: NAME: This variable is read only.

Unsetting Variables
Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.
Following is the syntax to unset a defined variable using the **unset** command –
Unset variable_name
The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works –

```
#!/bin/sh
Name="Deepika"
Unset Name
Echo $NAME
```

Wildcard Symbol
A wildcard in Linux means it might be a symbol or set of symbols representing other characters. It is generally used in substituting any string or a character. The basic set of wildcards in Linux includes below:

- * – This wildcard represents all the characters
- ? – This wildcard represents a single character
- [] – This wildcard represents a range of characters.

Types of Linux Wildcard:

- **Question Mark Wildcard '?'** – The wildcard '?' means it will match a single character. For example, S?? n will match anything that will begin with S and end with n and has two characters between them.
- **Star Wildcard '*'** – The wildcard '*' means it will match any number of characters or a set of characters. For example, S**n will match anything between S and n. The number of characters between them do not count.
- **Bracket value Wildcard '[]'**: The wildcard '[]' means it will match characters that are enclosed in square braces. For example, S[on]n will match only Son and Snn. We can also specify characters in braces like S[a-d]n, which will match San, Sbn, Scn, Sdn.

Shell Meta Characters

The command options, option arguments and command arguments are separated by the space character. However, we can also use special characters called metacharacters in a Linux command that the shell interprets rather than passing to the command.

Command line arguments

Command-line arguments are parameters that are passed to a script while executing them in the bash shell. They are also known as positional parameters in Linux. We use command-line arguments to denote the position in memory where the command and it's associated parameters are stored.

Echo

echo is a shell builtin in Bash and most of the other popular shells like Zsh and Ksh. Its behavior is slightly different from shell to shell.

The syntax for the echo command is as follows:

echo [-neE] [ARGUMENTS]

- When the –n option I used, the trailing newline is suppressed.
- If the –e option I given, the following backslash-escaped characters will be interpreted:
- \\-Displays a backslash character.
- \a- Alert (BEL)
- \b- Displays a backspace character.
- \c- Suppress any further output.
- \e- Displays an escape character.
- \f- Displays a form feed character.
- \n- Displays a new line.
- \r- Displays a carriage return.
- \t- Displays a horizontal tab.
- \v- Displays a vertical tab.
- The –E option disables the interpretation of the escape characters. This is the default.

Processes:

Process: Instance of an executing program.

- State of execution

  program counter, stack pointer

- Parts and temporary holding area

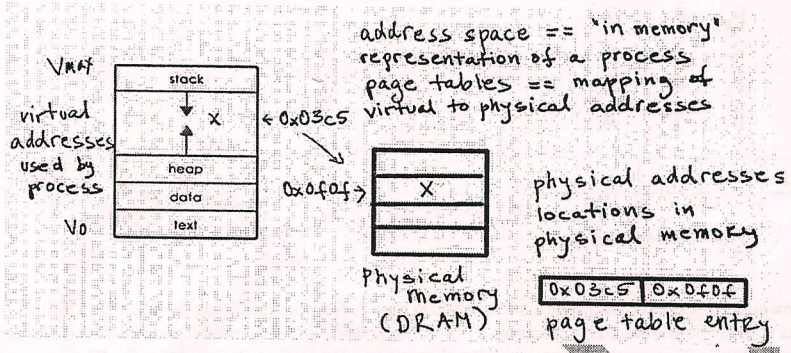
  data, register state, occupies state in memory

- May require special hardware

  I/O devices

Process is a state of a program when executing and loaded in memory (active state) as opposed to application (static state).
In other words, we write the computer programs in the form of a text file, thus when we run them, these turn into processes that complete all of the duties specified in the program.

Process memory is divided into four section.



The process contains several sections: Text, Data, Heap and Stack.

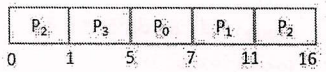
- Text Section contains the program code. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers.
- Process stack contains temporary data such as function parameters, return addresses and local variables.
- Data section contains the global and static variables.
- Heap is memory that is dynamically allocated during process run time.
- Preemptive Scheduling

Preemptive scheduling is one which can be done in the circumstances when a process switches from **running state** to **ready state** or from **waiting state** to **ready state**. Here, the resources (CPU cycles) are allocated to the process for the **limited** amount of time and then is taken away, and the process is placed back in the ready queue again if it still has CPU burst time remaining. The process stays in ready queue till it gets next chance to execute.

If a process with high priority arrives in the ready queue, it does not have to wait for the current process to complete its burst time. Instead, the current process is interrupted in the middle of execution and is placed in the ready queue till the process with high priority is utilizing the CPU cycles. In this way, each process in the ready queue gets some time to run CPU. It makes the preemptive scheduling flexible but, increases the overhead of switching the process from running state to ready state and vise-verse.
Algorithms that work on preemptive scheduling are Round Robin. Shortest Job First (SJF) and Priority scheduling may or may not come under preemptive scheduling.

Let us take an example of Preemptive Scheduling, look in the picture below. We have four processes P0, P1, P2, P3. Out of which, P2 arrives at time 0. So the CPU is allocated to the process P2 as there is no other process in the queue. Meanwhile, P2 was executing, P3 arrives at time 1, now the remaining time for process P2 (5 milliseconds) which is larger than the time required by P3 (4 milli-sec). So CPU is allocated to processor P3.

Process	Arrival Time	CPU Burst Time in millisec.
P_0	3	2
P_1	2	4
P_2	0	6
P_3	1	4



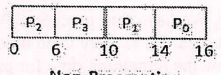
Preemptive Scheduling

Meanwhile, P3 was executing, process P1 arrives at time 2. Now the remaining time for P3 (3 milliseconds) is less than the time required by processes P1 (4 milliseconds) and P2 (5 milliseconds). So P3 is allowed to continue. While P3 is continuing process P0 arrives at time 3, now the remaining time for P3 (2 milliseconds)
is equal to the time required by P0 (2 milliseconds). So P3 continues and after P3 terminates the CPU is allocated to P0 as it has less burst time than other. After P0 terminates, the CPU is allocated to P1 and then to P2.

Non-Preemptive Scheduling

Non-preemptive Scheduling is one which can be applied in the circumstances when a process **terminates**, or a process switches from **running** to **waiting state**. In Non-Preemptive Scheduling, once the resources (CPU) is allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting state
Unlike preemptive scheduling, non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits for the process to complete its CPU burst time and then it can allocate the CPU to another process.
In Non-preemptive scheduling, if a process with long CPU burst time is executing then the other process will have to wait for a long time which increases the average waiting time of the processes in the ready queue. However, the non-preemptive scheduling does not have any overhead of switching the processes from ready queue to CPU but it makes the scheduling rigid as the process in execution is not even preempted for a process with higher priority.

Process	Arrival Time	CPU Burst Time in millisec.
P_0	3	2
P_1	2	4
P_2	0	6
P_3	1	4



Non-Preemptive Scheduling

Let us solve the above scheduling example in non-preemptive fashion. As initially, the process P2 arrives at time 0, so CPU is allocated to the process P2 it takes 6 milliseconds to execute. In between all the processes i.e. P0, P1, P3 arrives into ready queue. But all waits till process P2 completes its CPU burst time. Then process that arrives after P2 i.e. P3 is then allocated the CPU till it finishes it's burst time. Similarly, then P1 executes, and CPU is later given to process P0.

Process management; Process life cycle

Program vs Process: A process is a program in execution. For example, when we write a program in C or C++ and compile it, the compiler creates binary code. The original code and binary code are both programs. When we actually run the binary code, it becomes a process.
A process is an 'active' entity instead of a program, which is considered a 'passive' entity. A single program can create many processes when run multiple times; for example, when we open a .exe or binary file multiple times, multiple instances begin (multiple processes are created).

Process Management
If operating system supports multiple users than services under this are very important. In this regard operating systems has to keep track of all the completing processes, Schedule them, dispatch them one after another. But user should feel that he has the full control of the CPU.

some of the systems calls in this category are as follows.

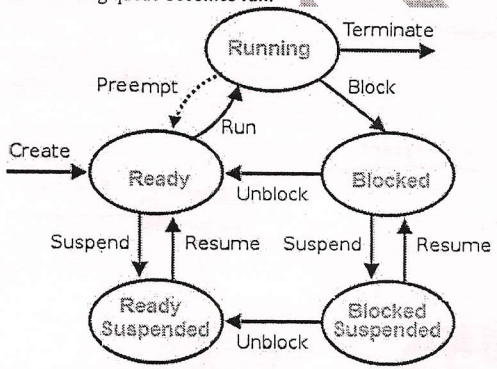
1. create a child process identical to the parent.
2. Terminate a process
3. Wait for a child process to terminate
4. Change the priority of process
5. Block the process
6. Ready the process
7. Dispatch a process
8. Suspend a process
9. Resume a process
10. Delay a process
11. Fork a process

Attributes or Characteristics of a Process: A process has the following attributes.

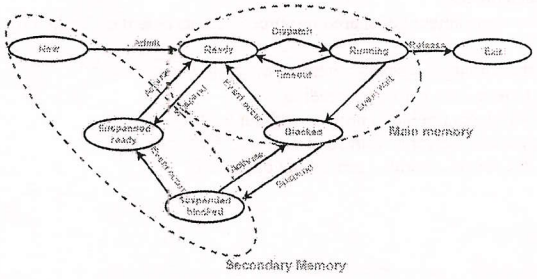
1. Process Id: A unique identifier assigned by the operating system
2. Process State: Can be ready, running, etc.
3. CPU registers: Like the Program Counter (CPU registers must be saved and restored when a process is swapped in and out of CPU)
4. Accounts information: Amount of CPU used for process execution, time limits, execution ID etc
5. I/O status information: For example, devices allocated to the process,open files, etc
6. CPU scheduling information: For example, Priority (Different processes may have different priorities, for example a shorter process assigned high priority in the shortest job first scheduling)
All of the above attributes of a process are also known as the ***context of the process***. Every process has its own process control block(PCB), i.e each process will have a unique PCE. All of the above attributes are part of the FCB.

States of Process: A process is in one of the following states:

1. **New:** Newly Created Process (or) being-created process.
2. **Ready:** After creation process moves to Ready state, i.e. the process is ready for execution.
3. **Run:** Currently running process in CPU (only one process at a time can be under execution in a single processor).
4. **Wait (or Block):** When a process requests I/O access.
5. **Complete (or Terminated):** The process completed its execution.
6. **Suspended Ready:** When the ready queue becomes full, some processes are moved to suspended ready state
7. **Suspended Block:** When waiting queue becomes full.



States of a process are as following:



- **New (Create) –** In this step, the process is about to be created but not yet created, it is the program which is present in secondary memory that will be picked up by OS to create the process.
- **Ready –** New -> Ready to run. After the creation of a process, the process enters the ready state i.e. the process is loaded into the main memory. The process here is ready to run and is waiting to get the CPU time for its execution. Processes that are ready for execution by the CPU are maintained in a queue for ready processes.
- **Run –** The process is chosen by CPU for execution and the instructions within the process are executed by any one of the available CPU cores.
- **Blocked or wait –** Whenever the process requests access to I/O or needs input from the user or needs access to a critical region(the lock for which is already acquired) it enters the blocked or wait state. The process continues to wait in the main memory and does not require CPU. Once the I/O operation is completed the process goes to the ready state.
- **Terminated or completed –** Process is killed as well as PCB is deleted.
- **Suspend ready –** Process that was initially in the ready state but was swapped out of main memory(refer Virtual Memory topic) and placed onto external storage by scheduler is said to be in suspend ready state. The process will transition back to ready state whenever the process is again brought onto the main memory.
- **Suspend wait or suspend blocked –** Similar to suspend ready but uses the process which was performing I/O operation and lack of main memory caused them to move to secondary memory. When work is finished it may go to suspend ready.

CPU and I/O Bound Processes: If the process is intensive in terms of CPU operations then it is called CPU bound process. Similarly, If the process is intensive in terms of I/O operations then it is called I/O bound process.

Types of schedulers:

- **Long term – performance –** Makes a decision about how many processes should be made to stay in the ready state, this decides the degree of multiprogramming. Once a decision is taken it lasts for a long time hence called long term scheduler.
- **Short term – Context switching time –** Short term scheduler will decide which process to be executed next and then it will call dispatcher. A dispatcher is a software that moves process from ready to run and vice versa. In other words, it is context switching.
- **Medium term – Swapping time –** Suspension decision is taken by medium term scheduler. Medium term scheduler is used for swapping that is moving the process from main memory to secondary and vice versa.

Process Schedulers in Operating System

The Process manager's activity is process scheduling, which involves removing the running process from the CPU and selecting another process based on a specific strategy. The scheduler's purpose is to implement the virtual machine so that each process appears to be running on its own computer to the user.

Process scheduling is an essential part of a Multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

There are three types of process schedulers.

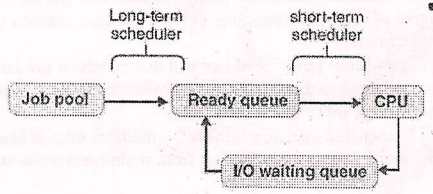
Long Term or job scheduler:
It brings the new process to the 'Ready State'. It controls the ***Degree of Multi-programming***, i.e., the number of processes present in a ready state at any point in time. It is important that the long-term *scheduler make a careful selection of both I/O and CPU-bound processes.* I/O-bound tasks are which use much of their time in input and output operations while CPU-bound processes are which spend their time on the CPU. The job scheduler increases efficiency by maintaining a balance between the two. They operate at a high level and are typically used in batch-processing systems.

Short-term or CPU scheduler:
It is responsible for selecting one process from the ready state for scheduling it on the running state. Note: Short-term scheduler only selects the process to schedule it doesn't load the process on running. Here is when all the scheduling algorithms are used. The CPU scheduler is responsible for ensuring there is no starvation owing to high burst time processes.
The dispatcher is responsible for loading the process selected by the Short-term scheduler on the CPU (Ready to Running State) Context switching is done by the dispatcher only. A dispatcher does the following:

1. Switching context.
2. Switching to user mode.
3. Jumping to the proper location in the newly loaded program.

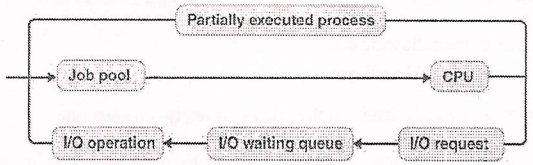
CPU scheduler is another name for Short-Term scheduler. It chooses one job from the ready queue and then sends it to the CPU for processing.

To determine which work will be dispatched for execution, a scheduling method is utilised. The Short-Term scheduler's task can be essential in the sense that if it chooses a job with a long CPU burst time, all subsequent jobs will have to wait in a ready queue for a long period. This is known as hunger, and it can occur if the Short-Term scheduler makes a mistake when selecting the work.



Medium-term scheduler:
It is responsible for suspending and resuming the process. It mainly does swapping (moving processes from main memory to disk and vice versa). 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. It is helpful in maintaining a perfect balance between the I/O bound and the CPU bound. It reduces the degree of multiprogramming.

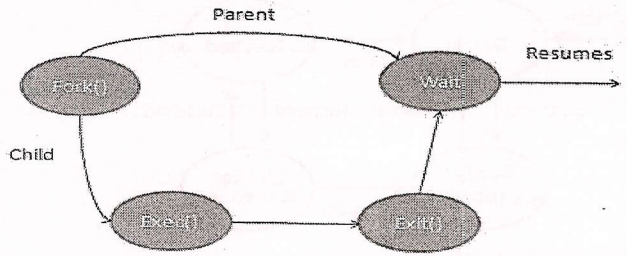
The switched-out processes are handled by the Medium-Term scheduler. If the running state processes require some IO time to complete, the state must be changed from running to waiting.
This is accomplished using a Medium-Term scheduler. It stops the process from executing in order to make space for other processes. Swapped out processes are examples of this, and the operation is known as swapping. The Medium-Term scheduler here is in charge of stopping and starting processes.
The degree of multiprogramming is reduced. To have a great blend of operations in the ready queue, swapping is required.



Some other Schedulers:

- **I/O schedulers:** I/O schedulers are in charge of managing the execution of I/O operations such as reading and writing to discs or networks. They can use various algorithms to determine the order in which I/O operations are executed, such as FCFS (First-Come, First-Served) or RR (Round Robin).
- **Real-time schedulers:** In real-time systems, real-time schedulers ensure that critical tasks are completed within a specified time frame. They can prioritize and schedule tasks using various algorithms such as EDF (Earliest Deadline First) or RM (Rate Monotonic).

Operations on Process
Process Creation



- Through appropriate system calls, processes may create other processes.
- The process which creates other process, is termed the **parent** of the other process, while the created sub-process is termed its **child**.
- Each process is given an integer identifier, termed as process identifier, or PID. The parent PID (PPID) is also stored for each process.
- A child process may receive some amount of shared resources with its parent depending on system implementation.
- There are two options for the parent process after creating the child :
- Wait for the child process to terminate before proceeding.
- Run concurrently with the child, continuing to process without waiting.
- It is also possible for the parent to run for a while, and then wait for the child
- Later, which might occur in a sort of a parallel processing operation.

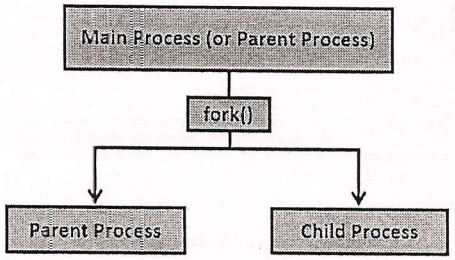
Process Termination

Processes may also be terminated by the system for a variety of reasons, including:

- The inability of the system to deliver the necessary system resources.
- In response to a KILL command or other unhandled process interrupts.
- A parent may kill its children if the task assigned to them is no longer needed
- If the parent exits, the system may or may not allow the child to continue without a parent.
- When a process ends, all of its system resources are freed up, open files flushed and closed, etc. The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate, or eventually returned to init if the process already became an orphan.

The processes which are trying to terminate but cannot do so because their parent is not waiting for them are termed **orphans**.
These are eventually inherited by init as orphans and killed off.

Process creation is achieved through the **fork() system call**. The newly created process is called the child process and the process that initiated it (or the process when execution is started) is called the parent process. After the fork() system call, now we have two processes - parent and child processes.



After creation of the child process, let us see the fork() system call details.

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

pid_t fork(void);
```

Creates the child process. After this call, there are two processes, the existing one is called the parent process and the newly created one is called the child process.
The fork() system call returns either of the three values –

- Negative value to indicate an error, i.e., unsuccessful in creating the child process.
- Returns a zero for child process.
- Returns a positive value for the parent process. This value is the process ID of the newly created
- child process.

Let us consider a simple program.

Execution Steps

```
File name: basicfork.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
  fork();
  printf("Called fork() system call\n");
  return 0;
}
```

Compilation
gcc basicfork.c -o basicfork

Execution/Output
Called fork() system call
Called fork() system call

Note – Usually after fork() call, the child process and the parent process would perform different tasks. If the same task needs to be run, then for each fork() call it would run 2 power n times, where **n** is the number of times fork() is invoked.
In the above case, fork() is called once, hence the output is printed twice (2 power 1). If fork() is called, say 3 times, then the output would be printed 8 times (2 power 3). If it is called 5 times, then it prints 32 times and so on and so forth.
Having seen fork() create the child process, it is time to see the details of the parent and the child processes.
fork(): System call to create a child process.

```
shashi@linuxtechi ~}$ man fork
```

This will yield output mentioning what is fork used for, syntax and along with all the required details.
The syntax used for the fork system call is as below,

```
pid_t fork(void);
```

Fork system call creates a child that differs from its parent process only in **pid(process ID)** and **ppid(parent process ID)**. Resource utilization is set to zero. File locks and pending signals are not inherited. (In Linux "fork" is implemented as "**copy-on-write()**").

Note:- "Copy on write" -> Whenever a fork() system call is called, a copy of all the pages(memory) related

to the parent process is created and loaded into a separate memory location by the Operating System for the child process. But this is not needed in all cases and may be required only when some process writes to this address space or memory area, then only separate copy is created/provided.

wait()

wait() system call suspends execution of current process until a child has exited or until a signal has delivered whose action is to terminate the current process or call signal handler.

```
pid_t wait(int * status);
```

There are other system calls related to wait as below,

1) **waitpid()**: suspends execution of current process until a child as specified by pid arguments has exited or until a signal is delivered.

```
pid_t waitpid (pid_t pid, int *status, int options);
```

2) **wait3()**: Suspends execution of current process until a child has exited or until signal is delivered.

```
pid_t wait3(int *status, int options, struct rusage *rusage);
```

3) **wait4()**: As same as wait3() but includes pid_t pid value.

```
pid_t wait3(pid_t pid, int *status, int options, struct rusage *rusage);
```

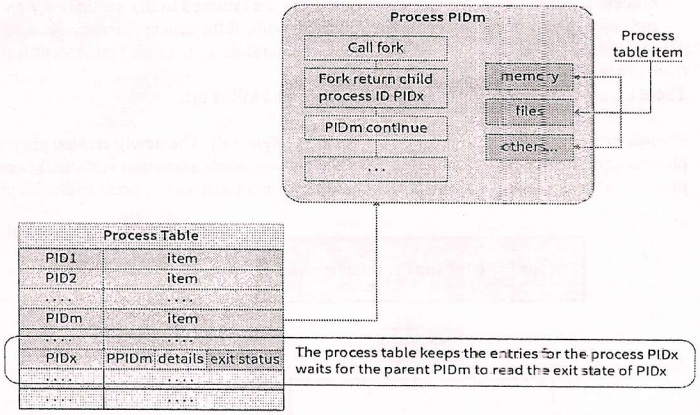
exec()

exec() family of functions or sys calls replaces current process image with new process image.
There are functions like **execl, execlp,execle,execv, execvp** and **execvpe** are used to execute a file.
These functions are combinations of array of pointers to null terminated strings that represent the argument list, this will have path variable with some environment variable combinations.

Orphan and zombie processes:

Zombie Process:

- A process which has finished the execution but still has entry in the process table to report to its
- parent process is known as a zombie process.
- **Zombie process** is also known as ***"dead"*** process. Ideally when a process completes its execution, it's entry from the process table should be removed but this does not happen in case of zombie a process.



Note: Process table is a data structure in RAM to store information about a process.

What happens with the zombie processes?

- wait() system call is used for removal of zombie processes.
- wait() call ensures that the parent doesn't execute or sits idle till the child process is completed.
- When the child process completes executing ,the parent process removes entries of the child process from the process table. This is called "reaping of child".

```
#include<stdlib.h> #include<sys/types.h> #include<unistd.h>
int main()
{
int pid=fork(); //create new process
if(pid>0)
{
exit(0); //parent exit before child
}
else if(pid==0)
{
sleep(60);      // child sleeps for 60 second
}
return 0;
}
```

Signals

In Linux, Signals are the interrupts that are sent to the program to specify that an important event has occurred.

Types of signals:

SIGHUP

This signal indicates that someone has killed the controlling terminal.

SIGINT

This is the signal that being sent to your application when it is running in a foreground in a terminal and someone presses CTRL-C.

SIGKILL

The SIGKILL signal is sent to a process to cause it to terminate immediately (**kill**).

SIGBUS

The SIGBUS signal is sent to a process when it causes a **bus error**.

SIGCHLD

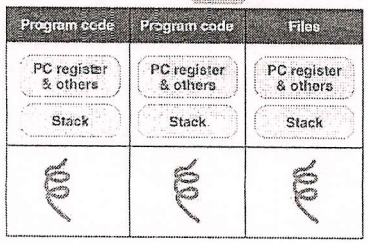
The SIGCHLD signal is sent to a process when a **child process** terminates, is interrupted, or resumes after being interrupted.

Generating and handling signal

A signal is said to be *generated* for (or sent to) a process when the event that causes the signal first occurs. Examples of such events include detection of hardware faults, timer expiration, and terminal activity, as well as the invocation of kill.

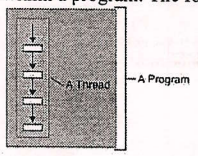
Threads

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.

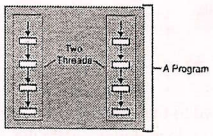


Three threads of same process

Each thread belongs to exactly one process and no thread can exist outside a process. A single thread also has
a beginning, a sequence, and an end. At any given time during the runtime of the thread, there is a single point of execution. However, a thread itself is not a program; a thread cannot run on its own. Rather, it runs within a program. The following figure shows this relationship.



The real excitement surrounding threads is not about a single sequential thread. Rather, it's about the use of multiple threads running at the same time and performing different tasks in a single program. This use is illustrated in the next figure.



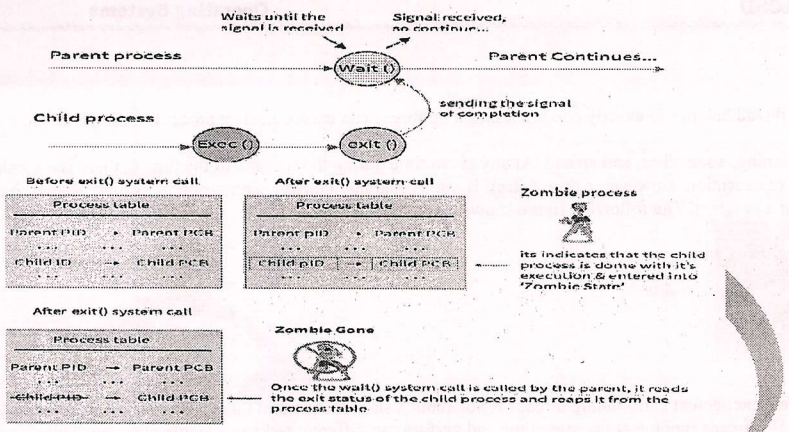
user and kernel threads

User-Level Thread

The user-level thread is ignored by the operating system. User threads are simple to implement and are done so by the user. The entire process is blocked if a user executes a user-level operation of thread blocking. The kernel-level thread is completely unaware of the user-level thread. User-level threads are managed as single-threaded processes by the kernel-level thread.

Kernel-Level Thread

The operating system is recognized by the kernel thread. Each thread and process in the kernel-level thread has its own thread control block as well as process control block in the system. The operating system implements the kernel-level thread. The kernel is aware of all threads and controls them. The kernel-level thread provides a system call for user-space thread creation and management. Kernel threads are more complex to build than user threads. The kernel thread's context switch time is longer.



Zombie Process Code Example

In this code given below, we'll see how a zombie process is created.

- The child process completes its execution by using exit() system call.
- So when the child finishes it's execution **'SIGCHLD'** signal is delivered to the parent process by the kernel. Parents should,ideally, read the child's status from the process table and then delete the child's entry.
- But here the parent does not wait for the child to terminate , rather it does it's own subsequent job, i.e. here sleeping for 60 seconds.
- So the child's exit status is never read by the parent and the child's entry continues to remain there in the process table even when the child has died.

Note: Kernel sends a SIGCHLD signal to the parent process to indicate that the child process has ended

```
#include<stdlib.h>

#include<sys/types.h>
#include<unistd.h>
int main()
{
int pid=fork(); //create new process
if(pid>0)
{
sleep(60); //parent sleeps for 60 sec
}
else
{       exit(0); // child exits before the parent & child becomes zombie
}


return 0;
}
```

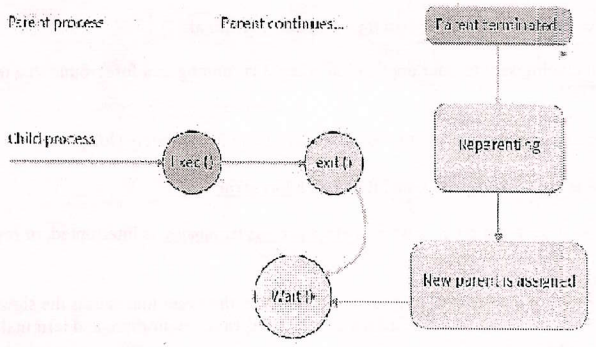

Orphan Process

We'll again use real life analogy to understand the orphan process.

- In the real world orphans are those children whose parents are dead.
- Similarly, a process which is executing (is alive) but it's parent process has terminated (dead) is called
- an orphan process.

What will happen with the orphan processes?

- In the real world orphans are adopted by guardians who look after them.
- Similarly,the orphan process in linux is adopted by a new process , which is mostly init process
- (pid=1) . This is called ***re-parenting.***
- Reparenting is done by the kernel,when the kernel detects an orphan process in os, and assigns a new parent process.
- New parent process asks the kernel for cleaning of the PCB of the orphan process and the new parent waits till the child completes its execution.



Orphan Process Code Example

In the example,the parent process sleeps for 20 seconds while the child process sleeps for 30 seconds.

1. So after sleeping for 20 seconds the parent completes its execution while the child process is still
2. there at least till 30 seconds.
3. When the child process becomes an orphan process , the kernel reassigns the parent process to the
4. child process.
5. As a result the parent process id of the child process before and after sleep() will be different.

Note: Kernel is central component of an operating system that manages operations of computer and hardware.

Difference between Process and Thread

Parameter	Process	Thread
Definition	Process means a program is in execution.	Thread means a segment of a process.
Lightweight	The process is not Lightweight.	Threads are Lightweight.
Termination time	The process takes more time to terminate.	The thread takes less time to terminate.
Creation time	It takes more time for creation.	It takes less time for creation.
Communication	Communication between processes needs more time compared to thread.	Communication between threads requires less time compared to processes.
Context switching time	It takes more time for context switching.	It takes less time for context switching.
Resource	Process consume more resources.	Thread consume fewer resources.
Treatment by OS	Different process are tread separately by OS.	All the level peer threads are treated as a single task by OS.
Memory	The process is mostly isolated.	Threads share memory.
Sharing	It does not share data	Threads share data with each other.

Multithreading

Thread is a programming technique to improve application performance through parallel processes. For example, in a browser, multiple tabs can be different threads.

Threads operate faster than processes due to following reasons:

1. Thread creation is much faster
2. Context switching between threads is much faster
3. Threads can be terminated easily
4. Communication between threads is faster

Multithreading is not supported by the language standard in C. POSIX Threads (or Pthreads) is a POSIX standard for threads. Implementation of pthread is available with gcc compiler.

Following is a basic program using **pthreads:**

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


void *myThread(void *vargp)
{
   sleep(1);
   printf("Inside Thread \n");
   return NULL;
}

int main()
{
   pthread_t thread_id;
   printf("Before Thread\n");
   pthread_create(&thread_id, NULL, myThread, NULL);
   pthread_join(thread_id, NULL);
   printf("After Thread\n");
   exit(0);
}
```

Explanation of the above code

In main(), we declare a variable called thread_id, which is of type pthread_t, which is an integer used to identify the thread in the system. After declaring thread_id, we call pthread_create() function to create a thread. pthread_create() takes 4 arguments.

- The first argument is a pointer to thread_id which is set by this function.
- The second argument specifies attributes. If the value is NULL, then default attributes shall be used.
- The third argument is name of function to be executed for the thread to be created.
- The fourth argument is used to pass arguments to the function, myThread.

The pthread_join() function for threads is the equivalent of wait() for processes. A call to pthread_join blocks the calling thread until the thread with identifier equal to the first argument terminates.

Compile the above code using:

gcc code.c –lpthread

Memory management

Memory is the important part of the computer that is used to store the data. At any time, many processes are competing for it. Moreover, to increase performance, several processes are executed simultaneously. For this, we must keep several processes in the main memory, so it is even more important to manage them

effectively.

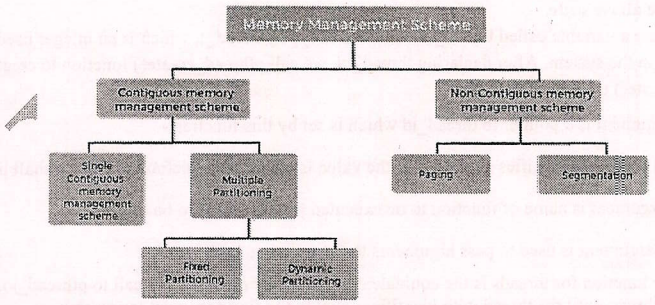
What is the need of Memory management:

- Memory manager is used to keep track of the status of memory locations, whether it is free or allocated. It addresses primary memory by providing abstractions so that software perceives a large memory is allocated to it.
- Memory manager permits computers with a small amount of main memory to execute programs larger than the size or amount of available memory. It does this by moving information back and forth between primary memory and secondary memory by using the concept of swapping.
- The memory manager is responsible for protecting the memory allocated to each process from being corrupted by another process. If this is not ensured, then the system may exhibit unpredictable behavior.
- Memory managers should enable sharing of memory space between processes. Thus, two programs can reside at the same memory location although at different times.

Memory Management Techniques:

The memory management techniques can be classified into following main categories:

- Contiguous memory management schemes
- Non-Contiguous memory management schemes



Classification of memory management schemes

Contiguous memory management schemes:

In a Contiguous memory management scheme, each program occupies a single contiguous block of storage locations, i.e., a set of memory locations with consecutive addresses.

Single contiguous memory management schemes:

In this scheme, the main memory is divided into two contiguous areas or partitions. The operating systems reside permanently in one partition, generally at the lower memory, and the user process is loaded into the other partition.

Multiple Partitioning:

The single Contiguous memory management scheme is inefficient as it limits computers to execute only one program at a time resulting in wastage in memory space and CPU time. The problem of inefficient CPU use can be overcome using multiprogramming that allows more than one program to run concurrently.
To switch between two processes, the operating systems need to load both processes into the main memory The operating system needs to divide the available main memory into multiple parts to load multiple processes into the main memory. Thus multiple processes can reside in the main memory simultaneously.

The multiple partitioning schemes can be of two types:

- Fixed Partitioning
- Dynamic Partitioning

Fixed Partitioning

The main memory is divided into several fixed-sized partitions in a fixed partition memory management scheme or static partitioning. These partitions can be of the same size or different sizes. Each partition can hold a single process. The number of partitions determines the degree of multiprogramming, i.e., the maximum number of processes in memory. These partitions are made at the time of system generation and remain fixed after that.

Dynamic Partitioning

The dynamic partitioning was designed to overcome the problems of a fixed partitioning scheme. In a dynamic partitioning scheme, each process occupies only as much memory as they require when loaded for processing. Requested processes are allocated memory until the entire physical memory is exhausted or the remaining space is insufficient to hold the requesting process. In this scheme the partitions used are of variable size, and the number of partitions is not defined at the system generation time.

Partitioning Algorithms:

There are various algorithms which are implemented by the Operating System in order to find out the holes in the linked list and allocate them to the processes.

1. First Fit Algorithm

First Fit algorithm scans the linked list and whenever it finds the first big enough hole to store a process, it stops scanning and load the process into that hole.
This procedure produces two partitions. Out of them, one partition will be a hole while the other partition will store the process.
First Fit algorithm maintains the linked list according to the increasing order of starting index.

2. Best Fit Algorithm

The Best Fit algorithm tries to find out the smallest hole possible in the list that can accommodate the size requirement of the process.

3. Worst Fit Algorithm

The worst fit algorithm scans the entire list every time and tries to find out the biggest hole in the list which can fulfill the requirement of the process.

Despite of the fact that this algorithm produces the larger holes to load the other processes, this is not the better approach due to the fact that it is slower because it searches the entire list every time again and again.

The first fit algorithm is **the best algorithm** among all because

1. It takes lesser time compare to the other algorithms.
2. It produces bigger holes that can be used to load other processes later on.
3. It is easiest to implement.

PROBLEM: Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worstfit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?

ANSWER:
First-fit: 212K is put in 500K partition
417K is put in 600K partition
112K is put in 288K partition (new partition 288K=500K-212K) 426K must wait
Best-fit: 212K is put in 300K partition
417K is put in 500K partition
112K is put in 200K partition
426K is put in 600K partition
Worst-fit: 212K is put in 600K partition
417K is put in 500K partition
112K is put in 388K partition
426K must wait

In this example, best-fit turns out to be the best.

Compaction in Operating System

Compaction is a technique to collect all the free memory present in form of fragments into one large chunk of free memory, which can be used to run other processes.

It does that by moving all the processes towards one end of the memory and all the available free space towards the other end of the memory so that it becomes contiguous.

It is not always easy to do compaction. Compaction can be done only when the relocation is dynamic and done at execution time. Compaction can not be done when relocation is static and is performed at load time or assembly time.

Before Compaction

Before compaction, the main memory has some free space between occupied space. This condition is known as external fragmentation. Due to less free space between occupied spaces, large processes cannot be loaded into them.

After Compaction

After compaction, all the occupied space has been moved up and the free space at the bottom. This makes the space contiguous and removes external fragmentation. Processes with large memory requirements can be now loaded into the main memory.

Purpose of Compaction in Operating System

While allocating memory to process, the operating system often faces a problem when there's a sufficient amount of free space within the memory to satisfy the memory demand of a process. however the process's memory request can't be fulfilled because the free memory available is in a non-contiguous manner, this problem is referred to as external fragmentation. To solve such kinds of problems compaction technique is used.

Advantages of Compaction

- Reduces external fragmentation.
- Make memory usage efficient.
- Memory becomes contiguous.
- Since memory becomes contiguous more processes can be loaded to memory.

Disadvantages of Compaction

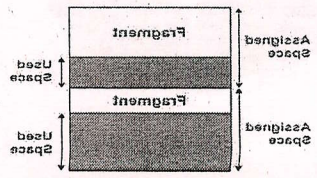
- System efficiency reduces.
- A huge amount of time is wasted in performing compaction.
- CPU sits idle for a long time.
- Not always easy to perform compaction.

Fragmentation:

"Fragmentation is a process of data storage in which memory space is used inadequately, decreasing ability or efficiency and sometimes both."

Internal Fragmentation:

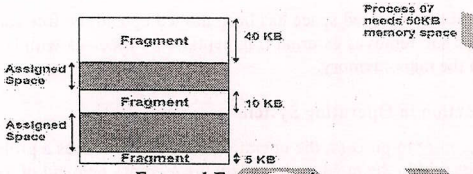
Internal fragmentation happens when the memory is split into mounted-sized blocks.
Whenever a method is requested for the memory, the mounted-sized block is allotted to the method. In the case where the memory allotted to the method is somewhat larger than the memory requested, then the difference between allotted and requested memory is called internal fragmentation. We fixed the sizes of the memory blocks, which has caused this issue. If we use dynamic partitioning to allot space to the process, this issue can be solved.



Internal Fragmentation

The above diagram clearly shows the internal fragmentation because the difference between memory allocated and required space or memory is called Internal fragmentation.

1. **External Fragmentation:**
   External fragmentation happens when there's a sufficient quantity of area within the memory to satisfy the memory request of a method. However, the process's memory request cannot be fulfilled because the memory offered is in a non-contiguous manner. Whether you apply a first-fit or best-fit memory allocation strategy
   it'll cause external fragmentation.



External Fragmentation

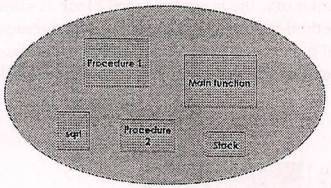
In the above diagram, we can see that, there is enough space (55 KB) to run a process-07 (required 50 KB) but the memory (fragment) is not contiguous. Here, we use compaction, paging, or segmentation to use the free space to run a process.

Segmentation in OS

In Operating Systems, Segmentation is a memory management technique in which the memory is divided into the variable size parts. Each part is known as a segment which can be allocated to a process. Segmentation is a memory management technique whereby data items are stored in segments on the storage media. It divides the process or user-accessible space into fixed-sized blocks, called **segments**.

Segmentation is a **memory management technique** that splits up the virtual address space of an application into chunks.

By splitting the memory up into manageable chunks, the operating system can track which parts of the memory are in use and which parts are free. This makes allocating and deallocating memory much faster and simpler for the operating system. The segments are of unequal size and are not placed in a contiguous way. As it's a non-contiguous memory allocation technique, internal fragmentation doesn't occur. The length is decided on the base of the purpose of the segment in a user program.



Why is Segmentation technique required?

Segmentation is a memory management technique which divides the program from the user's view of memory. That means the program is divided into modules/segments, unlike paging in which the program was divided into different pages, and those pages may or may not be loaded into the memory simultaneously. Segmentation prevents internal fragmentation.

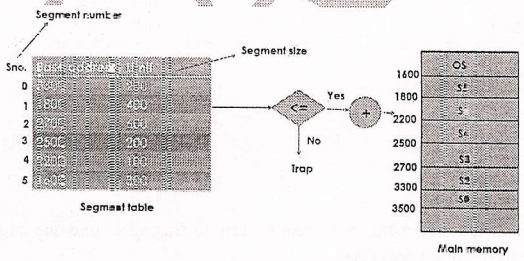
Advantages of using segmentation technique

- It can improve the system's efficiency by allowing different kernel parts to run on separate processor cores.
- It can also improve the system's responsiveness by allowing different threads to run in parallel.
- Better CPU utilization
- Provide a solution to internal fragmentation.
- The segment table is there for storing records of segments. This segment table also consumes some memory to get stored.
- Security procedures can be separated from data.

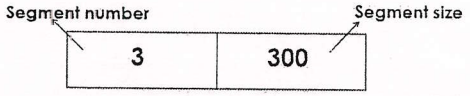
Disadvantages of using segmentation technique

- It can increase the cost of system performance. This is because the kernel must be able to detect when a segment is being used and allocate new memory space for it.
- The system may require more resources to manage Segmentation than if it used a more straightforward memory management technique.
- It suffers from external fragmentation.
- Overhead of maintaining a segment table for each process.

Segmentation Example



Suppose the CPU wants to access segment 3. It will generate a logical address(as shown below) specifying the segment number and size of the segment it wants to read. The segment size is supposed 300. This means the CPU wants to read 300 instructions from segment 3. So after consulting the segment table, the CPU comes to know that the base address is 2500. But the size of the segment is 200 instructions. So here **offset(d)>Limit**. So this will generate an error.



Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. The process of retrieving processes in the form of pages from the secondary storage into the main memory is known as paging. The basic purpose of paging is to separate each procedure into pages.

Paging

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. The process of retrieving processes in the form of pages from the secondary storage into the main memory is known as paging. The basic purpose of paging is to separate each procedure into pages.

Features of paging:

1. Mapping logical address to physical address.
2. Page size is equal to frame size.
3. Number of entries in a page table is equal to number of pages in logical address space.
4. The page table entry contains the frame number.
5. All the page table of the processes are placed in main memory.

Example:

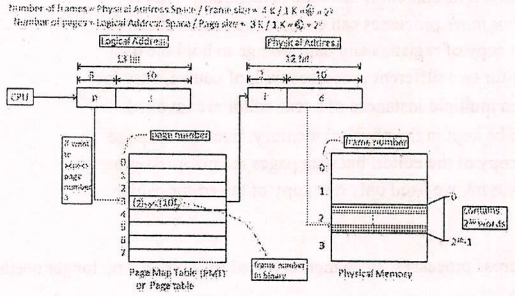
- If Logical Address = 31 bit, then Logical Address Space = 2^{31} words = 2 G words (1 G = 2^{30})
- If Logical Address Space = 128 M words = $2^7 * 2^{20}$ words, then Logical Address = $\log_2 2^{27}$ = 27 bits
- If Physical Address = 22 bit, then Physical Address Space = 2^{22} words = 4 M words (1 M = 2^{20})
- If Physical Address Space = 16 M words = $2^4 * 2^{20}$ words, then Physical Address = $\log_2 2^{24}$ = 24 bits

The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as paging technique.

- The Physical Address Space is conceptually divided into a number of fixed-size blocks, called **frames**.
- The Logical address Space is also splitted into fixed-size blocks, called **pages**.
- Page Size = Frame Size

Let us consider an example:

- Physical Address = 12 bits, then Physical Address Space = 4 K words
- Logical Address = 13 bits, then Logical Address Space = 8 K words
- Page size = frame size = 1 K words (assumption)



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.

The hardware implementation of page table can be done by using dedicated registers. But the usage of register for the page table is satisfactory only if page table is small. If page table contain large number of entries then we can use TLB(translation Look-aside buffer), a special, small, fast look up hardware cache.

- The TLB is associative, high speed memory.
- Each entry in TLB consists of two parts: a tag and a value.
- When this memory is used, then an item is compared with all tags simultaneously. If the item is found, then corresponding value is returned.

Concept of dirty bit

A **dirty bit** is a bit in memory switched on when an update is made to a page by computer hardware. When the dirty bit is switched on, the page is modified and can be replaced in memory. If it is off, no replacement is necessary since no updates have been made.

Shared pages and reentrant code

A computer's memory paging function stores and retrieves data from secondary storage for use in its main memory, which is called a memory management scheme. A page is a block of data retrieved from secondary storage by the operating system.

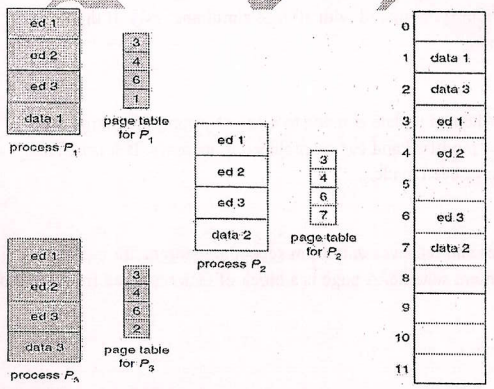
Sharing in Paging

Another advantage of paging is the possibility of sharing common code. Reentrant (read-only) code pages of a process address can be shared. If the code is reentrant, it never changes during execution. Thus two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process' execution. The data for two different processes will, of course, vary for each process. Consider the case when multiple instances of a text editor are invoked. Only one copy of the editor needs to be kept in the physical memory. Each user's page table maps on to the same physical copy of the editor, but data pages are mapped onto different frames. Thus to support 40 users, we need only one copy of the editor, which results in saving total space.

Paging allows sharing of memory across processes, since memory used by a process no longer needs to be contiguous.

- Shared code must be reentrant,that means the processes that are using it cannot change it (e.g., no data in reentrant code).
- Sharing of pages is similar to the way threads share text and memory with each other.
- A shared page may exist in different parts of the virtual address space of each process, but the virtual addresses map to the same physical address.
- The user program (e.g., emacs) marks text segment of a program as reentrant with a system call.
- The OS keeps track of available reentrant code in memory and reuses them if a new process requests the same program.
- Can greatly reduce overall memory requirements for commonly used applications.

Shared Pages Example



Reentrant code is non-self modifying code; it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for different processes will be different.
Only one copy of the editor need to be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Other heavily used programs such as compilers, window systems, run – time libraries, database systems can also be shared. To be sharable, the code must be reentrant.

Virtual Memory

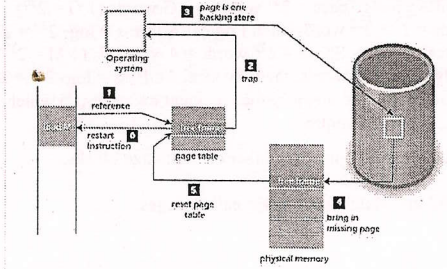
Whenever some pages needs to be loaded in the main memory for the execution and the memory is not available for those many pages, then in that case, instead of stopping the pages from entering in the main memory, the OS search for the RAM area that are least used in the recent times or that are not referenced and copy that into the secondary memory to make the space for the new pages in the main memory.

Demand paging

Demand Paging is a popular method of virtual memory management. In demand paging, the pages of a process which are least used, get stored in the secondary memory.
A page is copied to the main memory when its demand is made or page fault occurs. There are various page replacement algorithms which are used to determine the pages which will be replaced.

Page Fault

A page fault will happen if a program tries to access a piece of memory that does not exist in physical memory (main memory). The fault specifies the operating system to trace all data into virtual memory management and then relocate it from secondary memory to its primary memory, such as a hard disk.



Page replacement Algorithm

Page Replacement Algorithm is used when a page fault occurs. Page Fault means the page referenced by the CPU is not present in the main memory.
When the CPU generates the reference of a page, if there is any vacant frame available in the main memory then the page is loaded in that vacant frame. In another case, if there is no vacant frame available in the main memory, it is required to replace one of the pages in the main memory with the page referenced by the CPU.

Page Replacement Algorithm is used to decide which page will be replaced to allocate memory to the current referenced page.

1. FIFO Page Replacement Algorithm

- As the name suggests, this algorithm works on the principle of **"First in First out"**.
- It replaces the oldest page that has been present in the main memory for the longest time.
- It is implemented by keeping track of all the pages in a queue.

2. LIFO Page Replacement Algorithm

- As the name suggests, this algorithm works on the principle of **"Last in First out"**.
- It replaces the newest page that arrived at last in the main memory.
- It is implemented by keeping track of all the pages in a stack.

3. LRU Page Replacement Algorithm

- As the name suggests, this algorithm works on the principle of **"Least Recently Used"**.
- It replaces the page that has not been referred by the CPU for the longest time.

4. Optimal Page Replacement Algorithm

- This algorithm replaces the page that will not be referred by the CPU in future for the longest time.
- It is practically impossible to implement this algorithm.
- This is because the pages that will not be used in future for the longest time can not be
- predicted.
- However, it is the best known algorithm and gives the least number of page faults.
- Hence, it is used as a performance measure criterion for other algorithms.

5. Random Page Replacement Algorithm

- As the name suggests, this algorithm randomly replaces any page.
- So, this algorithm may behave like any other algorithm like FIFO, LIFO, LRU, Optimal etc.

Q.Example: Consider the Pages referenced by the CPU in the order are 6, 7, 8, 9, 6, 7, 1, 6, 7, 8, 9, 1

Solve this example with FIFO page replacement algorithm

Pages >>	6	7	8	9	6	7	1	6	7	8	9	1
Frame 3			8	9	9	7	7	7	7	7	9	9
Frame 2		7	7	9	6	6	6	6	6	8	8	8
Frame 1	6	6	6	9	9	9	1	1	1	1	1	1
	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Hit

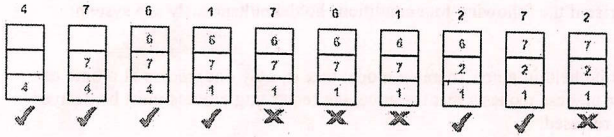
Q.system uses 3 page frames for storing process pages in main memory. It uses the Least Recently Used (LRU) page replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below-

4 , 7, 6, 1, 7, 6, 1, 2, 7, 2

Also calculate the hit ratio and miss ratio.

Solution-

Total number of references = 10



From here,

Total number of page faults occurred = 6

In the similar manner as above-

- Hit ratio = 0.4 or 40%
- Miss ratio = 0.6 or 60%

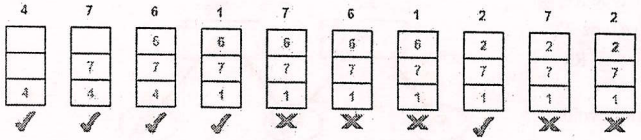
Q.A system uses 3 page frames for storing process pages in main memory. It uses the Optimal page replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below-

4 , 7, 6, 1, 7, 6, 1, 2, 7, 2

Also calculate the hit ratio and miss ratio.

Solution-

Total number of references = 10



From here,

Total number of page faults occurred = 5

In the similar manner as above-

- Hit ratio = 0.5 or 50%
- Miss ratio = 0.5 or 50%

Deadlocks

In a multi programming environment, several processes may compete for a finite number of resources. A process requests resources and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state because the resources it has requested are held by other waiting processes. This situation is called a dead lock.

Necessary Conditions

A deadlock situation can arise if the following four conditions holdsimultaneously in a system:

1. Mutual exclusion.

At least one resource must be held in a non sharable mode; that is, only oneprocess at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and wait.

A process must be holding at least one resource and waiting to acquireadditional resources that are currently being held by other processes

3. No preemption.

Resources cannot be preempted that is, a resource can be released only voluntarilyby the process holding it, after that process has completed its task. /.2 Deadlock Characterization 249

4. Circular wait.

A set {P1, P2, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P2 is waiting for a resource held by P3, •••, Pn-1 is waiting for aresource held by Pn, and P1 is waiting for a resource held by Pn.

Resource-Allocation Graph

The sets P, R, and £:

P={P1,P2,P3}

R= {R1, R2,R3, R4}

£ = {p1 -> R1, P2 _> R3 R1 -> p2, R2 -> P2, R2 -> p1, R3 -> P3 }

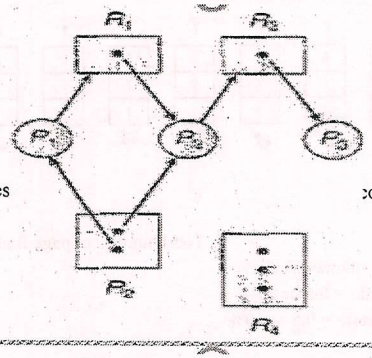
Resource instances:

One instance of resource type R1

Two instances of resource type R2

One instance of resource type R3

Three instances of R4



Deadlock prevention and avoidance

Deadlock prevention

As we are already familiar with all the neces conditions are as follows:

- Mutual Exclusion
- Hold and Wait
- No preemption
- Circular Wait

Deadlock Prevention in Operating System

let us take an example of a chair, as we know that chair always stands on its four legs. Likewise, for the deadlock problem, all the above given four conditions are needed. If anyone leg of the chair gets broken, then definitely it will fall. The same is the situation with the deadlock if we become able to violate any condition among the four and do not let them occur together then there can be prevented from the deadlock problem.

We will elaborate deadlock prevention approach by examining each of the four necessary conditions separately.

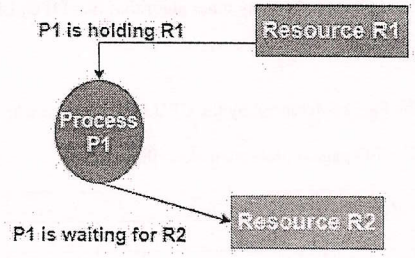
Mutual Exclusion

This condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. In contrast, Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock. A good example of a sharable resource is Read-only files because if several processes attempt to open a read-only file at the same time, then they can be granted simultaneous access to the file.

A process need not to wait for the sharable resource. Generally, deadlocks cannot be prevented by denying the mutual exclusion condition because there are some resources that are intrinsically non-sharable.

Hold and Wait

Hold and wait condition occurs when a process holds a resource and is also waiting for some other resource in order to complete its execution. Thus if we did not want the occurrence of this condition then we must guarantee that when a process requests a resource, it does not hold any other resource.



Hold and wait condition

There are some protocols that can be used in order to ensure that the Hold and Wait condition never occurs:

- According to the first protocol; Each process must request and gets all its resources before the begining of its execution.
- The second protocol allows a process to request resources only when it does not occupy any resource.

Let us illustrate the difference between these two protocols:

We will consider a process that mainly copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all the resources must be requested at the beginning of the process according to the first protocol, then the process requests the DVD drive, disk file, and printer initially. It will hold the printer during its entire execution, even though the printer is needed only at the end. While the second method allows the process to request initially only the DVD drive and disk file. It copies the data from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and printer. After copying the disk file to the printer, the process releases these two resources as well and then terminates.

Deadlock Avoidance in Operating System

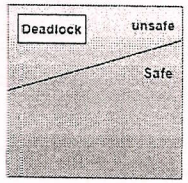
The deadlock Avoidance method is used by the operating system in order to check whether the system is in a safe state or in an unsafe state and in order to avoid the deadlocks, the process must need to tell the operating system about the maximum number of resources a process can request in order to complete its execution.

How does Deadlock Avoidance work?

In this method, the request for any resource will be granted only if the resulting state of the system doesn't cause any deadlock in the system. This method checks every step performed by the operating system. Any process continues its execution until the system is in a safe state. Once the system enters into an unsafe state, the operating system has to take a step back.
With the help of a deadlock-avoidance algorithm, you can dynamically assess the resource-allocation state so that there can never be a circular-wait situation.
According to the simplest and useful approach, any process should declare the maximum number of resources of each type it will need. The algorithms of deadlock avoidance mainly examine the resource allocations so
that there can never be an occurrence of circular wait conditions.
Deadlock avoidance can mainly be done with the help of Banker's Algorithm.

Safe State and Unsafe State

A state is safe if the system can allocate resources to each process(up to its maximum requirement) in some order and still avoid a deadlock. Formally, a system is in a safe state only, if there exists a safe sequence. So a safe state is not a deadlocked state and conversely a deadlocked state is an unsafe state.In an Unsafe state, the operating system cannot prevent processes from requesting resources in such a way that any deadlock occurs. It is not necessary that all unsafe states are deadlocks; an unsafe state may lead to a deadlock.



The above Figure shows the Safe, unsafe, and deadlocked state spaces

Processes	Maximum Needs	Current Needs
P1	10	5
P2	4	2
P3	9	2

Deadlock Avoidance Example

Let us consider a system having 12 magnetic tapes and three processes P1, P2, P3. Process P1 requires 10 magnetic tapes, process P2 may need as many as 4 tapes, process P3 may need up to 9 tapes. Suppose at a time to, process P1 is holding 5 tapes, process P2 is holding 2 tapes and process P3 is holding 2 tapes. (There are 3 free magnetic tapes)

So at time t0, the system is in a safe state. The sequence is <P2,P1,P3> satisfies the safety condition. Process P2 can immediately be allocated all its tape drives and then return them. After the return the system will have 5 available tapes, then process P1 can get all its tapes and return them (the system will then have 10 tapes); finally, process P3 can get all its tapes and return them (The system will then have 12 available tapes).
A system can go from a safe state to an unsafe state. Suppose at time t1, process P3 requests and is allocated one more tape. The system is no longer in a safe state. At this point, only process P2 can be allocated all its tapes. When it returns them the system will then have only 4 available tapes. Since P1 is allocated five tapes but has a maximum of ten so it may request 5 more tapes. If it does so, it will have to wait because they are unavailable. Similarly, process P3 may request its additional 6 tapes and have to wait which then results in a deadlock.

The mistake was granting the request from P3 for one more tape. If we made P3 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock

Note: In a case, if the system is unable to fulfill the request of all processes then the state of the system is called unsafe.
The main key of the deadlock avoidance method is whenever the request is made for resources then the request must only be approved only in the case if the **resulting state is a safe state.**

Semaphores

Semaphores are integer variables that are used to solve the critical section problem by using two *atomic operations, wait and signal that are used for process synchronization.*
The definitions of wait and signal are as follows –

- **Wait**
  The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

```
wait(A)
{
while (A<=0);
A–;
}
```

- **Signal**
  The signal operation increments the value of its argument S.

```
signal(A)
{
A++;
}
```

Types of Semaphores
There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows –

- **Counting Semaphores**
  These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.
- **Binary Semaphores**
  The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

Advantages of Semaphores
Some of the advantages of semaphores are as follows –

- Semaphores allow only one process into the critical section. They follow the mutual exclusion
- principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are
- machine independent.

Disadvantages of Semaphores
Some of the disadvantages of semaphores are as follows –

- Semaphores are complicated so the wait and signal operations must be implemented in the correct
- order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical
- section first and high priority processes later.

Mutex in Operating System
Mutex lock in OS is essentially a variable that is binary nature that provides code wise functionality for mutual exclusion. At times, there may be multiple threads that may be trying to access same resource like memory or I/O etc. To make sure that there is no overriding. Mutex provides a locking mechanism.
Only one thread at a time can take the ownership of a mutex and apply the lock. Once it done utilising the resource and it may release the mutex lock.

Mutex

Mutex is a mutual exclusion object that synchronizes access to a resource. It is created with a unique name at the start of a program. The Mutex is a locking mechanism that makes sure only one thread can acquire the Mutex at a time and enter the critical section. This thread only releases the Mutex when it exits the critical section.
This is shown with the help of the following example –

```
wait (mutex);
.....
Critical Section
.....
signal (mutex);
```

A Mutex is different than a semaphore as it is a locking mechanism while a semaphore is a signalling mechanism. A binary semaphore can be used as a Mutex but a Mutex can never be used as a semaphore.

Producer consumer problem

What is Producer-Consumer Problem?
The Producer-Consumer problem is a classical problem. The Producer-Consumer problem is used for multi-process synchronization, which means synchronization between more than one processes.
In this problem, we have one producer and one consumer. The producer is the one who produces something, and the consumer is the one who consumes something, produced by the producer. The producer and consumer both share the common memory buffer, and the memory buffer is of fixed-size.
The task performed by the producer is to generate the data, and when the data gets generated, and then it put the data into the buffer and again generates the data. The task performed by the consumer is to consume the data which is present in the memory buffer.

What are the Problems in the Producer-Consumer Problem?
There are various types of problems in the Producer-Consumer problem:

1. At the same time, the producer and consumer cannot access the buffer.
2. The producer cannot produce the data if the memory buffer is full. It means when the memory buffer is not full, then only the producer can produce the data.
3. The consumer can only consume the data if the memory buffer is not vacant. In a condition where memory buffer is empty, the consumer is not allowed to take data from the memory buffer.

What is the solution for the Producer-Consumer Problem?
There are three semaphore variables that we use to solve the problems that occur in the Producer-Consumer problem.

1. Semaphore S
2. Semaphore E
3. Semaphore F

Semaphore S: - With the help of the Semaphore 'S' variable, we can achieve mutual exclusion among the processes. By using the Semaphore variable 'S,' the producer or consumer can access and use the shared buffer at a specific time. Initially, the value of the Semaphore variable 'S' is set to 1.
Semaphore E: - With the help of the Semaphore 'E' variable, we can define the vacant (empty) space in the memory buffer. Initially, the value of the Semaphore 'E' variable is set to 'n' because initially, the memory buffer is empty.

Semaphore F: - We use Semaphore 'F' variable to define the filled space, which is filled by the producer. Firstly, we set the value of Semaphore variable 'F' to 0 because in starting, no space is filled by the producer.
With the help of these Semaphore variables, we can solve the problems that occur in the Producer-Consumer problem. We can also use two types of functions to solve this problem, and the functions are wait() and signal().

1. wait(): - By using wait() function, we can decrease the value of the Semaphore variable by 1.
2. signal(): - By using signal() function, the value of the Semaphore variable is incremented by 1.

Below is the pseudocode for the producer:

```
void producer( ) {
   while(T) {
        produce( )
        wait(E)
        wait(S)
        append()
        signal(S)
        signal(F)
   }
}
```


Explanation of the above code

1. while(): - We used while() to produce the data again and again.
2. produce(): - We called the produce() function to tell producer to produce the data.
3. wait(E): - With the help wait() function, the value of the Semaphore variable 'E' can be decremented
4. by 1. If a producer produces something, then we have to decrease the value of the Semaphore variable 'E' by 1.
5. wait(S): - The wait(S) function is used to set the value of the Semaphore variable 'S' to '0', so that other processes cannot enter into the critical section.
6. append(): - By using append() function, new data is added in the memory buffer.
7. signal(S): - We used the signal(S) function to set the value of the Semaphore variable 'S' to 1, so that another process can enter into the critical section.
8. signal(F): - By using the signal(F) function, the value of the Semaphore variable 'F' is incremented by one. In this, we increment the value by 1 because if we add the data into the memory buffer, there is one space that is filled in the memory buffer. So, we have to update the variable 'F'.

```
void consumer( ) {
   while(T) {
      wait(F)
      wait(S)
      take( )
      signal(S)
      use( )
   }
}
```

Below is the pseudocode for the consumer:
Explanation of the above code

1. while(): - By using while(), the data can be consumed again and again.
2. wait(F): - We used wait(F) function to decrease the value of the Semaphore variable 'F' by 1. It is because if the consumer consumes some data, we have to reduce the value of the Semaphore variable 'F' by 1.
3. wait(S): - The wait(S) function is used to set the value of the Semaphore variable 'S' to '0', so that
4. other processes cannot enter into the critical section.
5. take():- We used take() function to take the data from the memory buffer by the consumer.
6. signal(S): - We used the signal(S) function to set the value of the Semaphore variable 'S' to 1, so that other processes can enter into the critical section.
7. signal(E): - The signal(E) function is used to increment the value of the Semaphore variable 'E' by 1.
8. It is because after taking data from the memory buffer, space is freed from the buffer, and it is must to increase the value of the Semaphore variable 'E'.
9. use(): - By using the use() function, we can use the data taken from the memory buffer, so that we can perform some operations.

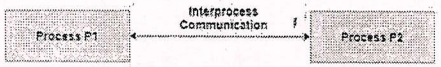
Dead-lock vs Starvation

Difference between Deadlock and Starvation:

S.NO	Deadlock	Starvation
1.	All processes keep waiting for each other to complete and none get executed	High priority processes keep executing and low priority processes are blocked
2.	Resources are blocked by the processes	Resources are continuously utilized by high priority processes
3.	Necessary conditions Mutual Exclusion, Hold and Wait, No preemption, Circular Wait	Priorities are assigned to the processes
4.	Also known as Circular wait	Also known as lived lock
5.	It can be prevented by avoiding the necessary conditions for deadlock	It can be prevented by Aging

Inter process Communication

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



Approaches to Inter process Communication

- **Message Queue**

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

- **Shared Memory**

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

- **Pipe**

  A pipe is a data channel that is unidirectional. Two pipes can be used to create a two-way data channel between two processes. This uses standard input and output methods. Pipes are used in all POSIX systems as well as Windows operating systems.

- **FIFO**

  Used to communicate between two processes that are not related.Full-duplex method - Process P1 is able to communicate with Process P2, and vice versa.